Chapter 4

THE GREEDY METHOD

4.1 THE GENERAL METHOD

The greedy method is perhaps the most straightforward design technique we consider in this text, and what's more it can be applied to a wide variety of problems. Most, though not all, of these problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a *feasible* solution. We need to find a feasible solution that either maximizes or minimizes a given *objective function.* A feasible solution that does this is called an *optimal solution.* There is usually an obvious way to determine a feasible solution but not necessarily an optimal solution.

The greedy method suggests that one can devise an algorithm that works in stages, considering one input at a time. At each stage, a decision is made regarding whether a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input into the partially constructed optimal solution will result in an infeasible solution, then this input is not added to the partial solution. Otherwise, it is added. The selection procedure itself is based on some optimization measure. This measure may be the objective function. In fact, several different optimization measures may be plausible for a given problem. Most of these, however, will result in algorithms that generate suboptimal solutions. This version of the greedy technique is called the *subset paradigm.*

We can describe the subset paradigm abstractly, but more precisely than above, by considering the control abstraction in Algorithm 4.1.

The function Select selects an input from $a[\,]$ and removes it. The selected input's value is assigned to x. Feasible is a Boolean-valued function that determines whether x can be included into the solution vector. The function Union combines x with the solution and updates the objective function. The

```
1   Algorithm Greedy(a, n)
2   // a[1 : n] contains the n inputs.
3   {
4       solution := ∅; // Initialize the solution.
5       for i := 1 to n do
6       {
7           x := Select(a);
8           if Feasible(solution, x) then
9               solution := Union(solution, x);
10      }
11      return solution;
12  }
```

Algorithm 4.1 Greedy method control abstraction for the subset paradigm

function Greedy describes the essential way that a greedy algorithm will look, once a particular problem is chosen and the functions Select, Feasible, and Union are properly implemented.

For problems that do not call for the selection of an optimal subset, in the greedy method we make decisions by considering the inputs in some order. Each decision is made using an optimization criterion that can be computed using decisions already made. Call this version of the greedy method the *ordering paradigm.* Sections 4.2, 4.3, 4.4, and 4.5 consider problems that fit the subset paradigm, and Sections 4.6, 4.7, and 4.8 consider problems that fit the ordering paradigm.

EXERCISE

1. Write a control abstraction for the ordering paradigm.

4.2 KNAPSACK PROBLEM

Let us try to apply the greedy method to solve the knapsack problem. We are given n objects and a knapsack or bag. Object i has a weight w_i and the knapsack has a capacity m. If a fraction x_i, $0 \leq x_i \leq 1$, of object i is placed into the knapsack, then a profit of $p_i x_i$ is earned. The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is m, we require the total weight of all chosen objects to be at most m. Formally, the problem can be stated as

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i \tag{4.1}$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le m \tag{4.2}$$

$$\text{and } 0 \le x_i \le 1, \quad 1 \le i \le n \tag{4.3}$$

The profits and weights are positive numbers.

A feasible solution (or filling) is any set $(x_1, \ldots, x_n)$ satisfying (4.2) and (4.3) above. An optimal solution is a feasible solution for which (4.1) is maximized.

Example 4.1 Consider the following instance of the knapsack problem: $n = 3, m = 20, (p_1, p_2, p_3) = (25, 24, 15)$, and $(w_1, w_2, w_3) = (18, 15, 10)$. Four feasible solutions are:

	(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
1.	(1/2, 1/3, 1/4)	16.5	24.25
2.	(1, 2/15, 0)	20	28.2
3.	(0, 2/3, 1)	20	31
4.	(0, 1, 1/2)	20	31.5

Of these four feasible solutions, solution 4 yields the maximum profit. As we shall soon see, this solution is optimal for the given problem instance. □

Lemma 4.1 In case the sum of all the weights is $\le m$, then $x_i = 1$, $1 \le i \le n$ is an optimal solution. □

So let us assume the sum of weights exceeds m. Now all the x_i's cannot be 1. Another observation to make is:

Lemma 4.2 All optimal solutions will fill the knapsack exactly. □

Lemma 4.2 is true because we can always increase the contribution of some object i by a fractional amount until the total weight is exactly m.

Note that the knapsack problem calls for selecting a subset of the objects and hence fits the subset paradigm. In addition to selecting a subset, the knapsack problem also involves the selection of an x_i for each object. Several simple greedy strategies to obtain feasible solutions whose sums are identically m suggest themselves. First, we can try to fill the knapsack by including next the object with largest profit. If an object under consideration doesn't fit, then a fraction of it is included to fill the knapsack. Thus each time an object is included (except possibly when the last object is included)

into the knapsack, we obtain the largest possible increase in profit value. Note that if only a fraction of the last object is included, then it may be possible to get a bigger increase by using a different object. For example, if we have two units of space left and two objects with ($p_i = 4$, $w_i = 4$) and ($p_j = 3, w_j = 2$) remaining, then using j is better than using half of i. Let us use this selection strategy on the data of Example 4.1.

Object one has the largest profit value ($p_1 = 25$). So it is placed into the knapsack first. Then $x_1 = 1$ and a profit of 25 is earned. Only 2 units of knapsack capacity are left. Object two has the next largest profit ($p_2 = 24$). However, $w_2 = 15$ and it doesn't fit into the knapsack. Using $x_2 = 2/15$ fills the knapsack exactly with part of object 2 and the value of the resulting solution is 28.2. This is solution 2 and it is readily seen to be suboptimal. The method used to obtain this solution is termed a greedy method because at each step (except possibly the last one) we chose to introduce that object which would increase the objective function value the most. However, this greedy method did not yield an optimal solution. Note that even if we change the above strategy so that in the last step the objective function increases by as much as possible, an optimal solution is not obtained for Example 4.1.

We can formulate at least two other greedy approaches attempting to obtain optimal solutions. From the preceding example, we note that considering objects in order of nonincreasing profit values does not yield an optimal solution because even though the objective function value takes on large increases at each step, the number of steps is few as the knapsack capacity is used up at a rapid rate. So, let us try to be greedy with capacity and use it up as slowly as possible. This requires us to consider the objects in order of nondecreasing weights w_i. Using Example 4.1, solution 3 results. This too is suboptimal. This time, even though capacity is used slowly, profits aren't coming in rapidly enough.

Thus, our next attempt is an algorithm that strives to achieve a balance between the rate at which profit increases and the rate at which capacity is used. At each step we include that object which has the maximum profit per unit of capacity used. This means that objects are considered in order of the ratio p_i/w_i. Solution 4 of Example 4.1 is produced by this strategy. If the objects have already been sorted into nonincreasing order of p_i/w_i, then function GreedyKnapsack (Algorithm 4.2) obtains solutions corresponding to this strategy. Note that solutions corresponding to the first two strategies can be obtained using this algorithm if the objects are initially in the appropriate order. Disregarding the time to initially sort the objects, each of the three strategies outlined above requires only $O(n)$ time.

We have seen that when one applies the greedy method to the solution of the knapsack problem, there are at least three different measures one can attempt to optimize when determining which object to include next. These measures are total profit, capacity used, and the ratio of accumulated profit to capacity used. Once an optimization measure has been chosen, the greedy

2. [0/1 Knapsack] Consider the knapsack problem discussed in this section. We add the requirement that $x_i = 1$ or $x_i = 0$, $1 \leq i \leq n$; that is, an object is either included or not included into the knapsack. We wish to solve the problem

$$\max \sum_{1}^{n} p_i x_i$$

$$\text{subject to} \sum_{1}^{n} w_i x_i \leq m$$

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

One greedy strategy is to consider the objects in order of nonincreasing density p_i/w_i and add the object into the knapsack if it fits. Show that this strategy doesn't necessarily yield an optimal solution.

4.3 TREE VERTEX SPLITTING

Consider a directed binary tree each edge of which is labeled with a real number (called its *weight*). Trees with edge weights are called *weighted trees*. A weighted tree can be used, for example, to model a distribution network in which electric signals or commodities such as oil are transmitted. Nodes in the tree correspond to receiving stations and edges correspond to transmission lines. It is conceivable that in the process of transmission some loss occurs (drop in voltage in the case of electric signals or drop in pressure in the case of oil). Each edge in the tree is labeled with the loss that occurs in traversing that edge. The network may not be able to tolerate losses beyond a certain level. In places where the loss exceeds the tolerance level, boosters have to be placed. Given a network and a loss tolerance level, the *Tree Vertex Splitting Problem (TVSP)* is to determine an optimal placement of boosters. It is assumed that the boosters can only be placed in the nodes of the tree.

The TVSP can be specified more precisely as follows: Let $T = (V, E, w)$ be a weighted directed tree, where V is the vertex set, E is the edge set, and w is the weight function for the edges. In particular, $w(i, j)$ is the weight of the edge $\langle i, j \rangle \in E$. The weight $w(i, j)$ is undefined for any $\langle i, j \rangle \notin E$. A *source vertex* is a vertex with in-degree zero, and a *sink vertex* is a vertex with out-degree zero. For any path P in the tree, its *delay*, $d(P)$, is defined to be the sum of the weights on that path. The delay of the tree T, $d(T)$, is the maximum of all the path delays.

Let T/X be the forest that results when each vertex u in X is split into two nodes u^i and u^o such that all the edges $\langle u, j \rangle \in E$ ($\langle j, u \rangle \in E$) are

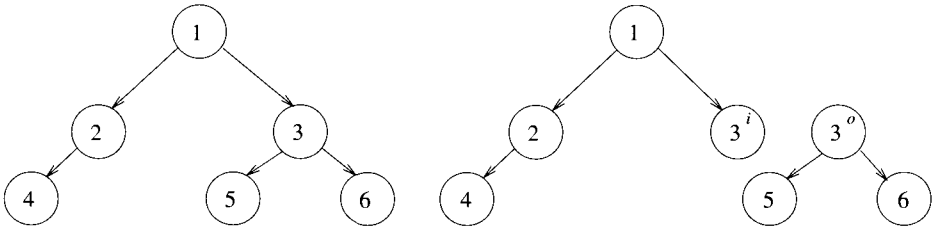


Figure 4.1 A tree before and after splitting the node 3

replaced by edges of the form $\langle u^o, j\rangle$ ($\langle j, u^i\rangle$). In other words, outbound edges from u now leave from u^o and inbound edges to u now enter at u^i. Figure 4.1 shows a tree before and after splitting the node 3. A node that gets split corresponds to a booster station. The TVSP is to identify a set $X \subseteq V$ of minimum cardinality for which $d(T/X) \leq \delta$, for some specified tolerance limit δ. Note that the TVSP has a solution only if the maximum edge weight is $\leq \delta$. Also note that the TVSP naturally fits the subset paradigm.

Given a weighted tree $T(V, E, w)$ and a tolerance limit δ, any subset X of V is a feasible solution if $d(T/X) \leq \delta$. Given an X, we can compute $d(T/X)$ in $O(|V|)$ time. A trivial way of solving the TVSP is to compute $d(T/X)$ for each possible subset X of V. But there are $2^{|V|}$ such subsets! A better algorithm can be obtained using the greedy method.

For the TVSP, the quantity that is optimized (minimized) is the number of nodes in X. A greedy approach to solving this problem is to compute for each node $u \in V$, the maximum delay $d(u)$ from u to any other node in its subtree. If u has a parent v such that $d(u) + w(v, u) > \delta$, then the node u gets split and $d(u)$ is set to zero. Computation proceeds from the leaves toward the root.

In the tree of Figure 4.2, let $\delta = 5$. For each of the leaf nodes $7, 8, 5, 9$, and 10 the delay is zero. The delay for any node is computed only after the delays for its children have been determined. Let u be any node and $C(u)$ be the set of all children of u. Then $d(u)$ is given by

$$d(u) = \max_{v \in C(u)} \{d(v) + w(u, v)\}$$

Using the above formula, for the tree of Figure 4.2, $d(4) = 4$. Since $d(4) + w(2, 4) = 6 > \delta$, node 4 gets split. We set $d(4) = 0$. Now $d(2)$ can be

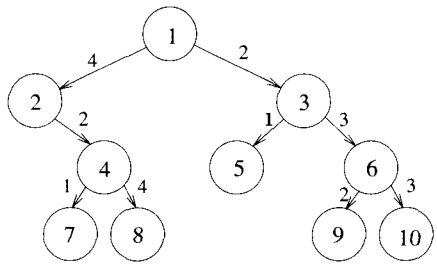


Figure 4.2 An example tree

computed and is equal to 2. Since $d(2) + w(1,2)$ exceeds δ, node 2 gets split and $d(2)$ is set to zero. Then $d(6)$ is equal to 3. Also, since $d(6)+w(3,6) > \delta$, node 6 has to be split. Set $d(6)$ to zero. Now $d(3)$ is computed as 3. Finally, $d(1)$ is computed as 5.

Figure 4.3 shows the final tree that results after splitting the nodes 2, 4, and 6. This algorithm is described in Algorithm 4.3, which is invoked as TVS($root, \delta$), *root* being the root of the tree. The order in which TVS visits (i.e., computes the delay values of) the nodes of the tree is called the *post order* and is studied again in Chapter 6.

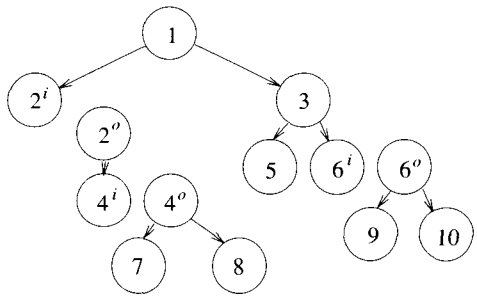


Figure 4.3 The final tree after splitting the nodes 2, 4, and 6

```
1   Algorithm TVS(T, δ)
2   // Determine and output the nodes to be split.
3   // w() is the weighting function for the edges.
4   {
5       if (T ≠ 0) then
6       {
7           d[T] := 0;
8           for each child v of T do
9           {
10              TVS(v, δ);
11              d[T] := max{d[T], d[v] + w(T, v)};
12          }
13          if ((T is not the root)  and
14                  (d[T] + w(parent(T), T) > δ)) then
15          {
16              write (T); d[T] := 0;
17          }
18      }
19  }
```

Algorithm 4.3 The tree vertex splitting algorithm

Algorithm TVS takes $\Theta(n)$ time, where n is the number of nodes in the tree. This can be seen as follows: When TVS is called on any node T, only a constant number of operations are performed (excluding the time taken for the recursive calls). Also, TVS is called only once on each node T in the tree.

Algorithm 4.4 is a revised version of Algorithm 4.3 for the special case of directed binary trees. A sequential representation of the tree (see Section 2.2) has been employed. The tree is stored in the array *tree*[ ] with the root at *tree*[1]. Edge weights are stored in the array *weight*[ ]. If *tree*[*i*] has a tree node, the weight of the incoming edge from its parent is stored in *weight*[*i*]. The delay of node i is stored in $d[i]$. The array $d[\ ]$ is initialized to zero at the beginning. Entries in the arrays *tree*[ ] and *weight*[ ] corresponding to nonexistent nodes will be zero. As an example, for the tree of Figure 4.2, *tree*[ ] will be set to $\{1, 2, 3, 0, 4, 5, 6, 0, 0, 7, 8, 0, 0, 9, 10\}$ starting at cell 1. Also, *weight*[ ] will be set to $\{0, 4, 2, 0, 2, 1, 3, 0, 0, 1, 4, 0, 0, 2, 3\}$ at the beginning, starting from cell 1. The algorithm is invoked as TVS$(1, \delta)$. Now we show that TVS (Algorithm 4.3) will always split a minimal number of nodes.

```
1   Algorithm TVS(i, δ)
2   // Determine and output a minimum cardinality split set.
3   // The tree is realized using the sequential representation.
4   // Root is at tree[1]. N is the largest number such that
5   // tree[N] has a tree node.
6   {
7       if (tree[i] ≠ 0) then // If the tree is not empty
8           if (2i > N) then d[i] := 0; // i is a leaf.
9           else
10          {
11              TVS(2i, δ);
12              d[i] := max(d[i], d[2i] + weight[2i]);
13              if (2i + 1 ≤ N) then
14              {
15                  TVS(2i + 1, δ);
16                  d[i] := max(d[i], d[2i + 1] + weight[2i + 1]);
17              }
18          }
19          if ((tree[i] ≠ 1) and (d[i] + weight[i] > δ)) then
20          {
21              write (tree[i]); d[i] := 0;
22          }
23  }
```

Algorithm 4.4 TVS for the special case of binary trees

Theorem 4.2 Algorithm TVS outputs a minimum cardinality set U such that $d(T/U) \leq \delta$ on any tree T, provided no edge of T has weight $> \delta$.

Proof: The proof is by induction on the number of nodes in the tree. If the tree has a single node, the theorem is true. Assume the theorem for all trees of size $\leq n$. We prove it for trees of size $n+1$ also.

Let T be any tree of size $n+1$ and let U be the set of nodes split by TVS. Also let W be a minimum cardinality set such that $d(T/W) \leq \delta$. We have to show that $|U| \leq |W|$. If $|U| = 0$, this is true. Otherwise, let x be the first vertex split by TVS. Let T_x be the subtree rooted at x. Let T' be the tree obtained from T by deleting T_x except for x. Note that W has to have at least one node, say y, from T_x. Let $W' = W - \{y\}$. If there is a W^* such that $|W^*| < |W'|$ and $d(T'/W^*) \leq \delta$, then since $d(T/(W^* + \{x\})) \leq \delta$, W is not a minimum cardinality split set for T. Thus, W' has to be a minimum cardinality split set such that $d(T'/W') \leq \delta$.

If algorithm TVS is run on tree T', the set of split nodes output is $U-\{x\}$. Since T' has $\leq n$ nodes, $U-\{x\}$ is a minimum cardinality split set for T'. This in turn means that $|W'| \geq |U| - 1$. In other words, $|W| \geq |U|$. □

EXERCISES

1. For the tree of Figure 4.2 solve the TVSP when (a) $\delta = 4$ and (b) $\delta = 6$.

2. Rewrite TVS (Algorithm 4.3) for general trees. Make use of pointers.

4.4 JOB SEQUENCING WITH DEADLINES

We are given a set of n jobs. Associated with job i is an integer deadline $d_i \geq 0$ and a profit $p_i > 0$. For any job i the profit p_i is earned iff the job is completed by its deadline. To complete a job, one has to process the job on a machine for one unit of time. Only one machine is available for processing jobs. A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its deadline. The value of a feasible solution J is the sum of the profits of the jobs in J, or $\sum_{i \in J} p_i$. An optimal solution is a feasible solution with maximum value. Here again, since the problem involves the identification of a subset, it fits the subset paradigm.

Example 4.2 Let $n = 4, (p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$ and $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$. The feasible solutions and their values are:

	feasible solution	processing sequence	value
1.	(1, 2)	2, 1	110
2.	(1, 3)	1, 3 or 3, 1	115
3.	(1, 4)	4, 1	127
4.	(2, 3)	2, 3	25
5.	(3, 4)	4, 3	42
6.	(1)	1	100
7.	(2)	2	10
8.	(3)	3	15
9.	(4)	4	27

Solution 3 is optimal. In this solution only jobs 1 and 4 are processed and the value is 127. These jobs must be processed in the order job 4 followed by job 1. Thus the processing of job 4 begins at time zero and that of job 1 is completed at time 2. □

To formulate a greedy algorithm to obtain an optimal solution, we must formulate an optimization measure to determine how the next job is chosen. As a first attempt we can choose the objective function $\sum_{i \in J} p_i$ as our optimization measure. Using this measure, the next job to include is the one that increases $\sum_{i \in J} p_i$ the most, subject to the constraint that the resulting J is a feasible solution. This requires us to consider jobs in nonincreasing order of the p_i's. Let us apply this criterion to the data of Example 4.2. We begin with $J = \emptyset$ and $\sum_{i \in J} p_i = 0$. Job 1 is added to J as it has the largest profit and $J = \{1\}$ is a feasible solution. Next, job 4 is considered. The solution $J = \{1, 4\}$ is also feasible. Next, job 3 is considered and discarded as $J = \{1, 3, 4\}$ is not feasible. Finally, job 2 is considered for inclusion into J. It is discarded as $J = \{1, 2, 4\}$ is not feasible. Hence, we are left with the solution $J = \{1, 4\}$ with value 127. This is the optimal solution for the given problem instance. Theorem 4.4 proves that the greedy algorithm just described always obtains an optimal solution to this sequencing problem.

Before attempting the proof, let us see how we can determine whether a given J is a feasible solution. One obvious way is to try out all possible permutations of the jobs in J and check whether the jobs in J can be processed in any one of these permutations (sequences) without violating the deadlines. For a given permutation $\sigma = i_1, i_2, i_3, \ldots, i_k$, this is easy to do, since the earliest time job $i_q, 1 \leq q \leq k$, will be completed is q. If $q > d_{i_q}$, then using σ, at least job i_q will not be completed by its deadline. However, if $|J| = i$, this requires checking $i!$ permutations. Actually, the feasibility of a set J can be determined by checking only one permutation of the jobs in J. This permutation is any one of the permutations in which jobs are ordered in nondecreasing order of deadlines.

Theorem 4.3 Let J be a set of k jobs and $\sigma = i_1, i_2, \ldots, i_k$ a permutation of jobs in J such that $d_{i_1} \leq d_{i_2} \leq \cdots \leq d_{i_k}$. Then J is a feasible solution iff the jobs in J can be processed in the order σ without violating any deadline.

Proof: Clearly, if the jobs in J can be processed in the order σ without violating any deadline, then J is a feasible solution. So, we have only to show that if J is feasible, then σ represents a possible order in which the jobs can be processed. If J is feasible, then there exists $\sigma' = r_1, r_2, \ldots, r_k$ such that $d_{r_q} \geq q$, $1 \leq q \leq k$. Assume $\sigma' \neq \sigma$. Then let a be the least index such that $r_a \neq i_a$. Let $r_b = i_a$. Clearly, $b > a$. In σ' we can interchange r_a and r_b. Since $d_{r_a} \geq d_{r_b}$, the resulting permutation $\sigma'' = s_1, s_2, \ldots, s_k$ represents an order in which the jobs can be processed without violating a deadline. Continuing in this way, σ' can be transformed into σ without violating any deadline. Hence, the theorem is proved. □

Theorem 4.3 is true even if the jobs have different processing times $t_i \geq 0$ (see the exercises).

Theorem 4.4 The greedy method described above always obtains an optimal solution to the job sequencing problem.

Proof: Let $(p_i, d_i), 1 \leq i \leq n$, define any instance of the job sequencing problem. Let I be the set of jobs selected by the greedy method. Let J be the set of jobs in an optimal solution. We now show that both I and J have the same profit values and so I is also optimal. We can assume $I \neq J$ as otherwise we have nothing to prove. Note that if $J \subset I$, then J cannot be optimal. Also, the case $I \subset J$ is ruled out by the greedy method. So, there exist jobs a and b such that $a \in I$, $a \notin J$, $b \in J$, and $b \notin I$. Let a be a highest-profit job such that $a \in I$ and $a \notin J$. It follows from the greedy method that $p_a \geq p_b$ for all jobs b that are in J but not in I. To see this, note that if $p_b > p_a$, then the greedy method would consider job b before job a and include it into I.

Now, consider feasible schedules S_I and S_J for I and J respectively. Let i be a job such that $i \in I$ and $i \in J$. Let i be scheduled from t to $t+1$ in S_I and t' to $t'+1$ in S_J. If $t < t'$, then we can interchange the job (if any) scheduled in $[t', t'+1]$ in S_I with i. If no job is scheduled in $[t', t'+1]$ in I, then i is moved to $[t', t'+1]$. The resulting schedule is also feasible. If $t' < t$, then a similar transformation can be made in S_J. In this way, we can obtain schedules S'_I and S'_J with the property that all jobs common to I and J are scheduled at the same time. Consider the interval $[t_a, t_a + 1]$ in S'_I in which the job a (defined above) is scheduled. Let b be the job (if any) scheduled in S'_J in this interval. From the choice of $a, p_a \geq p_b$. Scheduling a from t_a to $t_a + 1$ in S'_J and discarding job b gives us a feasible schedule for job set $J' = J - \{b\} \cup \{a\}$. Clearly, J' has a profit value no less than that of J and differs from I in one less job than J does.

By repeatedly using the transformation just described, J can be transformed into I with no decrease in profit value. So I must be optimal. □

A high-level description of the greedy algorithm just discussed appears as Algorithm 4.5. This algorithm constructs an optimal set J of jobs that can be processed by their due times. The selected jobs can be processed in the order given by Theorem 4.3.

Now, let us see how to represent the set J and how to carry out the test of lines 7 and 8 in Algorithm 4.5. Theorem 4.3 tells us how to determine whether all jobs in $J \cup \{i\}$ can be completed by their deadlines. We can avoid sorting the jobs in J each time by keeping the jobs in J ordered by deadlines. We can use an array $d[1:n]$ to store the deadlines of the jobs in the order of their p-values. The set J itself can be represented by a one-dimensional array $J[1:k]$ such that $J[r]$, $1 \leq r \leq k$ are the jobs in J and $d[J[1]] \leq d[J[2]] \leq \cdots \leq d[J[k]]$. To test whether $J \cup \{i\}$ is feasible, we have just to insert i into J preserving the deadline ordering and then verify that $d[J[r]] \leq r$, $1 \leq r \leq k+1$. The insertion of i into J is simplified by the use of a fictitious job 0 with $d[0] = 0$ and $J[0] = 0$. Note also that if job i is to be inserted at position q, then only the positions of jobs $J[q]$, $J[q+1]$,

```
1   Algorithm GreedyJob(d, J, n)
2   // J is a set of jobs that can be completed by their deadlines.
3   {
4       J := {1};
5       for i := 2 to n do
6       {
7           if (all jobs in J ∪ {i} can be completed
8               by their deadlines) then J := J ∪ {i};
9       }
10  }
```

Algorithm 4.5 High-level description of job sequencing algorithm

$\ldots$, $J[k]$ are changed after the insertion. Hence, it is necessary to verify only that these jobs (and also job i) do not violate their deadlines following the insertion. The algorithm that results from this discussion is function JS (Algorithm 4.6). The algorithm assumes that the jobs are already sorted such that $p_1 \geq p_2 \geq \cdots \geq p_n$. Further it assumes that $n \geq 1$ and the deadline $d[i]$ of job i is at least 1. Note that no job with $d[i] < 1$ can ever be finished by its deadline. Theorem 4.5 proves that JS is a correct implementation of the greedy strategy.

Theorem 4.5 Function JS is a correct implementation of the greedy-based method described above.

Proof: Since $d[i] \geq 1$, the job with the largest p_i will always be in the greedy solution. As the jobs are in nonincreasing order of the p_i's, line 8 in Algorithm 4.6 includes the job with largest p_i. The **for** loop of line 10 considers the remaining jobs in the order required by the greedy method described earlier. At all times, the set of jobs already included in the solution is maintained in J. If $J[i]$, $1 \leq i \leq k$, is the set already included, then J is such that $d[J[i]] \leq d[J[i+1]]$, $1 \leq i < k$. This allows for easy application of the feasibility test of Theorem 4.3. When job i is being considered, the **while** loop of line 15 determines where in J this job has to be inserted. The use of a fictitious job 0 (line 7) allows easy insertion into position 1. Let w be such that $d[J[w]] \leq d[i]$ and $d[J[q]] > d[i]$, $w < q \leq k$. If job i is included into J, then jobs $J[q]$, $w < q \leq k$, have to be moved one position up in J (line 19). From Theorem 4.3, it follows that such a move retains feasibility of J iff $d[J[q]] \neq q$, $w < q \leq k$. This condition is verified in line 15. In addition, i can be inserted at position $w+1$ iff $d[i] > w$. This is verified in line 16 (note $r = w$ on exit from the **while** loop if $d[J[q]] \neq q$, $w < q \leq k$). The correctness of JS follows from these observations. □

```
1   Algorithm JS(d, j, n)
2   // d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines, n ≥ 1. The jobs
3   // are ordered such that p[1] ≥ p[2] ≥ ··· ≥ p[n]. J[i]
4   // is the ith job in the optimal solution, 1 ≤ i ≤ k.
5   // Also, at termination d[J[i]] ≤ d[J[i + 1]], 1 ≤ i < k.
6   {
7       d[0] := J[0] := 0; // Initialize.
8       J[1] := 1; // Include job 1.
9       k := 1;
10      for i := 2 to n do
11      {
12          // Consider jobs in nonincreasing order of p[i]. Find
13          // position for i and check feasibility of insertion.
14          r := k;
15          while ((d[J[r]] > d[i]) and (d[J[r]] ≠ r)) do r := r − 1;
16          if ((d[J[r]] ≤ d[i]) and (d[i] > r)) then
17          {
18              // Insert i into J[ ].
19              for q := k to (r + 1) step −1 do J[q + 1] := J[q];
20              J[r + 1] := i; k := k + 1;
21          }
22      }
23      return k;
24  }
```

Algorithm 4.6 Greedy algorithm for sequencing unit time jobs with deadlines and profits

For JS there are two possible parameters in terms of which its complexity can be measured. We can use n, the number of jobs, and s, the number of jobs included in the solution J. The **while** loop of line 15 in Algorithm 4.6 is iterated at most k times. Each iteration takes $\Theta(1)$ time. If the conditional of line 16 is true, then lines 19 and 20 are executed. These lines require $\Theta(k - r)$ time to insert job i. Hence, the total time for each iteration of the **for** loop of line 10 is $\Theta(k)$. This loop is iterated $n - 1$ times. If s is the final value of k, that is, s is the number of jobs in the final solution, then the total time needed by algorithm JS is $\Theta(sn)$. Since $s \leq n$, the worst-case time, as a function of n alone is $\Theta(n^2)$. If we consider the job set $p_i = d_i = n - i + 1$, $1 \leq i \leq n$, then algorithm JS takes $\Theta(n^2)$ time to determine J. Hence, the worst-case computing time for JS is $\Theta(n^2)$. In addition to the space needed for d, JS needs $\Theta(s)$ amount of space for J.

Note that the profit values are not needed by JS. It is sufficient to know that $p_i \geq p_{i+1}$, $1 \leq i < n$.

The computing time of JS can be reduced from $O(n^2)$ to nearly $O(n)$ by using the disjoint set union and find algorithms (see Section 2.5) and a different method to determine the feasibility of a partial solution. If J is a feasible subset of jobs, then we can determine the processing times for each of the jobs using the rule: if job i hasn't been assigned a processing time, then assign it to the slot $[\alpha - 1, \alpha]$, where α is the largest integer r such that $1 \leq r \leq d_i$ and the slot $[\alpha - 1, \alpha]$ is free. This rule simply delays the processing of job i as much as possible. Consequently, when J is being built up job by job, jobs already in J do not have to be moved from their assigned slots to accommodate the new job. If for the new job being considered there is no α as defined above, then it cannot be included in J. The proof of the validity of this statement is left as an exercise.

Example 4.3 Let $n = 5, (p_1, \ldots, p_5) = (20, 15, 10, 5, 1)$ and $(d_1, \ldots, d_5) = (2, 2, 1, 3, 3)$. Using the above feasibility rule, we have

J	assigned slots	job considered	action	profit
$\emptyset$	none	1	assign to [1, 2]	0
{1}	[1, 2]	2	assign to [0, 1]	20
{1, 2}	[0, 1], [1, 2]	3	cannot fit; reject	35
{1, 2}	[0, 1], [1, 2]	4	assign to [2, 3]	35
{1, 2, 4}	[0, 1], [1, 2], [2, 3]	5	reject	40

The optimal solution is $J = \{1, 2, 4\}$ with a profit of 40. □

Since there are only n jobs and each job takes one unit of time, it is necessary only to consider the time slots $[i - 1, i]$, $1 \leq i \leq b$, such that $b = \min\{n, \max\{d_i\}\}$. One way to implement the above scheduling rule is to partition the time slots $[i-1, i]$, $1 \leq i \leq b$, into sets. We use i to represent the time slots $[i - 1, i]$. For any slot i, let n_i be the largest integer such that $n_i \leq i$ and slot n_i is free. To avoid end conditions, we introduce a fictitious slot $[-1, 0]$ which is always free. Two slots i and j are in the same set iff $n_i = n_j$. Clearly, if i and j, $i < j$, are in the same set, then $i, i+1, i+2, \ldots, j$ are in the same set. Associated with each set k of slots is a value $f(k)$. Then $f(k) = n_i$ for all slots i in set k. Using the set representation of Section 2.5, each set is represented as a tree. The root node identifies the set. The function f is defined only for root nodes. Initially, all slots are free and we have $b + 1$ sets corresponding to the $b + 1$ slots $[i - 1, i]$, $0 \leq i \leq b$. At this time $f(i) = i$, $0 \leq i \leq b$. We use $p(i)$ to link slot i into its set tree. With the conventions for the union and find algorithms of Section 2.5, $p(i) = -1$, $0 \leq i \leq b$, initially. If a job with deadline d is to be scheduled, then we need to find the root of the tree containing the slot $\min\{n, d\}$. If this root is j,

then $f(j)$ is the nearest free slot, provided $f(j) \neq 0$. Having used this slot, the set with root j should be combined with the set containing slot $f(j)-1$.

Example 4.4 The trees defined by the $p(i)$'s for the first three iterations in Example 4.3 are shown in Figure 4.4. □

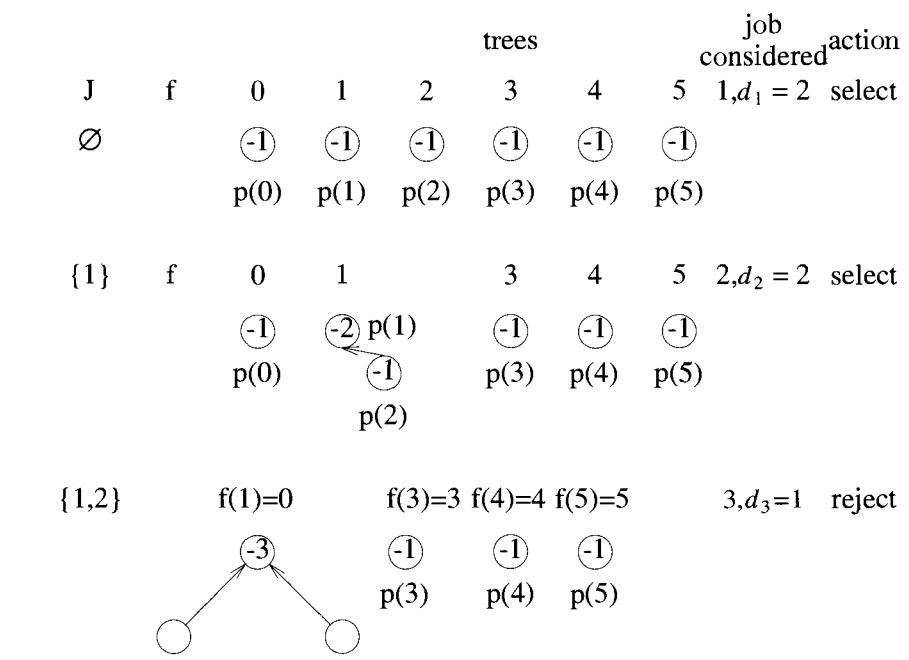


Figure 4.4 Fast job scheduling

The fast algorithm appears as FJS (Algorithm 4.7). Its computing time is readily observed to be $O(n\alpha(2n,n))$ (recall that $\alpha(2n,n)$ is the inverse of Ackermann's function defined in Section 2.5). It needs an additional $2n$ words of space for f and p.

```
1   Algorithm FJS(d, n, b, j)
2   // Find an optimal solution J[1 : k]. It is assumed that
3   // p[1] ≥ p[2] ≥ ··· ≥ p[n] and that b = min{n, max_i(d[i])}.
4   {
5       // Initially there are b + 1 single node trees.
6       for i := 0 to b do f[i] := i;
7       k := 0; // Initialize.
8       for i := 1 to n do
9       { // Use greedy rule.
10          q := CollapsingFind(min(n, d[i]));
11          if (f[q] ≠ 0) then
12          {
13              k := k + 1; J[k] := i; // Select job i.
14              m := CollapsingFind(f[q] − 1);
15              WeightedUnion(m, q);
16              f[q] := f[m]; // q may be new root.
17          }
18      }
19  }
```

Algorithm 4.7 Faster algorithm for job sequencing

EXERCISES

1. You are given a set of n jobs. Associated with each job i is a processing time t_i and a deadline d_i by which it must be completed. A feasible schedule is a permutation of the jobs such that if the jobs are processed in that order, then each job finishes by its deadline. Define a greedy schedule to be one in which the jobs are processed in nondecreasing order of deadlines. Show that if there exists a feasible schedule, then all greedy schedules are feasible.

2. [Optimal assignment] Assume there are n workers and n jobs. Let v_{ij} be the value of assigning worker i to job j. An assignment of workers to jobs corresponds to the assignment of 0 or 1 to the variables x_{ij}, $1 \leq i, j \leq n$. Then $x_{ij} = 1$ means worker i is assigned to job j, and $x_{ij} = 0$ means that worker i is not assigned to job j. A valid assignment is one in which each worker is assigned to exactly one job and exactly one worker is assigned to any one job. The value of an assignment is $\sum_i \sum_j v_{ij} x_{ij}$.

For example, assume there are three workers w_1, w_2, and w_3 and three jobs j_1, j_2, and j_3. Let the values of assignment be $v_{11} = 11$, $v_{12} = 5$, $v_{13} = 8$, $v_{21} = 3$, $v_{22} = 7$, $v_{23} = 15$, $v_{31} = 8$, $v_{32} = 12$, and $v_{33} = 9$. Then, a valid assignment is $x_{12} = 1$, $x_{23} = 1$, and $x_{31} = 1$. The rest of the x_{ij}'s are zeros. The value of this assignment is $5 + 15 + 8 = 28$.

An optimal assignment is a valid assignment of maximum value. Write algorithms for two different greedy assignment schemes. One of these assigns a worker to the best possible job. The other assigns to a job the best possible worker. Show that neither of these schemes is guaranteed to yield optimal assignments. Is either scheme always better than the other? Assume $v_{ij} > 0$.

3. (a) What is the solution generated by the function JS when $n = 7$, $(p_1, p_2, \ldots, p_7) = (3, 5, 20, 18, 1, 6, 30)$, and $(d_1, d_2, \ldots, d_7) = (1, 3, 4, 3, 2, 1, 2)$?

   (b) Show that Theorem 4.3 is true even if jobs have different processing requirements. Associated with job i is a profit $p_i > 0$, a time requirement $t_i > 0$, and a deadline $d_i \geq t_i$.

   (c) Show that for the situation of part (a), the greedy method of this section doesn't necessarily yield an optimal solution.

4. (a) For the job sequencing problem of this section, show that the subset J represents a feasible solution iff the jobs in J can be processed according to the rule: if job i in J hasn't been assigned a processing time, then assign it to the slot $[\alpha - 1, \alpha]$, where α is the least integer r such that $1 \leq r \leq d_i$ and the slot $[\alpha - 1, \alpha]$ is free.

   (b) For the problem instance of Exercise 3(a) draw the trees and give the values of $f(i), 0 \leq i \leq n$, after each iteration of the **for** loop of line 8 of Algorithm 4.7.

4.5 MINIMUM-COST SPANNING TREES

Definition 4.1 Let $G = (V, E)$ be an undirected connected graph. A subgraph $t = (V, E')$ of G is a *spanning tree* of G iff t is a tree. □

Example 4.5 Figure 4.5 shows the complete graph on four nodes together with three of its spanning trees. □

Spanning trees have many applications. For example, they can be used to obtain an independent set of circuit equations for an electric network. First, a spanning tree for the electric network is obtained. Let B be the set of network edges not in the spanning tree. Adding an edge from B to

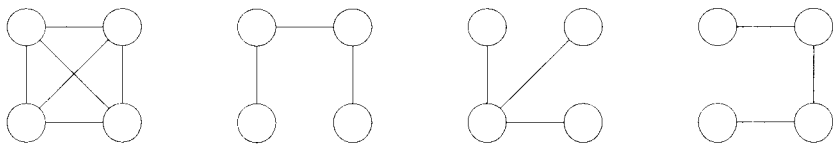


Figure 4.5 An undirected graph and three of its spanning trees

the spanning tree creates a cycle. Kirchoff's second law is used on each cycle to obtain a circuit equation. The cycles obtained in this way are independent (i.e., none of these cycles can be obtained by taking a linear combination of the remaining cycles) as each contains an edge from B that is not contained in any other cycle. Hence, the circuit equations so obtained are also independent. In fact, it can be shown that the cycles obtained by introducing the edges of B one at a time into the resulting spanning tree form a cycle basis, and so all other cycles in the graph can be constructed by taking a linear combination of the cycles in the basis.

Another application of spanning trees arises from the property that a spanning tree is a minimal subgraph G' of G such that $V(G') = V(G)$ and G' is connected. (A minimal subgraph is one with the fewest number of edges.) Any connected graph with n vertices must have at least $n-1$ edges and all connected graphs with $n-1$ edges are trees. If the nodes of G represent cities and the edges represent possible communication links connecting two cities, then the minimum number of links needed to connect the n cities is $n-1$. The spanning trees of G represent all feasible choices.

In practical situations, the edges have weights assigned to them. These weights may represent the cost of construction, the length of the link, and so on. Given such a weighted graph, one would then wish to select cities to have minimum total cost or minimum total length. In either case the links selected have to form a tree (assuming all weights are positive). If this is not so, then the selection of links contains a cycle. Removal of any one of the links on this cycle results in a link selection of less cost connecting all cities. We are therefore interested in finding a spanning tree of G with minimum cost. (The cost of a spanning tree is the sum of the costs of the edges in that tree.) Figure 4.6 shows a graph and one of its minimum-cost spanning trees. Since the identification of a minimum-cost spanning tree involves the selection of a subset of the edges, this problem fits the subset paradigm.

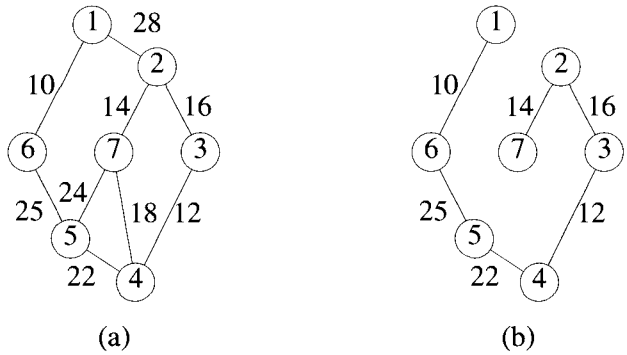


Figure 4.6 A graph and its minimum cost spanning tree

4.5.1 Prim's Algorithm

A greedy method to obtain a minimum-cost spanning tree builds this tree edge by edge. The next edge to include is chosen according to some optimization criterion. The simplest such criterion is to choose an edge that results in a minimum increase in the sum of the costs of the edges so far included. There are two possible ways to interpret this criterion. In the first, the set of edges so far selected form a tree. Thus, if A is the set of edges selected so far, then A forms a tree. The next edge (u, v) to be included in A is a minimum-cost edge not in A with the property that $A \cup \{(u, v)\}$ is also a tree. Exercise 2 shows that this selection criterion results in a minimum-cost spanning tree. The corresponding algorithm is known as Prim's algorithm.

Example 4.6 Figure 4.7 shows the working of Prim's method on the graph of Figure 4.6(a). The spanning tree obtained is shown in Figure 4.6(b) and has a cost of 99. □

Having seen how Prim's method works, let us obtain a pseudocode algorithm to find a minimum-cost spanning tree using this method. The algorithm will start with a tree that includes only a minimum-cost edge of G. Then, edges are added to this tree one by one. The next edge (i, j) to be added is such that i is a vertex already included in the tree, j is a vertex not yet included, and the cost of (i, j), $cost[i, j]$, is minimum among all edges (k, l) such that vertex k is in the tree and vertex l is not in the tree. To determine this edge (i, j) efficiently, we associate with each vertex j not yet included in the tree a value $near[j]$. The value $near[j]$ is a vertex in the tree such that $cost[j, near[j]]$ is minimum among all choices for $near[j]$. We define $near[j] = 0$ for all vertices j that are already in the tree. The next edge

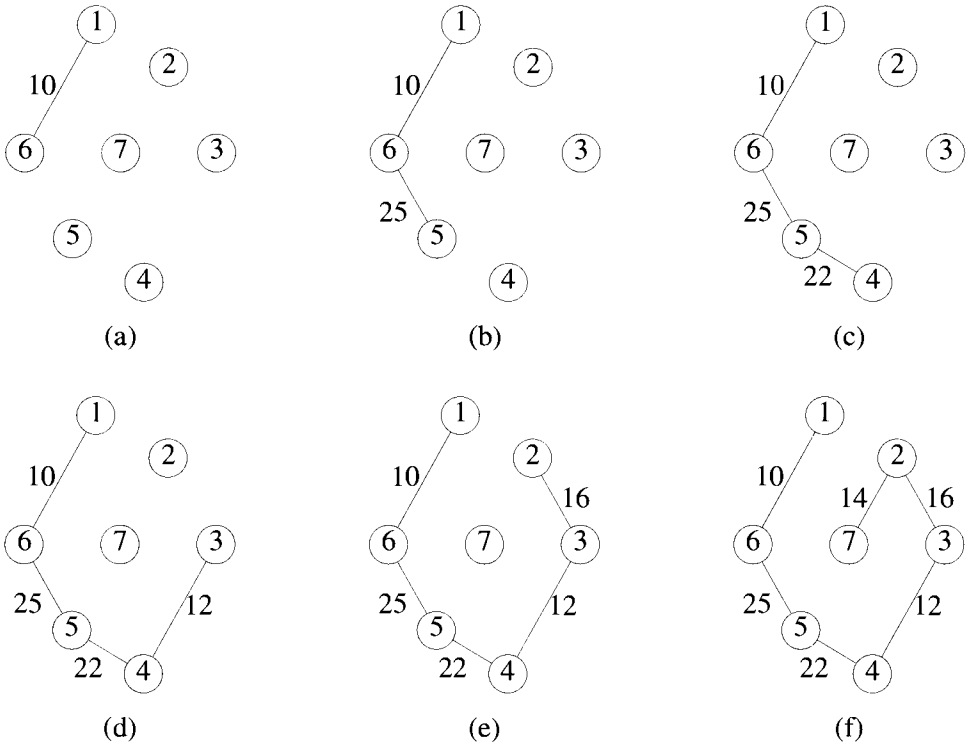


Figure 4.7 Stages in Prim's algorithm

to include is defined by the vertex j such that $near[j] \neq 0$ (j not already in the tree) and $cost[j, near[j]]$ is minimum.

In function Prim (Algorithm 4.8), line 9 selects a minimum-cost edge. Lines 10 to 15 initialize the variables so as to represent a tree comprising only the edge (k, l). In the **for** loop of line 16 the remainder of the spanning tree is built up edge by edge. Lines 18 and 19 select $(j, near[j])$ as the next edge to include. Lines 23 to 25 update $near[\,]$.

The time required by algorithm Prim is $O(n^2)$, where n is the number of vertices in the graph G. To see this, note that line 9 takes $O(|E|)$ time and line 10 takes $\Theta(1)$ time. The **for** loop of line 12 takes $\Theta(n)$ time. Lines 18 and 19 and the **for** loop of line 23 require $O(n)$ time. So, each iteration of the **for** loop of line 16 takes $O(n)$ time. The total time for the **for** loop of line 16 is therefore $O(n^2)$. Hence, Prim runs in $O(n^2)$ time.

If we store the nodes not yet included in the tree as a red-black tree (see Section 2.4.2), lines 18 and 19 take $O(\log n)$ time. Note that a red-black tree supports the following operations in $O(\log n)$ time: insert, delete (an arbitrary element), find-min, and search (for an arbitrary element). The **for** loop of line 23 has to examine only the nodes adjacent to j. Thus its overall frequency is $O(|E|)$. Updating in lines 24 and 25 also takes $O(\log n)$ time (since an update can be done using a delete and an insertion into the red-black tree). Thus the overall run time is $O((n + |E|) \log n)$.

The algorithm can be speeded a bit by making the observation that a minimum-cost spanning tree includes for each vertex v a minimum-cost edge incident to v. To see this, suppose t is a minimum-cost spanning tree for $G = (V, E)$. Let v be any vertex in t. Let (v, w) be an edge with minimum cost among all edges incident to v. Assume that $(v, w) \notin E(t)$ and $cost[v, w] < cost[v, x]$ for all edges $(v, x) \in E(t)$. The inclusion of (v, w) into t creates a unique cycle. This cycle must include an edge (v, x), $x \neq w$. Removing (v, x) from $E(t) \cup \{(v, w)\}$ breaks this cycle without disconnecting the graph $(V, E(t) \cup \{(v, w)\})$. Hence, $(V, E(t) \cup \{(v, w)\} - \{(v, x)\})$ is also a spanning tree. Since $cost[v, w] < cost[v, x]$, this spanning tree has lower cost than t. This contradicts the assumption that t is a minimum-cost spanning tree of G. So, t includes minimum-cost edges as stated above.

From this observation it follows that we can start the algorithm with a tree consisting of any arbitrary vertex and no edge. Then edges can be added one by one. The changes needed are to lines 9 to 17. These lines can be replaced by the lines

```
9'        mincost := 0;
10'       for i := 2 to n do near[i] := 1;
11'           // Vertex 1 is initially in t.
12'       near[1] := 0;
13'-16'   for i := 1 to n - 1 do
17'       { // Find n - 1 edges for t.
```

4.5.2 Kruskal's Algorithm

There is a second possible interpretation of the optimization criteria mentioned earlier in which the edges of the graph are considered in nondecreasing order of cost. This interpretation is that the set t of edges so far selected for the spanning tree be such that it is possible to *complete* t into a tree. Thus t may not be a tree at all stages in the algorithm. In fact, it will generally only be a forest since the set of edges t can be completed into a tree iff there are no cycles in t. We show in Theorem 4.6 that this interpretation of the greedy method also results in a minimum-cost spanning tree. This method is due to Kruskal.

```
1   Algorithm Prim(E, cost, n, t)
2   // E is the set of edges in G. cost[1 : n, 1 : n] is the cost
3   // adjacency matrix of an n vertex graph such that cost[i, j] is
4   // either a positive real number or ∞ if no edge (i, j) exists.
5   // A minimum spanning tree is computed and stored as a set of
6   // edges in the array t[1 : n − 1, 1 : 2]. (t[i, 1], t[i, 2]) is an edge in
7   // the minimum-cost spanning tree. The final cost is returned.
8   {
9       Let (k, l) be an edge of minimum cost in E;
10      mincost := cost[k, l];
11      t[1, 1] := k; t[1, 2] := l;
12      for i := 1 to n do  // Initialize near.
13          if (cost[i, l] < cost[i, k]) then near[i] := l;
14          else near[i] := k;
15      near[k] := near[l] := 0;
16      for i := 2 to n − 1 do
17      { // Find n − 2 additional edges for t.
18          Let j be an index such that near[j] ≠ 0 and
19          cost[j, near[j]] is minimum;
20          t[i, 1] := j; t[i, 2] := near[j];
21          mincost := mincost + cost[j, near[j]];
22          near[j] := 0;
23          for k := 1 to n do // Update near[ ].
24              if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j]))
25                  then near[k] := j;
26      }
27      return mincost;
28  }
```

Algorithm 4.8 Prim's minimum-cost spanning tree algorithm

Example 4.7 Consider the graph of Figure 4.6(a). We begin with no edges selected. Figure 4.8(a) shows the current graph with no edges selected. Edge $(1,6)$ is the first edge considered. It is included in the spanning tree being built. This yields the graph of Figure 4.8(b). Next, the edge $(3,4)$ is selected and included in the tree (Figure 4.8(c)). The next edge to be considered is $(2,7)$. Its inclusion in the tree being built does not create a cycle, so we get the graph of Figure 4.8(d). Edge $(2,3)$ is considered next and included in the tree Figure 4.8(e). Of the edges not yet considered, $(7,4)$ has the least cost. It is considered next. Its inclusion in the tree results in a cycle, so this edge is discarded. Edge $(5,4)$ is the next edge to be added to the tree being built. This results in the configuration of Figure 4.8(f). The next edge to be considered is the edge $(7,5)$. It is discarded, as its inclusion creates a cycle. Finally, edge $(6,5)$ is considered and included in the tree being built. This completes the spanning tree. The resulting tree (Figure 4.6(b)) has cost 99. □

For clarity, Kruskal's method is written out more formally in Algorithm 4.9. Initially E is the set of all edges in G. The only functions we wish to perform on this set are (1) determine an edge with minimum cost (line 4) and (2) delete this edge (line 5). Both these functions can be performed efficiently if the edges in E are maintained as a sorted sequential list. It is not essential to sort all the edges so long as the next edge for line 4 can be determined easily. If the edges are maintained as a minheap, then the next edge to consider can be obtained in $O(\log |E|)$ time. The construction of the heap itself takes $O(|E|)$ time.

To be able to perform step 6 efficiently, the vertices in G should be grouped together in such a way that one can easily determine whether the vertices v and w are already connected by the earlier selection of edges. If they are, then the edge (v,w) is to be discarded. If they are not, then (v,w) is to be added to t. One possible grouping is to place all vertices in the same connected component of t into a set (all connected components of t will also be trees). Then, two vertices v and w are connected in t iff they are in the same set. For example, when the edge $(2,6)$ is to be considered, the sets are $\{1,2\}, \{3,4,6\}$, and $\{5\}$. Vertices 2 and 6 are in different sets so these sets are combined to give $\{1,2,3,4,6\}$ and $\{5\}$. The next edge to be considered is $(1,4)$. Since vertices 1 and 4 are in the same set, the edge is rejected. The edge $(3,5)$ connects vertices in different sets and results in the final spanning tree. Using the set representation and the union and find algorithms of Section 2.5, we can obtain an efficient (almost linear) implementation of line 6. The computing time is, therefore, determined by the time for lines 4 and 5, which in the worst case is $O(|E| \log |E|)$.

If the representations discussed above are used, then the pseudocode of Algorithm 4.10 results. In line 6 an initial heap of edges is constructed. In line 7 each vertex is assigned to a distinct set (and hence to a distinct tree). The set t is the set of edges to be included in the minimum-cost spanning

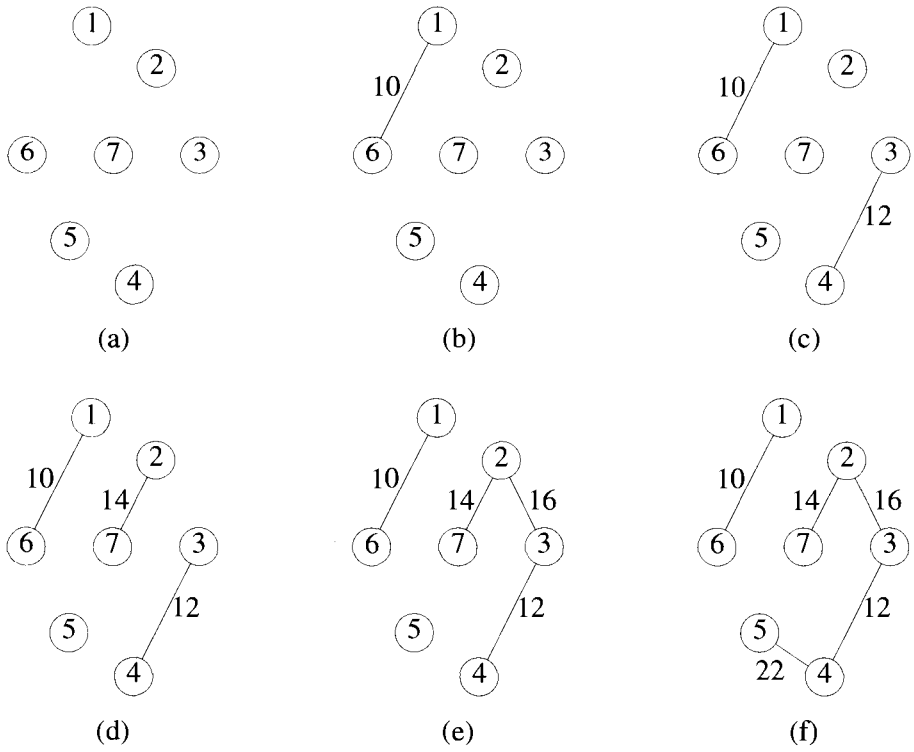


Figure 4.8 Stages in Kruskal's algorithm

tree and i is the number of edges in t. The set t can be represented as a sequential list using a two-dimensional array $t[1:n-1, 1:2]$. Edge (u, v) can be added to t by the assignments $t[i, 1] := u$; and $t[i, 2] := v$;. In the **while** loop of line 10, edges are removed from the heap one by one in nondecreasing order of cost. Line 14 determines the sets containing u and v. If $j \neq k$, then vertices u and v are in different sets (and so in different trees) and edge (u, v) is included into t. The sets containing u and v are combined (line 20). If $u = v$, the edge (u, v) is discarded as its inclusion into t would create a cycle. Line 23 determines whether a spanning tree was found. It follows that $i \neq n - 1$ iff the graph G is not connected. One can verify that the computing time is $O(|E| \log |E|)$, where E is the edge set of G.

Theorem 4.6 Kruskal's algorithm generates a minimum-cost spanning tree for every connected undirected graph G.

```
1  t := ∅;
2  while ((t has less than n − 1 edges) and (E ≠ ∅)) do
3  {
4      Choose an edge (v, w) from E of lowest cost;
5      Delete (v, w) from E;
6      if (v, w) does not create a cycle in t then add (v, w) to t;
7      else discard (v, w);
8  }
```

Algorithm 4.9 Early form of minimum-cost spanning tree algorithm due to Kruskal

```
1  Algorithm Kruskal(E, cost, n, t)
2  // E is the set of edges in G. G has n vertices. cost[u, v] is the
3  // cost of edge (u, v). t is the set of edges in the minimum-cost
4  // spanning tree. The final cost is returned.
5  {
6      Construct a heap out of the edge costs using Heapify;
7      for i := 1 to n do parent[i] := −1;
8      // Each vertex is in a different set.
9      i := 0; mincost := 0.0;
10     while ((i < n − 1) and (heap not empty)) do
11     {
12         Delete a minimum cost edge (u, v) from the heap
13         and reheapify using Adjust;
14         j := Find(u); k := Find(v);
15         if (j ≠ k) then
16         {
17             i := i + 1;
18             t[i, 1] := u; t[i, 2] := v;
19             mincost := mincost + cost[u, v];
20             Union(j, k);
21         }
22     }
23     if (i ≠ n − 1) then write ("No spanning tree");
24     else return mincost;
25 }
```

Algorithm 4.10 Kruskal's algorithm

Proof: Let G be any undirected connected graph. Let t be the spanning tree for G generated by Kruskal's algorithm. Let t' be a minimum-cost spanning tree for G. We show that both t and t' have the same cost.

Let $E(t)$ and $E(t')$ respectively be the edges in t and t'. If n is the number of vertices in G, then both t and t' have $n-1$ edges. If $E(t) = E(t')$, then t is clearly of minimum cost. If $E(t) \neq E(t')$, then let q be a minimum-cost edge such that $q \in E(t)$ and $q \notin E(t')$. Clearly, such a q must exist. The inclusion of q into t' creates a unique cycle (Exercise 5). Let $q, e_1, e_2, \ldots, e_k$ be this unique cycle. At least one of the e_i's, $1 \leq i \leq k$, is not in $E(t)$ as otherwise t would also contain the cycle $q, e_1, e_2, \ldots, e_k$. Let e_j be an edge on this cycle such that $e_j \notin E(t)$. If e_j is of lower cost than q, then Kruskal's algorithm will consider e_j before q and include e_j into t. To see this, note that all edges in $E(t)$ of cost less than the cost of q are also in $E(t')$ and do not form a cycle with e_j. So $cost(e_j) \geq cost(q)$.

Now, reconsider the graph with edge set $E(t') \cup \{q\}$. Removal of any edge on the cycle $q, e_1, e_2, \ldots, e_k$ will leave behind a tree t'' (Exercise 5). In particular, if we delete the edge e_j, then the resulting tree t'' will have a cost no more than the cost of t' (as $cost(e_j) \geq cost(e)$). Hence, t'' is also a minimum-cost tree.

By repeatedly using the transformation described above, tree t' can be transformed into the spanning tree t without any increase in cost. Hence, t is a minimum-cost spanning tree. □

4.5.3 An Optimal Randomized Algorithm (*)

Any algorithm for finding the minimum-cost spanning tree of a given graph $G(V, E)$ will have to spend $\Omega(|V| + |E|)$ time in the worst case, since it has to examine each node and each edge at least once before determining the correct answer. A randomized Las Vegas algorithm that runs in time $\tilde{O}(|V| + |E|)$ can be devised as follows: (1) Randomly sample m edges from G (for some suitable m). (2) Let G' be the induced subgraph; that is, G' has V as its node set and the sampled edges in its edge set. The subgraph G' need not be connected. Recursively find a minimum-cost spanning tree for each component of G'. Let F be the resultant *minimum-cost spanning forest* of G'. (3) Using F, eliminate certain edges (called the *F-heavy edges*) of G that cannot possibly be in a minimum-cost spanning tree. Let G'' be the graph that results from G after elimination of the F-heavy edges. (4) Recursively find a minimum-cost spanning tree for G''. This will also be a minimum-cost spanning tree for G.

Steps 1 to 3 are useful in reducing the number of edges in G. The algorithm can be speeded up further if we can reduce the number of nodes in the input graph as well. Such a node elimination can be effected using the *Borůvka steps*. In a Borůvka step, for each node, an incident edge with minimum weight is chosen. For example in Figure 4.9(a), the edge $(1, 3)$ is

chosen for node 1, the edge $(6,7)$ is chosen for node 7, and so on. All the chosen edges are shown with thick lines. The connected components of the induced graph are found. In the example of Figure 4.9(a), the nodes 1, 2, and 3 form one component, the nodes 4 and 5 form a second component, and the nodes 6 and 7 form another component. Replace each component with a single node. The component with nodes 1, 2, and 3 is replaced with the node a. The other two components are replaced with the nodes b and c, respectively. Edges within the individual components are thrown away. The resultant graph is shown in Figure 4.9(b). In this graph keep only an edge of minimum weight between any two nodes. Delete any isolated nodes.

Since an edge is chosen for every node, the number of nodes after one Borůvka step reduces by a factor of at least two. A minimum-cost spanning tree for the reduced graph can be extended easily to get a minimum-cost spanning tree for the original graph. If E' is the set of edges in the minimum-cost spanning tree of the reduced graph, we simply include into E' the edges chosen in the Borůvka step to obtain the minimum-cost spanning tree edges for the original graph. In the example of Figure 4.9, a minimum-cost spanning tree for (c) will consist of the edges (a,b) and (b,c). Thus a minimum-cost spanning tree for the graph of (a) will have the edges: $(1,3),(3,2),(4,5),(6,7),(3,4)$, and $(2,6)$. More details of the algorithms are given below.

Definition 4.2 Let F be a forest that forms a subgraph of a given weighted graph $G(V,E)$. If u and v are any two nodes in F, let $F(u,v)$ denote the path (if any) connecting u and v in F and let $Fcost(u,v)$ denote the maximum weight of any edge in the path $F(u,v)$. If there is no path between u and v in F, $Fcost(u,v)$ is taken to be ∞. Any edge (x,y) of G is said to be F-heavy if $cost[x,y] > Fcost(x,y)$ and F-light otherwise. □

Note that all the edges of F are F-light. Also, any F-heavy edge cannot belong to a minimum-cost spanning tree of G. The proof of this is left as an exercise. The randomized algorithm applies two Borůvka steps to reduce the number of nodes in the input graph. Next, it samples the edges of G and processes them to eliminate a constant fraction of them. A minimum-cost spanning tree for the resultant reduced graph is recursively computed. From this tree, a spanning tree for G is obtained. A detailed description of the algorithm appears as Algorithm 4.11.

Lemma 4.3 states that Step 4 can be completed in time $O(|V|+|E|)$. The proof of this can be found in the references supplied at the end of this chapter. Step 1 takes $O(|V|+|E|)$ time and step 2 takes $O(|E|)$ time. Step 6 takes $O(|E|)$ time as well. The time taken in all the recursive calls in steps 3 and 5 can be shown to be $O(|V|+|E|)$. For a proof, see the references at the end of the chapter. A crucial fact that is used in the proof is that both the number of nodes and the number of edges are reduced by a constant factor, with high probability, in each level of recursion.

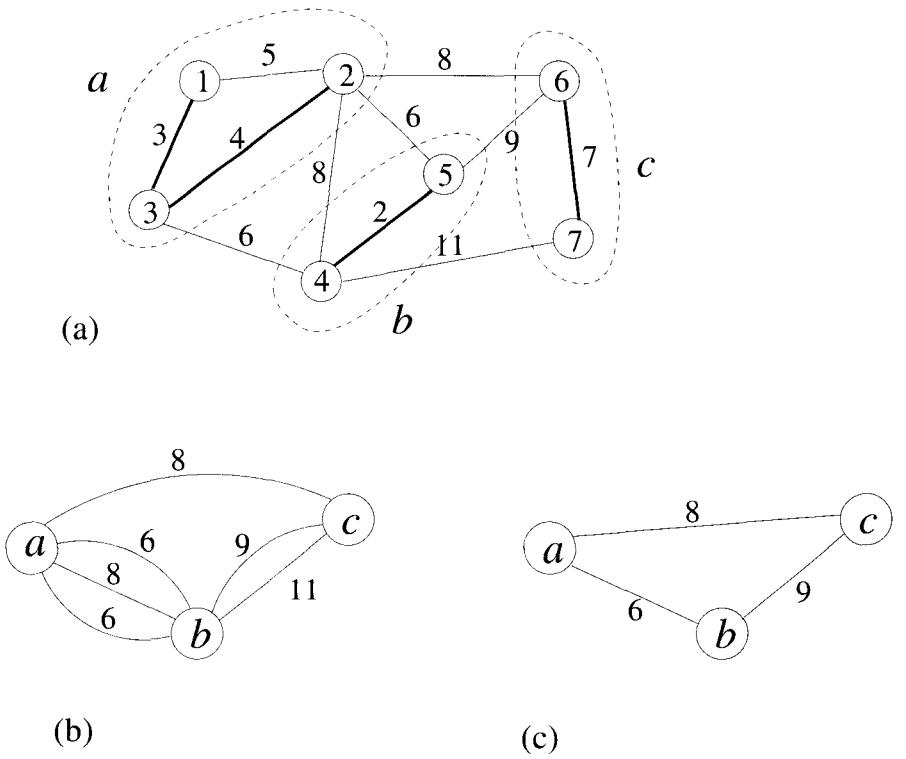


Figure 4.9 A Borůvka step

Lemma 4.3 Let $G(V, E)$ be any weighted graph and let F be a subgraph of G that forms a forest. Then, all the F-heavy edges of G can be identified in time $O(|V| + |E|)$. □

Theorem 4.7 A minimum-weight spanning tree for any given weighted graph can be computed in time $\tilde{O}(|V| + |E|)$. □

EXERCISES

1. Compute a minimum cost spanning tree for the graph of Figure 4.10 using (a) Prim's algorithm and (b) Kruskal's algorithm.

2. Prove that Prim's method of this section generates minimum-cost spanning trees.

Step 1. Apply two Borůvka steps. At the end, the number of nodes will have decreased by a factor at least 4. Let the resultant graph be $\widetilde{G}(\widetilde{V}, \widetilde{E})$.

Step 2. Form a subgraph $G'(V', E')$ of $\widetilde{G}$, where each edge of $\widetilde{G}$ is chosen randomly to be in E' with probability $\frac{1}{2}$. The expected number of edges in E' is $\frac{|\widetilde{E}|}{2}$.

Step 3. Recursively find a minimum-cost spanning forest F for G'.

Step 4. Eliminate all the F-heavy edges from $\widetilde{G}$. With high probability, at least a constant fraction of the edges of $\widetilde{G}$ will be eliminated. Let G'' be the resultant graph.

Step 5. Compute a minimum-cost spanning tree (call it T'') for G'' recursively. The tree T'' will also be a minimum-cost spanning tree for $\widetilde{G}$.

Step 6. Return the edges of T'' together with the edges chosen in the Borůvka steps of step 1. These are the edges of a minimum-cost spanning tree for G.

Algorithm 4.11 An optimal randomized algorithm

3. (a) Rewrite Prim's algorithm under the assumption that the graphs are represented by adjacency lists.
   (b) Program and run the above version of Prim's algorithm against Algorithm 4.9. Compare the two on a representative set of graphs.
   (c) Analyze precisely the computing time and space requirements of your new version of Prim's algorithm using adjacency lists.

4. Program and run Kruskal's algorithm, described in Algorithm 4.10. You will have to modify functions Heapify and Adjust of Chapter 2. Use the same test data you devised to test Prim's algorithm in Exercise 3.

5. (a) Show that if t is a spanning tree for the undirected graph G, then the addition of an edge q, $q \notin E(t)$ and $q \in E(G)$, to t creates a unique cycle.

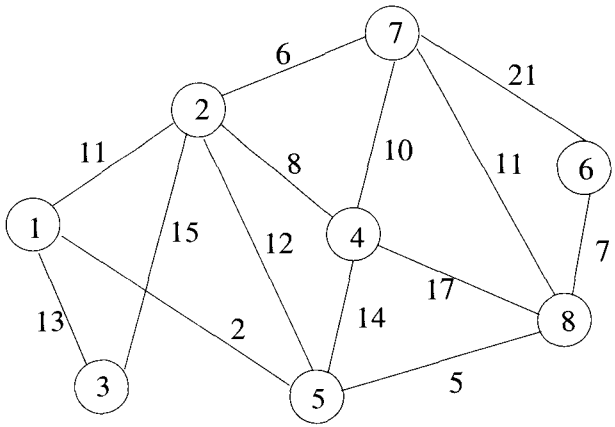


Figure 4.10 Graph for Exercise 1

(b) Show that if any of the edges on this unique cycle is deleted from $E(t) \cup \{q\}$, then the remaining edges form a spanning tree of G.

6. In Figure 4.9, find a minimum-cost spanning tree for the graph of part (c) and extend the tree to obtain a minimum cost spanning tree for the graph of part (a). Verify the correctness of your answer by applying either Prim's algorithm or Kruskal's algorithm on the graph of part (a).

7. Let $G(V, E)$ be any weighted connected graph.

   (a) If C is any cycle of G, then show that the heaviest edge of C cannot belong to a minimum-cost spanning tree of G.

   (b) Assume that F is a forest that is a subgraph of G. Show that any F-heavy edge of G cannot belong to a minimum-cost spanning tree of G.

8. By considering the complete graph with n vertices, show that the number of spanning trees in an n vertex graph can be greater than $2^{n-1}-2$.

4.6 OPTIMAL STORAGE ON TAPES

There are n programs that are to be stored on a computer tape of length l. Associated with each program i is a length $l_i, 1 \leq i \leq n$. Clearly, all programs can be stored on the tape if and only if the sum of the lengths of

the programs is at most l. We assume that whenever a program is to be retrieved from this tape, the tape is initially positioned at the front. Hence, if the programs are stored in the order $I = i_1, i_2, \ldots, i_n$, the time t_j needed to retrieve program i_j is proportional to $\sum_{1 \le k \le j} l_{i_k}$. If all programs are retrieved equally often, then the expected or *mean retrieval time* (MRT) is $(1/n) \sum_{1 \le j \le n} t_j$. In the optimal storage on tape problem, we are required to find a permutation for the n programs so that when they are stored on the tape in this order the MRT is minimized. This problem fits the ordering paradigm. Minimizing the MRT is equivalent to minimizing $d(I) = \sum_{1 \le j \le n} \sum_{1 \le k \le j} l_{i_k}$.

Example 4.8 Let $n = 3$ and $(l_1, l_2, l_3) = (5, 10, 3)$. There are $n! = 6$ possible orderings. These orderings and their respective d values are:

ordering I	$d(I)$		
$1, 2, 3$	$5 + 5 + 10 + 5 + 10 + 3$	$=$	38
$1, 3, 2$	$5 + 5 + 3 + 5 + 3 + 10$	$=$	31
$2, 1, 3$	$10 + 10 + 5 + 10 + 5 + 3$	$=$	43
$2, 3, 1$	$10 + 10 + 3 + 10 + 3 + 5$	$=$	41
$3, 1, 2$	$3 + 3 + 5 + 3 + 5 + 10$	$=$	29
$3, 2, 1$	$3 + 3 + 10 + 3 + 10 + 5$	$=$	34

The optimal ordering is $3, 1, 2$. □

A greedy approach to building the required permutation would choose the next program on the basis of some optimization measure. One possible measure would be the d value of the permutation constructed so far. The next program to be stored on the tape would be one that minimizes the increase in d. If we have already constructed the permutation $i_1, i_2, \ldots, i_r$, then appending program j gives the permutation $i_1, i_2, \ldots, i_r, i_{r+1} = j$. This increases the d value by $\sum_{1 \le k \le r} l_{i_k} + l_j$. Since $\sum_{1 \le k \le r} l_{i_k}$ is fixed and independent of j, we trivially observe that the increase in d is minimized if the next program chosen is the one with the least length from among the remaining programs.

The greedy algorithm resulting from the above discussion is so simple that we won't bother to write it out. The greedy method simply requires us to store the programs in nondecreasing order of their lengths. This ordering can be carried out in $O(n \log n)$ time using an efficient sorting algorithm (e.g., heap sort from Chapter 2). For the programs of Example 4.8, note that the permutation that yields an optimal solution is the one in which the programs are in nondecreasing order of their lengths. Theorem 4.8 shows that the MRT is minimized when programs are stored in this order.

Theorem 4.8 If $l_1 \leq l_2 \leq \cdots \leq l_n$, then the ordering $i_j = j, 1 \leq j \leq n$, minimizes

$$\sum_{k=1}^{n} \sum_{j=1}^{k} l_{i_j}$$

over all possible permutations of the i_j.

Proof: Let $I = i_1, i_2, \ldots, i_n$ be any permutation of the index set $\{1, 2, \ldots, n\}$. Then

$$d(I) = \sum_{k=1}^{n} \sum_{j=1}^{k} l_{i_j} = \sum_{k=1}^{n} (n - k + 1) l_{i_k}$$

If there exist a and b such that $a < b$ and $l_{i_a} > l_{i_b}$, then interchanging i_a and i_b results in a permutation I' with

$$d(I') = \left[\sum_{\substack{k \\ k \neq a \\ k \neq b}} (n - k + 1) l_{i_k} \right] + (n - a + 1) l_{i_b} + (n - b + 1) l_{i_a}$$

Subtracting $d(I')$ from $d(I)$, we obtain

$$\begin{aligned} d(I) - d(I') &= (n - a + 1)(l_{i_a} - l_{i_b}) + (n - b + 1)(l_{i_b} - l_{i_a}) \\ &= (b - a)(l_{i_a} - l_{i_b}) \\ &> 0 \end{aligned}$$

Hence, no permutation that is not in nondecreasing order of the l_i's can have minimum d. It is easy to see that all permutations in nondecreasing order of the l_i's have the same d value. Hence, the ordering defined by $i_j = j, 1 \leq j \leq n$, minimizes the d value. □

The tape storage problem can be extended to several tapes. If there are $m > 1$ tapes, $T_0, \ldots, T_{m-1}$, then the programs are to be distributed over these tapes. For each tape a storage permutation is to be provided. If I_j is the storage permutation for the subset of programs on tape j, then $d(I_j)$ is as defined earlier. The *total retrieval time* (TD) is $\sum_{0 \leq j \leq m-1} d(I_j)$. The objective is to store the programs in such a way as to minimize TD.

The obvious generalization of the solution for the one-tape case is to consider the programs in nondecreasing order of l_i's. The program currently

```
1   Algorithm Store(n, m)
2   // n is the number of programs and m the number of tapes.
3   {
4       j := 0; // Next tape to store on
5       for i := 1 to n do
6       {
7           write ("append program", i,
8               "to permutation for tape", j);
9           j := (j + 1) mod m;
10      }
11  }
```

Algorithm 4.12 Assigning programs to tapes

being considered is placed on the tape that results in the minimum increase in TD. This tape will be the one with the least amount of tape used so far. If there is more than one tape with this property, then the one with the smallest index can be used. If the jobs are initially ordered so that $l_1 \leq l_2 \leq \cdots \leq l_n$, then the first m programs are assigned to tapes $T_0, \ldots, T_{m-1}$ respectively. The next m programs will be assigned to tapes $T_0, \ldots, T_{m-1}$ respectively. The general rule is that program i is stored on tape $T_{i \bmod m}$. On any given tape the programs are stored in nondecreasing order of their lengths. Algorithm 4.12 presents this rule in pseudocode. It assumes that the programs are ordered as above. It has a computing time of $\Theta(n)$ and does not need to know the program lengths. Theorem 4.9 proves that the resulting storage pattern is optimal.

Theorem 4.9 If $l_1 \leq l_2 \leq \cdots \leq l_n$, then Algorithm 4.12 generates an optimal storage pattern for m tapes.

Proof: In any storage pattern for m tapes, let r_i be one greater than the number of programs following program i on its tape. Then the total retrieval time TD is given by

$$TD = \sum_{i=1}^{n} r_i l_i$$

In any given storage pattern, for any given n, there can be at most m programs for which $r_i = j$. From Theorem 4.8 it follows that TD is minimized if the m longest programs have $r_i = 1$, the next m longest programs have

$r_i = 2$, and so on. When programs are ordered by length, that is, $l_1 \leq l_2 \leq \cdots \leq l_n$, then this minimization criteria is satisfied if $r_i = \lceil (n-i+1)/m \rceil$. Observe that Algorithm 4.12 results in a storage pattern with these r_i's. □

The proof of Theorem 4.9 shows that there are many storage patterns that minimize TD. If we compute $r_i = \lceil (n-i+1)/m \rceil$ for each program i, then so long as all programs with the same r_i are stored on different tapes and have $r_i - 1$ programs following them, TD is the same. If n is a multiple of m, then there are at least $(m!)^{n/m}$ storage patterns that minimize TD. Algorithm 4.12 produces one of these.

EXERCISES

1. Find an optimal placement for 13 programs on three tapes T_0, T_1, and T_2, where the programs are of lengths $12, 5, 8, 32, 7, 5, 18, 26, 4, 3, 11, 10$, and 6.

2. Show that replacing the code of Algorithm 4.12 by

```
for i := 1 to n do
    write ("append program", i, "to permutation for
        tape", (i - 1) mod m);
```

   does not affect the output.

3. Let $P_1, P_2, \ldots, P_n$ be a set of n programs that are to be stored on a tape of length l. Program P_i requires a_i amount of tape. If $\sum a_i \leq l$, then clearly all the programs can be stored on the tape. So, assume $\sum a_i > l$. The problem is to select a maximum subset Q of the programs for storage on the tape. (A maximum subset is one with the maximum number of programs in it). A greedy algorithm for this problem would build the subset Q by including programs in nondecreasing order of a_i.

   (a) Assume the P_i are ordered such that $a_1 \leq a_2 \leq \cdots \leq a_n$. Write a function for the above strategy. Your function should output an array $s[1:n]$ such that $s[i] = 1$ if P_i is in Q and $s[i] = 0$ otherwise.

   (b) Show that this strategy always finds a maximum subset Q such that $\sum_{P_i \in Q} a_i \leq l$.

   (c) Let Q be the subset obtained using the above greedy strategy. How small can the tape utilization ratio $(\sum_{P_i \in Q} a_i)/l$ get?

   (d) Suppose the objective now is to determine a subset of programs that maximizes the tape utilization ratio. A greedy approach

would be to consider programs in nonincreasing order of a_i. If there is enough space left on the tape for P_i, then it is included in Q. Assume the programs are ordered so that $a_1 \geq a_2 \geq \cdots \geq a_n$. Write a function incorporating this strategy. What is its time and space complexity?

(e) Show that the strategy of part (d) doesn't necessarily yield a subset that maximizes $(\sum_{P_i \in Q} a_i)/l$. How small can this ratio get? Prove your bound.

4. Assume n programs of lengths $l_1, l_2, \ldots, l_n$ are to be stored on a tape. Program i is to be retrieved with frequency f_i. If the programs are stored in the order $i_1, i_2, \ldots, i_n$, the *expected retrieval time* (ERT) is

$$\left[\sum_j (f_{i_j} \sum_{k=1}^{j} l_{i_k})\right] / \sum f_i$$

(a) Show that storing the programs in nondecreasing order of l_i does not necessarily minimize the ERT.

(b) Show that storing the programs in nonincreasing order of f_i does not necessarily minimize the ERT.

(c) Show that the ERT is minimized when the programs are stored in nonincreasing order of f_i/l_i.

5. Consider the tape storage problem of this section. Assume that two tapes $T1$ and $T2$, are available and we wish to distribute n given programs of lengths $l_1, l_2, \ldots, l_n$ onto these two tapes in such a manner that the maximum retrieval time is minimized. That is, if A and B are the sets of programs on the tapes $T1$ and $T2$ respectively, then we wish to choose A and B such that $\max \{ \sum_{i \in A} l_i, \sum_{i \in B} l_i \}$ is minimized. A possible greedy approach to obtaining A and B would be to start with A and B initially empty. Then consider the programs one at a time. The program currently being considered is assigned to set A if $\sum_{i \in A} l_i = \min \{ \sum_{i \in A} l_i, \sum_{i \in B} l_i \}$; otherwise it is assigned to B. Show that this does not guarantee optimal solutions even if $l_1 \leq l_2 \leq \cdots \leq l_n$. Show that the same is true if we require $l_1 \geq l_2 \geq \cdots \geq l_n$.

4.7 OPTIMAL MERGE PATTERNS

In Section 3.4 we saw that two sorted files containing n and m records respectively could be merged together to obtain one sorted file in time $O(n+m)$. When more than two sorted files are to be merged together, the merge can be accomplished by repeatedly merging sorted files in pairs. Thus, if

files x_1, x_2, x_3, and x_4 are to be merged, we could first merge x_1 and x_2 to get a file y_1. Then we could merge y_1 and x_3 to get y_2. Finally, we could merge y_2 and x_4 to get the desired sorted file. Alternatively, we could first merge x_1 and x_2 getting y_1, then merge x_3 and x_4 and get y_2, and finally merge y_1 and y_2 and get the desired sorted file. Given n sorted files, there are many ways in which to pairwise merge them into a single sorted file. Different pairings require differing amounts of computing time. The problem we address ourselves to now is that of determining an optimal way (one requiring the fewest comparisons) to pairwise merge n sorted files. Since this problem calls for an ordering among the pairs to be merged, it fits the ordering paradigm.

Example 4.9 The files x_1, x_2, and x_3 are three sorted files of length 30, 20, and 10 records each. Merging x_1 and x_2 requires 50 record moves. Merging the result with x_3 requires another 60 moves. The total number of record moves required to merge the three files this way is 110. If, instead, we first merge x_2 and x_3 (taking 30 moves) and then x_1 (taking 60 moves), the total record moves made is only 90. Hence, the second merge pattern is faster than the first. □

A greedy attempt to obtain an optimal merge pattern is easy to formulate. Since merging an n-record file and an m-record file requires possibly $n + m$ record moves, the obvious choice for a selection criterion is: at each step merge the two smallest size files together. Thus, if we have five files $(x_1, \ldots, x_5)$ with sizes $(20, 30, 10, 5, 30)$, our greedy rule would generate the following merge pattern: merge x_4 and x_3 to get z_1 ($|z_1| = 15$), merge z_1 and x_1 to get z_2 ($|z_2| = 35$), merge x_2 and x_5 to get z_3 ($|z_3| = 60$), and merge z_2 and z_3 to get the answer z_4. The total number of record moves is 205. One can verify that this is an optimal merge pattern for the given problem instance.

The merge pattern such as the one just described will be referred to as a *two-way merge pattern* (each merge step involves the merging of two files). The two-way merge patterns can be represented by binary merge trees. Figure 4.11 shows a binary merge tree representing the optimal merge pattern obtained for the above five files. The leaf nodes are drawn as squares and represent the given five files. These nodes are called *external nodes*. The remaining nodes are drawn as circles and are called *internal nodes*. Each internal node has exactly two children, and it represents the file obtained by merging the files represented by its two children. The number in each node is the length (i.e., the number of records) of the file represented by that node.

The external node x_4 is at a distance of 3 from the root node z_4 (a node at level i is at a distance of $i - 1$ from the root). Hence, the records of file x_4 are moved three times, once to get z_1, once again to get z_2, and finally one more time to get z_4. If d_i is the distance from the root to the external

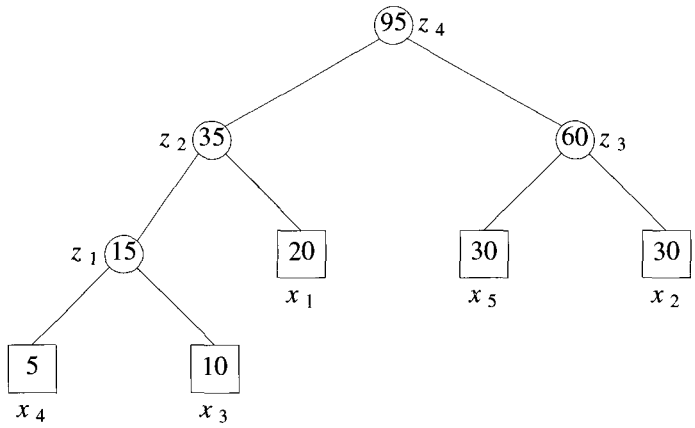


Figure 4.11 Binary merge tree representing a merge pattern

node for file x_i and q_i, the length of x_i is then the total number of record moves for this binary merge tree is

$$\sum_{i=1}^{n} d_i q_i$$

This sum is called the *weighted external path length* of the tree.

An optimal two-way merge pattern corresponds to a binary merge tree with minimum weighted external path length. The function Tree of Algorithm 4.13 uses the greedy rule stated earlier to obtain a two-way merge tree for n files. The algorithm has as input a list *list* of n trees. Each node in a tree has three fields, *lchild*, *rchild*, and *weight*. Initially, each tree in *list* has exactly one node. This node is an external node and has *lchild* and *rchild* fields zero whereas *weight* is the length of one of the n files to be merged. During the course of the algorithm, for any tree in *list* with root node t, $t \rightarrow weight$ is the length of the merged file it represents ($t \rightarrow weight$ equals the sum of the lengths of the external nodes in tree t). Function Tree uses two functions, Least(*list*) and Insert(*list*, t). Least(*list*) finds a tree in *list* whose root has least *weight* and returns a pointer to this tree. This tree is removed from *list*. Insert(*list*, t) inserts the tree with root t into *list*. Theorem 4.10 shows that Tree (Algorithm 4.13) generates an optimal two-way merge tree.

```
    treenode = record {
        treenode * lchild; treenode * rchild;
        integer weight;
    };

1   Algorithm Tree(n)
2   // list is a global list of n single node
3   // binary trees as described above.
4   {
5       for i := 1 to n - 1 do
6       {
7           pt := new treenode; // Get a new tree node.
8           (pt → lchild) := Least(list); // Merge two trees with
9           (pt → rchild) := Least(list); // smallest lengths.
10          (pt → weight) := ((pt → lchild) → weight)
11                  +((pt → rchild) → weight);
12          Insert(list, pt);
13      }
14      return Least(list); // Tree left in list is the merge tree.
15  }
```

Algorithm 4.13 Algorithm to generate a two-way merge tree

Example 4.10 Let us see how algorithm Tree works when *list* initially represents six files with lengths $(2, 3, 5, 7, 9, 13)$. Figure 4.12 shows *list* at the end of each iteration of the **for** loop. The binary merge tree that results at the end of the algorithm can be used to determine which files are merged. Merging is performed on those files which are lowest (have the greatest depth) in the tree. □

The main **for** loop in Algorithm 4.13 is executed $n - 1$ times. If *list* is kept in nondecreasing order according to the *weight* value in the roots, then Least(*list*) requires only $O(1)$ time and Insert($list, t$) can be done in $O(n)$ time. Hence the total time taken is $O(n^2)$. In case *list* is represented as a minheap in which the root value is less than or equal to the values of its children (Section 2.4), then Least(*list*) and Insert($list, t$) can be done in $O(\log n)$ time. In this case the computing time for Tree is $O(n \log n)$. Some speedup may be obtained by combining the Insert of line 12 with the Least of line 9.

Theorem 4.10 If *list* initially contains $n \geq 1$ single node trees with *weight* values $(q_1, q_2, \ldots, q_n)$, then algorithm Tree generates an optimal two-way merge tree for n files with these lengths.

Proof: The proof is by induction on n. For $n = 1$, a tree with no internal nodes is returned and this tree is clearly optimal. For the induction hypothesis, assume the algorithm generates an optimal two-way merge tree for all $(q_1, q_2, \ldots, q_m)$, $1 \leq m < n$. We show that the algorithm also generates optimal trees for all $(q_1, q_2, \ldots, q_n)$. Without loss of generality, we can assume that $q_1 \leq q_2 \leq \cdots \leq q_n$ and q_1 and q_2 are the values of the *weight* fields of the trees found by algorithm Least in lines 8 and 9 during the first iteration of the **for** loop. Now, the subtree T of Figure 4.13 is created. Let T' be an optimal two-way merge tree for $(q_1, q_2, \ldots, q_n)$. Let p be an internal node of maximum distance from the root. If the children of p are not q_1 and q_2, then we can interchange the present children with q_1 and q_2 without increasing the weighted external path length of T'. Hence, T is also a subtree in an optimal merge tree. If we replace T in T' by an external node with weight $q_1 + q_2$, then the resulting tree T'' is an optimal merge tree for $(q_1 + q_2, q_3, \ldots, q_n)$. From the induction hypothesis, after replacing T by the external node with value $q_1 + q_2$, function Tree proceeds to find an optimal merge tree for $(q_1 + q_2, q_3, \ldots, q_n)$. Hence, Tree generates an optimal merge tree for $(q_1, q_2, \ldots, q_n)$. □

The greedy method to generate merge trees also works for the case of k-ary merging. In this case the corresponding merge tree is a k-ary tree. Since all internal nodes must have degree k, for certain values of n there is no corresponding k-ary merge tree. For example, when $k = 3$, there is no k-ary merge tree with $n = 2$ external nodes. Hence, it is necessary to introduce a certain number of dummy external nodes. Each dummy node is assigned a q_i of zero. This dummy value does not affect the weighted external path length of the resulting k-ary tree. Exercise 2 shows that a k-ary tree with all internal nodes having degree k exists only when the number of external nodes n satisfies the equality $n \bmod (k-1) = 1$. Hence, at most $k-2$ dummy nodes have to be added. The greedy rule to generate optimal merge trees is: at each step choose k subtrees with least length for merging. Exercise 3 proves the optimality of this rule.

Huffman Codes

Another application of binary trees with minimal weighted external path length is to obtain an optimal set of codes for messages $M_1, \ldots, M_{n+1}$. Each code is a binary string that is used for transmission of the corresponding message. At the receiving end the code is decoded using a decode tree. A decode tree is a binary tree in which external nodes represent messages.

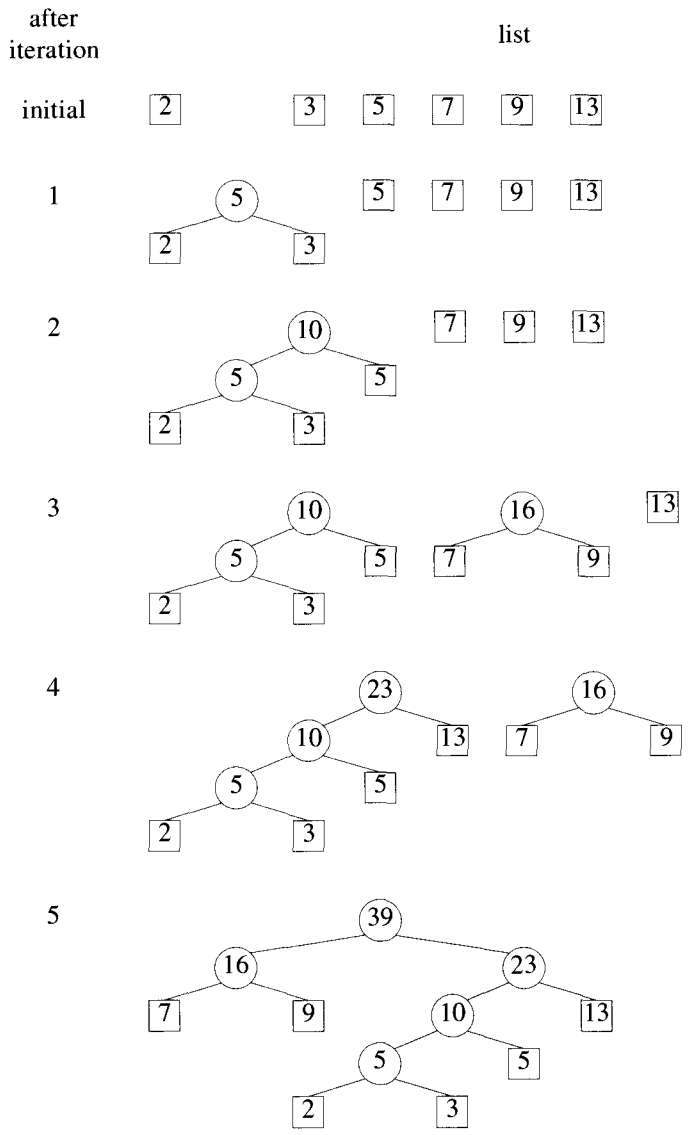


Figure 4.12 Trees in *list* of Tree for Example 4.10

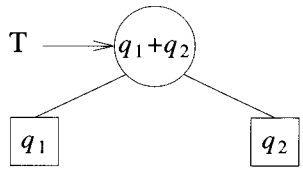


Figure 4.13 The simplest binary merge tree

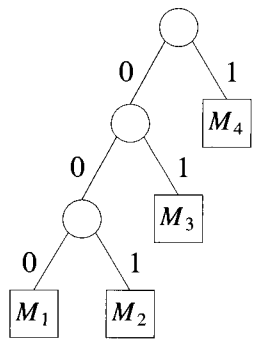


Figure 4.14 Huffman codes

The binary bits in the code word for a message determine the branching needed at each level of the decode tree to reach the correct external node. For example, if we interpret a zero as a left branch and a one as a right branch, then the decode tree of Figure 4.14 corresponds to codes 000, 001, 01, and 1 for messages M_1, M_2, M_3, and M_4 respectively. These codes are called Huffman codes. The cost of decoding a code word is proportional to the number of bits in the code. This number is equal to the distance of the corresponding external node from the root node. If q_i is the relative frequency with which message M_i will be transmitted, then the expected decode time is $\sum_{1 \le i \le n+1} q_i d_i$, where d_i is the distance of the external node for message M_i from the root node. The expected decode time is minimized by choosing code words resulting in a decode tree with minimal weighted external path length! Note that $\sum_{1 \le i \le n+1} q_i d_i$ is also the expected length of a transmitted message. Hence the code that minimizes expected decode time also minimizes the expected length of a message.

EXERCISES

1. Find an optimal binary merge pattern for ten files whose lengths are $28, 32, 12, 5, 84, 53, 91, 35, 3,$ and 11.

2. (a) Show that if all internal nodes in a tree have degree k, then the number n of external nodes is such that $n \bmod (k-1) = 1$.
   (b) Show that for every n such that $n \bmod (k-1) = 1$, there exists a k-ary tree T with n external nodes (in a k-ary tree all nodes have degree at most k). Also show that all internal nodes of T have degree k.

3. (a) Show that if $n \bmod (k-1) = 1$, then the greedy rule described following Theorem 4.10 generates an optimal k-ary merge tree for all $(q_1, q_2, \ldots, q_n)$.
   (b) Draw the optimal three-way merge tree obtained using this rule when $(q_1, q_2, \ldots, q_{11}) = (3, 7, 8, 9, 15, 16, 18, 20, 23, 25, 28)$.

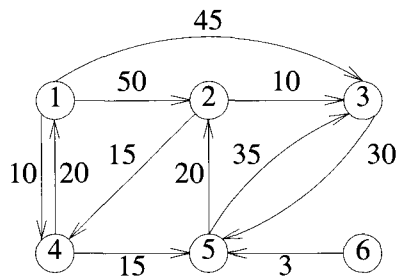
4. Obtain a set of optimal Huffman codes for the messages $(M_1, \ldots, M_7)$ with relative frequencies $(q_1, \ldots, q_7) = (4, 5, 7, 8, 10, 12, 20)$. Draw the decode tree for this set of codes.

5. Let T be a decode tree. An optimal decode tree minimizes $\sum q_i d_i$. For a given set of q's, let D denote all the optimal decode trees. For any tree $T \in D$, let $L(T) = \max\{d_i\}$ and let $\mathrm{SL}(T) = \sum d_i$. Schwartz has shown that there exists a tree $T^* \in D$ such that $L(T^*) = \min_{T \in D}\{L(T)\}$ and $\mathrm{SL}(T^*) = \min_{T \in D}\{SL(T)\}$.

   (a) For $(q_1, \ldots, q_8) = (1, 1, 2, 2, 4, 4, 4, 4)$ obtain trees $T1$ and $T2$ such that $L(T1) > L(T2)$.
   (b) Using the data of a, obtain $T1$ and $T2 \in D$ such that $L(T1) = L(T2)$ but $SL(T1) > SL(T2)$.
   (c) Show that if the subalgorithm Least used in algorithm Tree is such that in case of a tie it returns the tree with least depth, then Tree generates a tree with the properties of T^*.

4.8 SINGLE-SOURCE SHORTEST PATHS

Graphs can be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway. The edges can then be assigned weights which may be either the distance between the two cities connected by the edge or the average time to drive along that section of highway. A motorist wishing to drive from city A to B would be interested in answers to the following questions:



(a) Graph

Path	*Length*
1) 1, 4	10
2) 1, 4, 5	25
3) 1, 4, 5, 2	45
4) 1, 3	45

(b) Shortest paths from 1

Figure 4.15 Graph and shortest paths from vertex 1 to all destinations

- Is there a path from A to B?
- If there is more than one path from A to B, which is the shortest path?

The problems defined by these questions are special cases of the path problem we study in this section. The length of a path is now defined to be the sum of the weights of the edges on that path. The starting vertex of the path is referred to as the *source*, and the last vertex the *destination*. The graphs are digraphs to allow for one-way streets. In the problem we consider, we are given a directed graph $G = (V, E)$, a weighting function *cost* for the edges of G, and a source vertex v_0. The problem is to determine the shortest paths from v_0 to *all* the remaining vertices of G. It is assumed that all the weights are positive. The shortest path between v_0 and some other node v is an ordering among a subset of the edges. Hence this problem fits the ordering paradigm.

Example 4.11 Consider the directed graph of Figure 4.15(a). The numbers on the edges are the weights. If node 1 is the source vertex, then the shortest path from 1 to 2 is $1, 4, 5, 2$. The length of this path is $10 + 15 + 20 = 45$. Even though there are three edges on this path, it is shorter than the path $1, 2$ which is of length 50. There is no path from 1 to 6. Figure 4.15(b) lists the shortest paths from node 1 to nodes $4, 5, 2$, and 3, respectively. The paths have been listed in nondecreasing order of path length. □

To formulate a greedy-based algorithm to generate the shortest paths, we must conceive of a multistage solution to the problem and also of an optimization measure. One possibility is to build the shortest paths one by

one. As an optimization measure we can use the sum of the lengths of all paths so far generated. For this measure to be minimized, each individual path must be of minimum length. If we have already constructed i shortest paths, then using this optimization measure, the next path to be constructed should be the next shortest minimum length path. The greedy way (and also a systematic way) to generate the shortest paths from v_0 to the remaining vertices is to generate these paths in nondecreasing order of path length. First, a shortest path to the nearest vertex is generated. Then a shortest path to the second nearest vertex is generated, and so on. For the graph of Figure 4.15(a) the nearest vertex to $v_0 = 1$ is 4 ($cost[1,4] = 10$). The path $1,4$ is the first path generated. The second nearest vertex to node 1 is 5 and the distance between 1 and 5 is 25. The path $1,4,5$ is the next path generated. In order to generate the shortest paths in this order, we need to be able to determine (1) the next vertex to which a shortest path must be generated and (2) a shortest path to this vertex. Let S denote the set of vertices (including v_0) to which the shortest paths have already been generated. For w not in S, let $dist[w]$ be the length of the shortest path starting from v_0, going through only those vertices that are in S, and ending at w. We observe that:

1. If the next shortest path is to vertex u, then the path begins at v_0, ends at u, and goes through only those vertices that are in S. To prove this, we must show that all the intermediate vertices on the shortest path to u are in S. Assume there is a vertex w on this path that is not in S. Then, the v_0 to u path also contains a path from v_0 to w that is of length less than the v_0 to u path. By assumption the shortest paths are being generated in nondecreasing order of path length, and so the shorter path v_0 to w must already have been generated. Hence, there can be no intermediate vertex that is not in S.

2. The destination of the next path generated must be that of vertex u which has the minimum distance, $dist[u]$, among all vertices not in S. This follows from the definition of $dist$ and observation 1. In case there are several vertices not in S with the same $dist$, then any of these may be selected.

3. Having selected a vertex u as in observation 2 and generated the shortest v_0 to u path, vertex u becomes a member of S. At this point the length of the shortest paths starting at v_0, going though vertices only in S, and ending at a vertex w not in S may decrease; that is, the value of $dist[w]$ may change. If it does change, then it must be due to a shorter path starting at v_0 and going to u and then to w. The intermediate vertices on the v_0 to u path and the u to w path must all be in S. Further, the v_0 to u path must be the shortest such path; otherwise $dist[w]$ is not defined properly. Also, the u to w path can be chosen so as not to contain any intermediate vertices. Therefore,

we can conclude that if $dist[w]$ is to change (i.e., decrease), then it is because of a path from v_0 to u to w, where the path from v_0 to u is the shortest such path and the path from u to w is the edge $\langle u, w \rangle$. The length of this path is $dist[u] + cost[u, w]$.

The above observations lead to a simple Algorithm 4.14 for the single-source shortest path problem. This algorithm (known as Dijkstra's algorithm) only determines the lengths of the shortest paths from v_0 to all other vertices in G. The generation of the paths requires a minor extension to this algorithm and is left as an exercise. In the function ShortestPaths (Algorithm 4.14) it is assumed that the n vertices of G are numbered 1 through n. The set S is maintained as a bit array with $S[i] = 0$ if vertex i is not in S and $S[i] = 1$ if it is. It is assumed that the graph itself is represented by its cost adjacency matrix with $cost[i, j]$'s being the weight of the edge $\langle i, j \rangle$. The weight $cost[i, j]$ is set to some large number, ∞, in case the edge $\langle i, j \rangle$ is not in $E(G)$. For $i = j$, $cost[i, j]$ can be set to any nonnegative number without affecting the outcome of the algorithm.

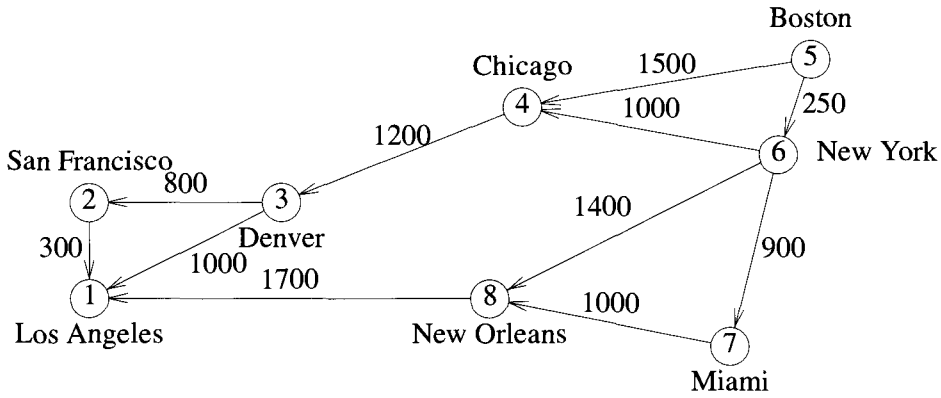
From our earlier discussion, it is easy to see that the algorithm is correct. The time taken by the algorithm on a graph with n vertices is $O(n^2)$. To see this, note that the **for** loop of line 7 in Algorithm 4.14 takes $\Theta(n)$ time. The **for** loop of line 12 is executed $n - 2$ times. Each execution of this loop requires $O(n)$ time at lines 15 and 16 to select the next vertex and again at the **for** loop of line 18 to update $dist$. So the total time for this loop is $O(n^2)$. In case a list t of vertices currently not in s is maintained, then the number of nodes on this list would at any time be $n - num$. This would speed up lines 15 and 16 and the **for** loop of line 18, but the asymptotic time would remain $O(n^2)$. This and other variations of the algorithm are explored in the exercises.

Any shortest path algorithm must examine each edge in the graph at least once since any of the edges could be in a shortest path. Hence, the minimum possible time for such an algorithm would be $\Omega(|E|)$. Since cost adjacency matrices were used to represent the graph, it takes $O(n^2)$ time just to determine which edges are in G, and so any shortest path algorithm using this representation must take $\Omega(n^2)$ time. For this representation then, algorithm ShortestPaths is optimal to within a constant factor. If a change to adjacency lists is made, the overall frequency of the **for** loop of line 18 can be brought down to $O(|E|)$ (since $dist$ can change only for vertices adjacent from u). If $V - S$ is maintained as a red-black tree (see Section 2.4.2), each execution of lines 15 and 16 takes $O(\log n)$ time. Note that a red-black tree supports the following operations in $O(\log n)$ time: insert, delete (an arbitrary element), find-min, and search (for an arbitrary element). Each update in line 21 takes $O(\log n)$ time as well (since an update can be done using a delete and an insertion into the red-black tree). Thus the overall run time is $O((n + |E|) \log n)$.

```
1   Algorithm ShortestPaths(v, cost, dist, n)
2   // dist[j], 1 ≤ j ≤ n, is set to the length of the shortest
3   // path from vertex v to vertex j in a digraph G with n
4   // vertices. dist[v] is set to zero. G is represented by its
5   // cost adjacency matrix cost[1 : n, 1 : n].
6   {
7       for i := 1 to n do
8       { // Initialize S.
9           S[i] := false; dist[i] := cost[v, i];
10      }
11      S[v] := true; dist[v] := 0.0; // Put v in S.
12      for num := 2 to n − 1 do
13      {
14          // Determine n − 1 paths from v.
15          Choose u from among those vertices not
16          in S such that dist[u] is minimum;
17          S[u] := true; // Put u in S.
18          for (each w adjacent to u with S[w] = false) do
19              // Update distances.
20              if (dist[w] > dist[u] + cost[u, w])) then
21                      dist[w] := dist[u] + cost[u, w];
22      }
23  }
```

Algorithm 4.14 Greedy algorithm to generate shortest paths

Example 4.12 Consider the eight vertex digraph of Figure 4.16(a) with cost adjacency matrix as in Figure 4.16(b). The values of *dist* and the vertices selected at each iteration of the **for** loop of line 12 in Algorithm 4.14 for finding all the shortest paths from Boston are shown in Figure 4.17. To begin with, S contains only Boston. In the first iteration of the **for** loop (that is, for $num = 2$), the city u that is not in S and whose $dist[u]$ is minimum is identified to be New York. New York enters the set S. Also the $dist[\,]$ values of Chicago, Miami, and New Orleans get altered since there are shorter paths to these cities via New York. In the next iteration of the **for** loop, the city that enters S is Miami since it has the smallest $dist[\,]$ value from among all the nodes not in S. None of the $dist[\,]$ values are altered. The algorithm continues in a similar fashion and terminates when only seven of the eight vertices are in S. By the definition of *dist*, the distance of the last vertex, in this case Los Angeles, is correct as the shortest path from Boston to Los Angeles can go through only the remaining six vertices. □



(a) Digraph

	1	2	3	4	5	6	7	8
1	0							
2	300	0						
3	100	800	0					
4			1200	0				
5				1500	0	250		
6				1000		0	900	1400
7							0	1000
8	1700							0

(b) Length-adjacency matrix

Figure 4.16 Figures for Example 4.12

One can easily verify that the edges on the shortest paths from a vertex v to all remaining vertices in a connected undirected graph G form a spanning tree of G. This spanning tree is called a *shortest-path spanning tree*. Clearly, this spanning tree may be different for different root vertices v. Figure 4.18 shows a graph G, its minimum-cost spanning tree, and a shortest-path spanning tree from vertex 1.

Iteration	S	Vertex selected	Distance							
			LA	SF	DEN	CHI	BOST	NY	MIA	NO
			[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Initial	--	----	+∞	+∞	+∞	1500	0	250	+∞	+∞
1	{5}	6	+∞	+∞	+∞	1250	0	250	1150	1650
2	{5,6}	7	+∞	+∞	+∞	1250	0	250	1150	1650
3	{5,6,7}	4	+∞	+∞	2450	1250	0	250	1150	1650
4	{5,6,7,4}	8	3350	+∞	2450	1250	0	250	1150	1650
5	{5,6,7,4,8}	3	3350	3250	2450	1250	0	250	1150	1650
6	{5,6,7,4,8,3}	2	3350	3250	2450	1250	0	250	1150	1650
	{5,6,7,4,8,3,2}									

Figure 4.17 Action of ShortestPaths

EXERCISES

1. Use algorithm ShortestPaths to obtain in nondecreasing order the lengths of the shortest paths from vertex 1 to all remaining vertices in the digraph of Figure 4.19.

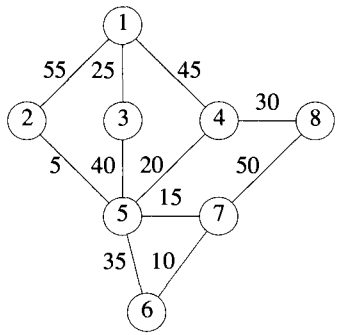
2. Using the directed graph of Figure 4.20 explain why ShortestPaths will not work properly. What is the shortest path between vertices v_1 and v_7 ?

3. Rewrite algorithm ShortestPaths under the following assumptions:

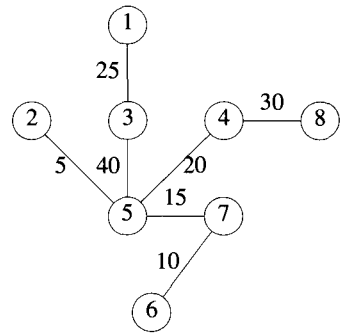
   (a) G is represented by its adjacency lists. The head nodes are HEAD(1),..., HEAD(n) and each list node has three fields: VERTEX, COST, and LINK. COST is the length of the corresponding edge and n the number of vertices in G.

   (b) Instead of representing S, the set of vertices to which the shortest paths have already been found, the set $T = V(G) - S$ is represented using a linked list. What can you say about the computing time of your new algorithm relative to that of ShortestPaths?

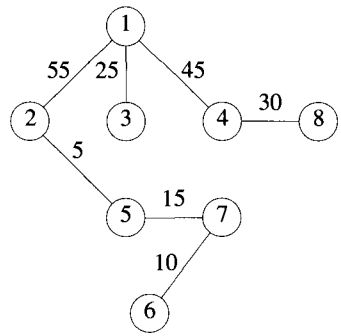
4. Modify algorithm ShortestPaths so that it obtains the shortest paths in addition to the lengths of these paths. What is the computing time of your algorithm?



(a) A Graph



(b) Minimum cost spanning tree



(c) Shortest path spanning tree from vertex 1.

Figure 4.18 Graphs and spanning trees

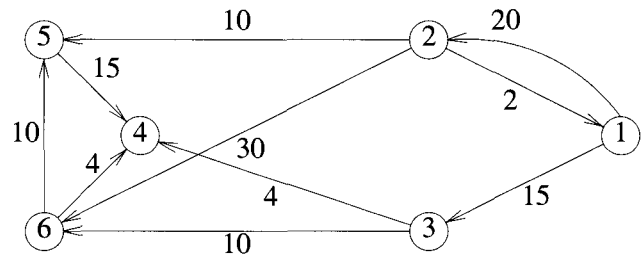


Figure 4.19 Directed graph

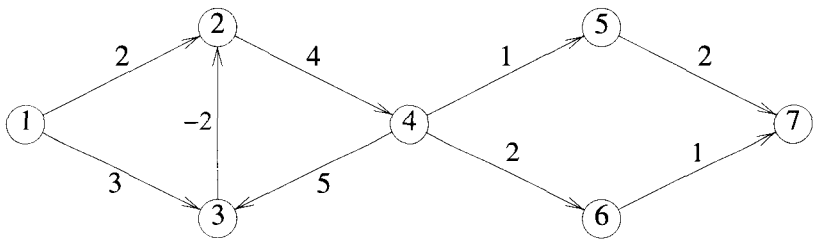


Figure 4.20 Another directed graph

4.9 REFERENCES AND READINGS

The linear time algorithm in Section 4.3 for the tree vertex splitting problem can be found in "Vertex upgrading problems for VLSI," by D. Paik, Ph.D. thesis, Department of Computer Science, University of Minnesota, October 1991.

The two greedy methods for obtaining minimum-cost spanning trees are due to R. C. Prim and J. B. Kruskal, respectively.

An $O(e \log \log v)$ time spanning tree algorithm has been given by A. C. Yao.

The optimal randomized algorithm for minimum-cost spanning trees presented in this chapter appears in "A randomized linear-time algorithm for finding minimum spanning trees," by P. N. Klein and R. E. Tarjan, in *Proceedings of the 26th Annual Symposium on Theory of Computing*, 1994, pp. 9–15. See also "A randomized linear-time algorithm to find minimum spanning trees," by D. R. Karger, P. N. Klein, and R. E. Tarjan, *Journal of the ACM* 42, no. 2 (1995): 321–328.

Proof of Lemma 4.3 can be found in "Verification and sensitivity analysis of minimum spanning trees in linear time," by B. Dixon, M. Rauch, and R. E. Tarjan, *SIAM Journal on Computing* 21 (1992): 1184–1192, and in "A simple minimum spanning tree verification algorithm," by V. King, *Proceedings of the Workshop on Algorithms and Data Structures*, 1995.

A very nearly linear time algorithm for minimum-cost spanning trees appears in "Efficient algorithms for finding minimum spanning trees in undirected and directed graphs," by H. N. Gabow, Z. Galil, T. Spencer, and R. E. Tarjan, *Combinatorica* 6 (1986): 109–122.

A linear time algorithm for minimum-cost spanning trees on a stronger model where the edge weights can be manipulated in their binary form is given in "Trans-dichotomous algorithms for minimum spanning trees and shortest paths," by M. Fredman and D. E. Willard, in *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, 1990, pp. 719–725.

The greedy method developed here to optimally store programs on tapes was first devised for a machine scheduling problem. In this problem n jobs have to be scheduled on m processors. Job i takes t_i amount of time. The time at which a job finishes is the sum of the job times for all jobs preceding and including job i. The average finish time corresponds to the mean access time for programs on tapes. The $(m!)^{n/m}$ schedules referred to in Theorem 4.9 are known as SPT (shortest processing time) schedules. The rule to generate SPT schedules as well as the rule of Exercise 4 (Section 4.6) are due to W. E. Smith.

The greedy algorithm for generating optimal merge trees is due to D. Huffman.

For a given set $\{q_1, \ldots, q_n\}$ there are many sets of Huffman codes minimizing $\sum q_i d_i$. From amongst these code sets there is one that has minimum $\sum d_i$ and minimum max $\{d_i\}$. An algorithm to obtain this code set was given by E. S. Schwartz.

The shortest-path algorithm of the text is due to E. W. Dijkstra.

For planar graphs, the shortest-path problem can be solved in linear time as has been shown in "Faster shortest-path algorithms for planar graphs," by P. Klein, S. Rao, and M. Rauch, in *Proceedings of the ACM Symposium on Theory of Computing*, 1994.

The relationship between greedy methods and matroids is discussed in *Combinatorial Optimization*, by E. Lawler, Holt, Rinehart and Winston, 1976.

4.10 ADDITIONAL EXERCISES

1. [Coin changing] Let $A_n = \{a_1, a_2, \ldots, a_n\}$ be a finite set of distinct coin types (for example, $a_1 = 50¢$, $a_2 = 25¢$, $a_3 = 10¢$, and so on.) We can assume each a_i is an integer and $a_1 > a_2 > \cdots > a_n$. Each type is available in unlimited quantity. The coin-changing problem is to make up an exact amount C using a minimum total number of coins. C is an integer > 0.

(a) Show that if $a_n \neq 1$, then there exists a finite set of coin types and a C for which there is no solution to the coin-changing problem.

(b) Show that there is always a solution when $a_n = 1$.

(c) When $a_n = 1$, a greedy solution to the problem makes change by using the coin types in the order $a_1, a_2, \ldots, a_n$. When coin type a_i is being considered, as many coins of this type as possible are given. Write an algorithm based on this strategy. Show that this algorithm doesn't necessarily generate solutions that use the minimum total number of coins.

(d) Show that if $A_n = \{k^{n-1}, k^{n-2}, \ldots, k^0\}$ for some $k > 1$, then the greedy method of part (c) always yields solutions with a minimum number of coins.

2. [Set cover] You are given a family S of m sets $S_i, 1 \leq i \leq m$. Denote by $|A|$ the size of set A. Let $|S_i| = j_i$; that is, $S_i = \{s_1, s_2, \ldots, s_{j_i}\}$. A subset $T = \{T_1, T_2, \ldots, T_k\}$ of S is a family of sets such that for each $i, 1 \leq i \leq k$, $T_i = S_r$ for some $r, 1 \leq r \leq m$. The subset T is a *cover* of S iff $\cup T_i = \cup S_i$. The size of T, $|T|$, is the number of sets in T. A minimum cover of S is a cover of smallest size. Consider the following greedy strategy: build T iteratively, at the kth iteration $T = \{T_1, \ldots, T_{k-1}\}$, now add to T a set S_j from S that contains the largest number of elements not already in T, and stop when $\cup T_i = \cup S_i$.

(a) Assume that $\cup S_i = \{1, 2, \ldots, n\}$ and $m < n$. Using the strategy outlined above, write an algorithm to obtain set covers. How much time and space does your algorithm require?

(b) Show that the greedy strategy above doesn't necessarily obtain a minimum set cover.

(c) Suppose now that a minimum cover is defined to be one for which $\sum_{i=1}^{k} |T_i|$ is minimum. Does the above strategy always find a minimum cover?

3. [Node cover] Let $G = (V, E)$ be an undirected graph. A node cover of G is a subset U of the vertex set V such that every edge in E is incident to at least one vertex in U. A minimum node cover is one with the fewest number of vertices. Consider the following greedy algorithm for this problem:

```
1   Algorithm Cover(V, E)
2   {
3       U := ∅;
4       repeat
5       {
6           Let q be a vertex from V of maximum degree;
7           Add q to U; Eliminate q from V;
8           E := E − {(x, y) such that x = q or y = q};
9       } until (E = ∅); // U is the node cover.
10  }
```

Does this algorithm always generate a minimum node cover?

4. [Traveling salesperson] Let G be a directed graph with n vertices. Let $length(u, v)$ be the length of the edge $\langle u, v \rangle$. A path starting at a given vertex v_0, going through every other vertex exactly once, and finally returning to v_0 is called a *tour*. The length of a tour is the sum of the lengths of the edges on the path defining the tour. We are concerned with finding a tour of minimum length. A greedy way to construct such a tour is: let (P, v) represent the path so far constructed; it starts at v_0 and ends at v. Initially P is empty and $v = v_0$, if all vertices in G are on P, then include the edge $\langle v, v_0 \rangle$ and stop; otherwise include an edge $\langle v, w \rangle$ of minimum length among all edges from v to a vertex w not on P. Show that this greedy method doesn't necessarily generate a minimum-length tour.

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

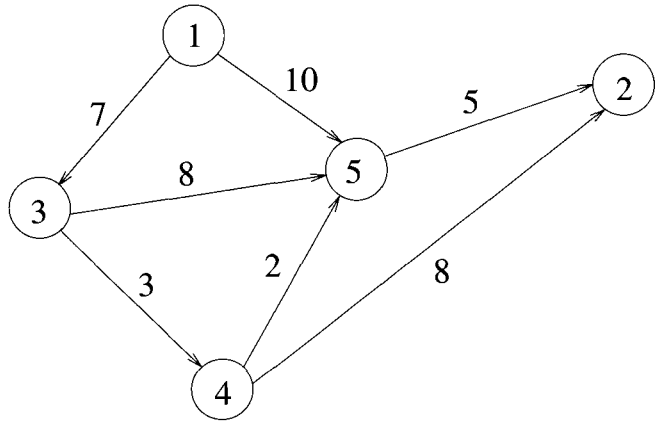


Figure 5.1 Graph for Exercise 2 (Section 5.1)

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.

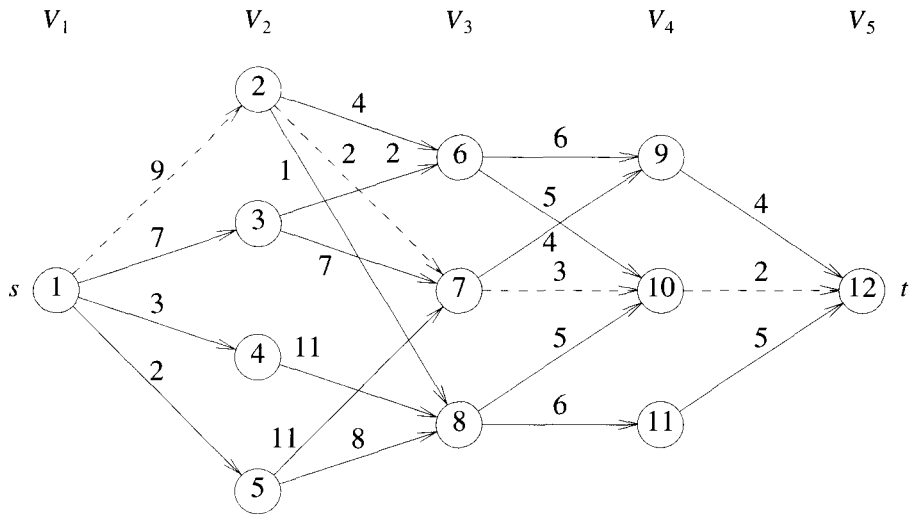


Figure 5.2 Five-stage graph

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

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

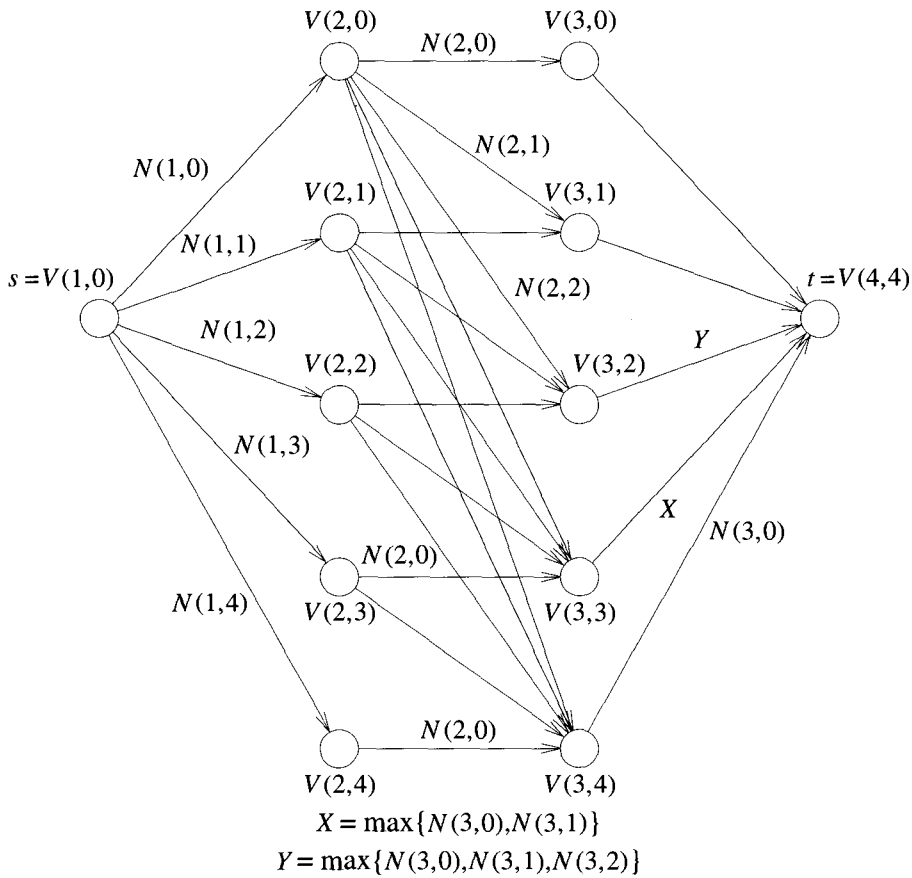


Figure 5.3 Four-stage graph corresponding to a three-project problem

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

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.

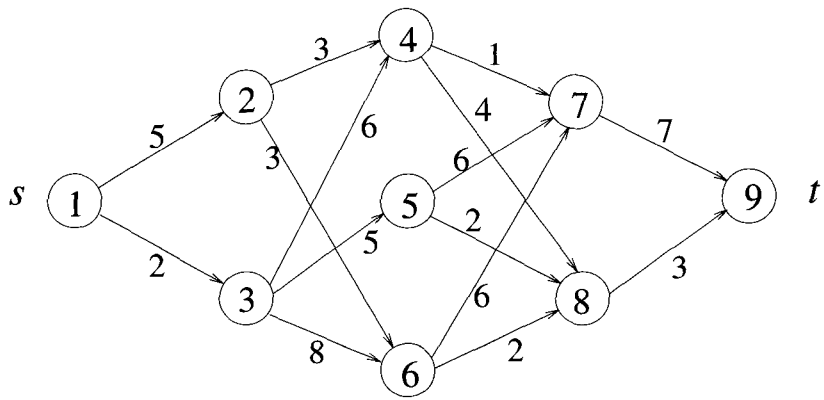


Figure 5.4 Multistage graph for Exercise 1

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

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

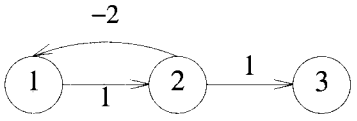
The following example shows that (5.8) is not true for graphs with cycles of negative length.

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.



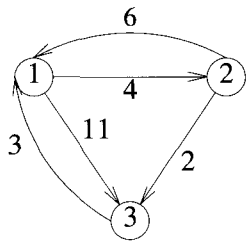
$$\begin{bmatrix} 0 & 1 & \infty \\ -2 & 0 & 1 \\ \infty & \infty & 0 \end{bmatrix}$$

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?

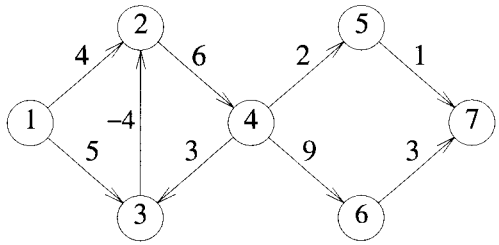


Figure 5.7 Graph for Exercise 1

   (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.

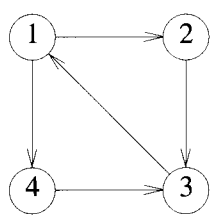


Figure 5.8 Graph for Exercise 3

   (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 $dist[3]$.

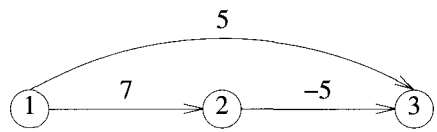


Figure 5.9 Directed graph with a negative-length edge

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 $dist$:

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

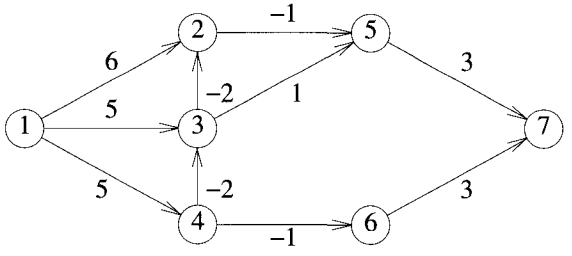
This recurrence can be used to compute $dist^{k}$ from $dist^{k-1}$, for $k = 2, 3, \ldots, n-1$.

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

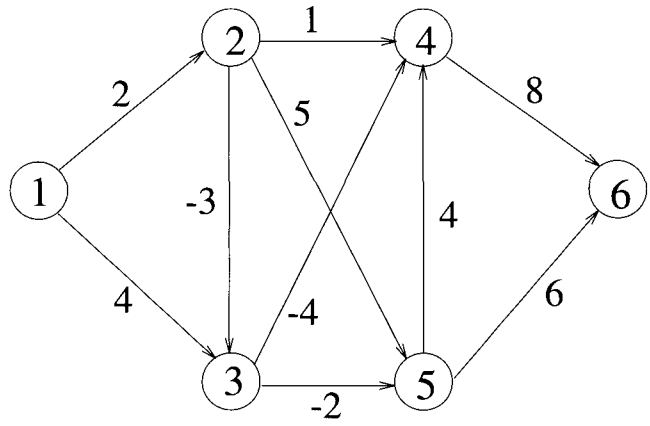


Figure 5.11 Graph for Exercise 1

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 (*)

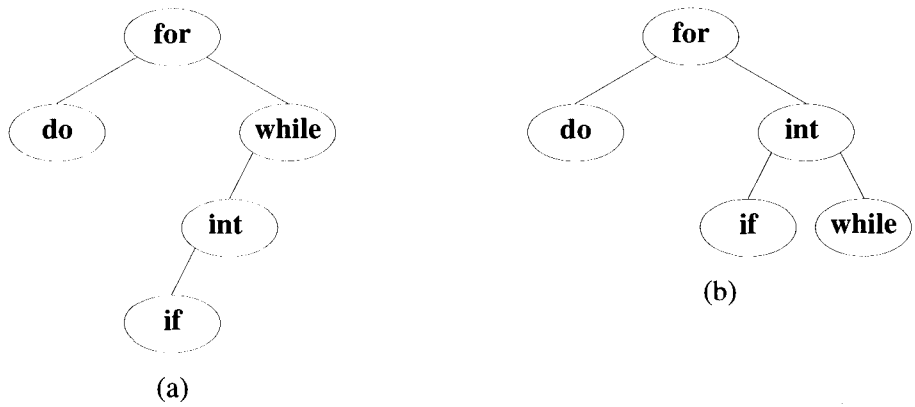


Figure 5.12 Two possible binary search trees

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.

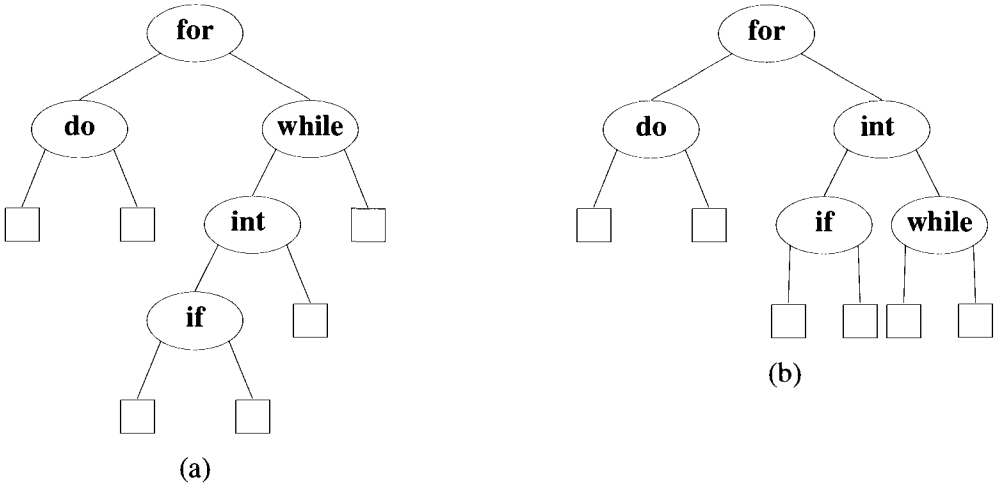


Figure 5.13 Binary search trees of Figure 5.12 with external nodes added

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

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

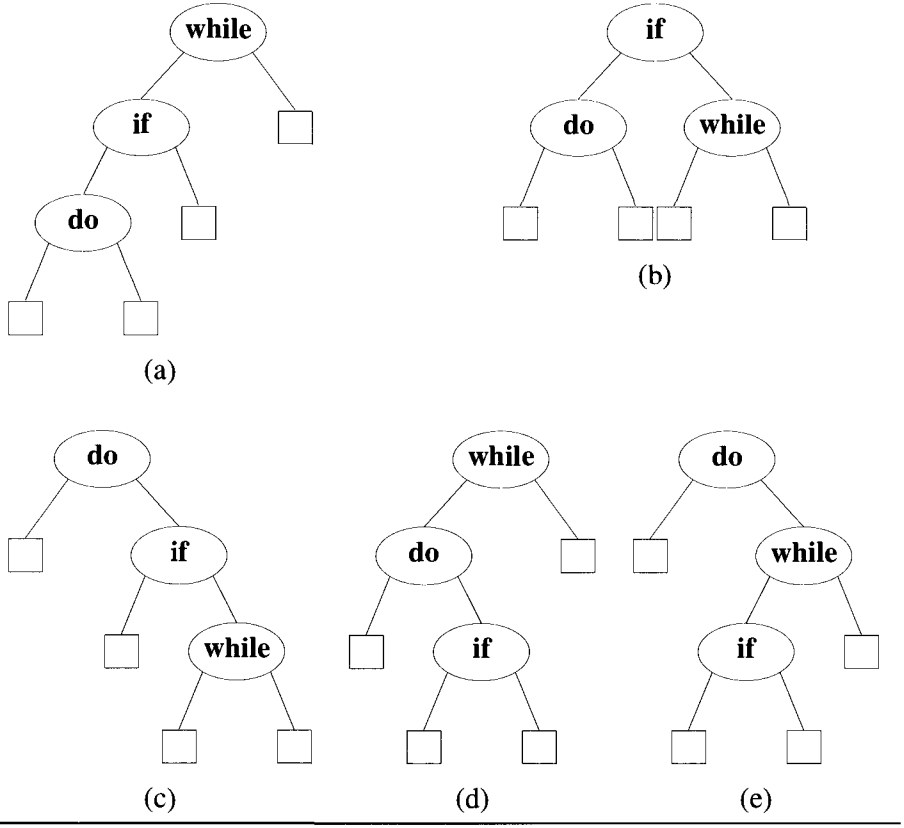


Figure 5.14 Possible binary search trees for the identifier set {**do, if, while**}

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.

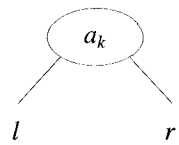


Figure 5.15 An optimal binary search tree with root a_k

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 $r(0,4)$

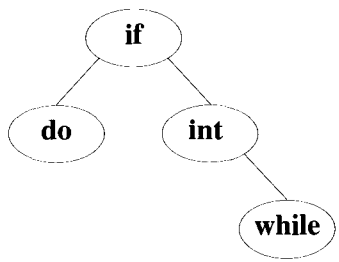


Figure 5.17 Optimal search tree for Example 5.18

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. □

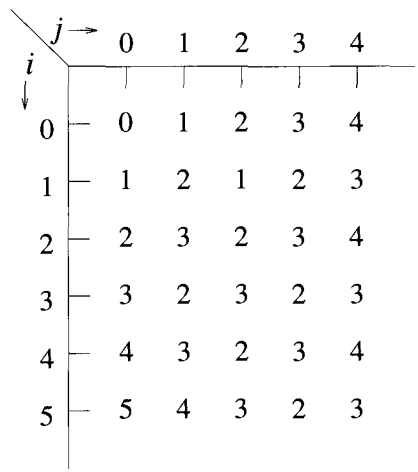


Figure 5.18 Cost table for Example 5.20

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

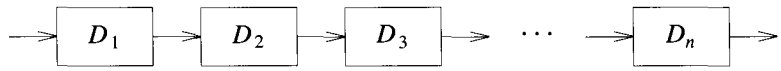


Figure 5.19 n devices D_i, $1 \leq i \leq n$, connected in series

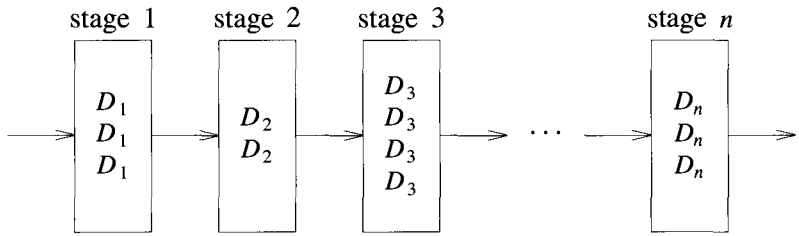


Figure 5.20 Multiple devices connected in parallel in each stage

$1-(1-r_i)^{m_i}$. 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

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

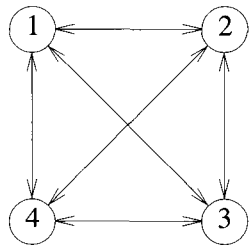
Generalizing (5.20), we obtain (for $i \notin S$)

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

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. □

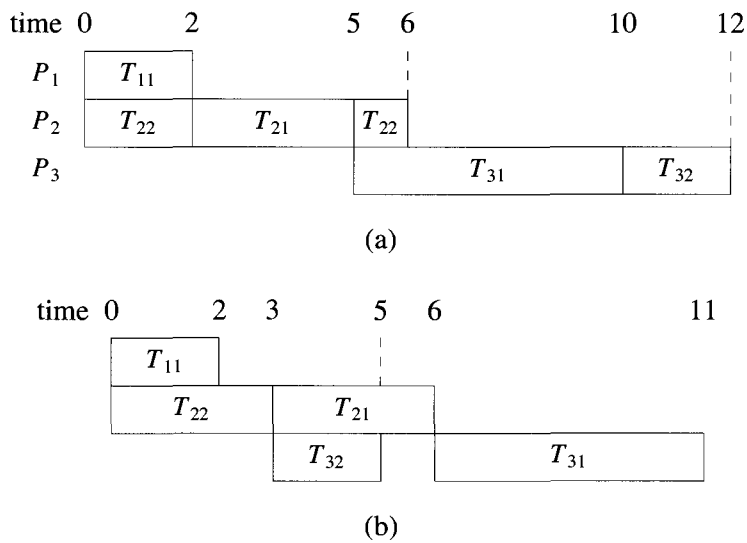


Figure 5.22 Two possible schedules for Example 5.27

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

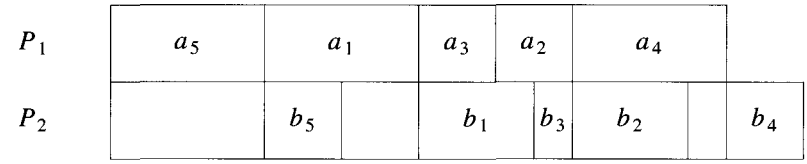


Figure 5.23 A schedule

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 6

BASIC TRAVERSAL AND SEARCH TECHNIQUES

The techniques to be discussed in this chapter are divided into two categories. The first category includes techniques applicable only to binary trees. As described, these techniques involve examining every node in the given data object instance. Hence, these techniques are referred to as *traversal methods.* The second category includes techniques applicable to graphs (and hence also to trees and binary trees). These may not examine all vertices and so are referred to only as *search methods.* During a traversal or search the fields of a node may be used several times. It may be necessary to distinguish certain uses of the fields of a node. During these uses, the node is said to be *visited.* Visiting a node may involve printing out its data field, evaluating the operation specified by the node in the case of a binary tree representing an expression, setting a mark bit to one or zero, and so on. Since we are describing traversals and searches of trees and graphs independently of their application, we use the term "visited" rather than the term for the specific function performed on the node at this time.

6.1 TECHNIQUES FOR BINARY TREES

The solution to many problems involves the manipulation of binary trees, trees, or graphs. Often this manipulation requires us to determine a vertex (node) or a subset of vertices in the given data object that satisfies a given property. For example, we may wish to find all vertices in a binary tree with a data value less than x or we may wish to find all vertices in a given graph G that can be reached from another given vertex v. The determination of this subset of vertices satisfying a given property can be carried out by systematically examining the vertices of the given data object. This often takes the form of a search in the data object. When the search necessarily

```
treenode = record
{
    Type data; // Type is the data type of data.
    treenode *lchild; treenode *rchild;
}

1   Algorithm InOrder(t)
2   // t is a binary tree. Each node of t has
3   // three fields: lchild, data, and rchild.
4   {
5       if t ≠ 0 then
6       {
7           InOrder(t → lchild);
8           Visit(t);
9           InOrder(t → rchild);
10      }
11  }
```

Algorithm 6.1 Recursive formulation of inorder traversal

involves the examination of every vertex in the object being searched, it is called a *traversal.*

We have already seen an example of a problem whose solution required a search of a binary tree. In Section 5.5 we presented an algorithm to search a binary search tree for an identifier x. This algorithm is not a traversal algorithm as it does not examine every vertex in the search tree. Sometimes, we may wish to traverse a binary search tree (e.g., when we wish to list out all the identifiers in the tree). Algorithms for this are studied in this chapter.

There are many operations that we want to perform on binary trees. One that arises frequently is traversing a tree, or visiting each node in the tree exactly once. A traversal produces a linear order for the information in a tree. This linear order may be familiar and useful. When traversing a binary tree, we want to treat each node and its subtrees in the same fashion. If we let L, D, and R stand for moving left, printing the data, and moving right when at a node, then there are six possible combinations of traversal: *LDR, LRD, DLR, DRL, RDL,* and *RLD.* If we adopt the convention that we traverse left before right, then only three traversals remain: *LDR, LRD,* and *DLR.* To these we assign the names *inorder*, *postorder*, and *preorder.* Recursive functions for these three traversals are given in Algorithms 6.1 and 6.2.

```
1   Algorithm PreOrder(t)
2   // t is a binary tree. Each node of t has
3   // three fields: lchild, data, and rchild.
4   {
5       if t ≠ 0 then
6       {
7           Visit(t);
8           PreOrder(t → lchild);
9           PreOrder(t → rchild);
10      }
11  }
```

```
1   Algorithm PostOrder(t)
2   // t is a binary tree. Each node of t has
3   // three fields: lchild, data, and rchild.
4   {
5       if t ≠ 0 then
6       {
7           PostOrder(t → lchild);
8           PostOrder(t → rchild);
9           Visit(t);
10      }
11  }
```

Algorithm 6.2 Preorder and postorder traversals

Figure 6.1 shows a binary tree and Figure 6.2 traces how InOrder works on it. This trace assumes that visiting a node requires only the printing of its *data* field. The output resulting from this traversal is FDHGIBEAC. With Visit(t) replaced by a printing statement, the application of Algorithm 6.2 to the binary tree of Figure 6.1 results in the outputs ABDFGHIEC and FHIGDEBCA, respectively.

Theorem 6.1 Let $T(n)$ and $S(n)$ respectively represent the time and space needed by any one of the traversal algorithms when the input tree t has $n \geq 0$ nodes. If the time and space needed to visit a node are $\Theta(1)$, then $T(n) = \Theta(n)$ and $S(n) = O(n)$.

Proof: Each traversal can be regarded as a walk through the binary tree. During this walk, each node is reached three times: once from its parent (or as the start node in case the node is the root), once on returning from its left

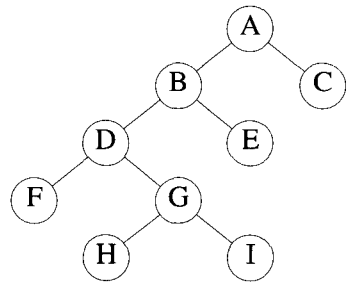


Figure 6.1 A binary tree

call of InOrder	value in root	action
main	A	
1	B	
2	D	
3	F	
4	—	print ('F')
4	—	print ('D')
3	G	
4	H	
5	—	print ('H')
5	—	print ('G')
4	I	
5	—	print ('I')
5	—	print ('B')
2	E	
3	—	print ('E')
3	—	print ('A')
1	C	
2	—	print ('C')
2	—	

Figure 6.2 Inorder traversal of the binary tree of Figure 6.1

subtree, and once on returning from its right subtree. In each of these three times a constant amount of work is done. So, the total time taken by the traversal is $\Theta(n)$. The only additional space needed is that for the recursion stack. If t has depth d, then this space is $\Theta(d)$. For an n-node binary tree, $d \leq n$ and so $S(n) = O(n)$. □

EXERCISES

Unless otherwise stated, all binary trees are represented using nodes with three fields: *lchild*, *data*, and *rchild*.

1. Give an algorithm to count the number of leaf nodes in a binary tree t. What is its computing time?
2. Write an algorithm SwapTree(t) that takes a binary tree and swaps the left and right children of every node. An example is given in Figure 6.3. Use one of the three traversal methods discussed in Section 6.1.

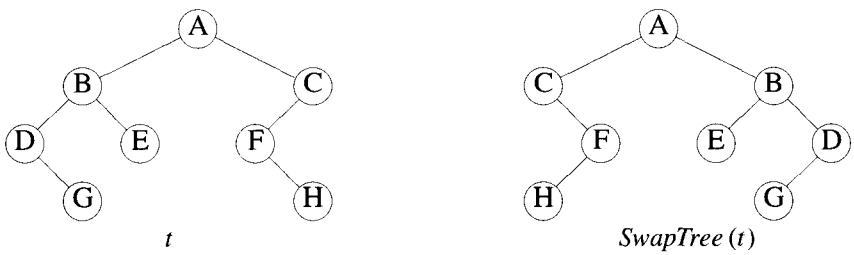


Figure 6.3 Swapping left and right children

3. Use one of the three traversal methods discussed in Section 6.1 to obtain an algorithm Equiv(t, u) that determines whether the binary trees t and u are equivalent. Two binary trees t and u are equivalent if and only if they are structurally equivalent and if the data in the corresponding nodes of t and u are the same.
4. Show the following:
   (a) Inorder and postorder sequences of a binary tree uniquely define the binary tree.
   (b) Inorder and preorder sequences of a binary tree uniquely define the binary tree.

(c) Preorder and postorder sequences of a binary tree do not uniquely define the binary tree.

5. In the proof of Theorem 6.1, show, using induction, that $T(n) \leq c_2 n + c_1$ (where c_2 is a constant $\geq 2c_1$).

6. Write a function to construct the binary tree with a given inorder sequence I and a given postorder sequence P. What is the complexity of your function?

7. Do Exercise 6 for a given inorder and preorder sequence.

8. Write a nonrecursive algorithm for the preorder traversal of a binary tree t. Your algorithm may use a stack. What are the time and space requirements of your algorithm?

9. Do Exercise 8 for postorder as well as inorder traversals.

10. [Triple-order traversal] A triple-order traversal of a binary tree t is defined recursively by Algorithm 6.3. A very simple nonrecursive algorithm for such a traversal is given in Algorithm 6.4. In this algorithm p, q, and r point respectively to the present node, previously visited node, and next node to visit. The algorithm assumes that $t \neq 0$ and that an empty subtree of node p is represented by a link to p rather than a zero. Prove that Algorithm 6.4 is correct. (*Hint*: Three links, $lchild, rchild$, and one from its parent, are associated with each node s. Each time s is visited, the links are rotated counterclockwise, and so after three visits they are restored to the original configuration and the algorithm backs up the tree.)

11. [Level-order traversal] In a level-order traversal of a binary tree t all nodes on level i are visited before any node on level $i+1$ is visited. Within a level, nodes are visited left to right. In level-order the nodes of the tree of Figure 6.1 are visited in the order ABCDEFGHI. Write an algorithm Level(t) to traverse the binary tree t in level order. How much time and space are needed by your algorithm?

6.2 TECHNIQUES FOR GRAPHS

A fundamental problem concerning graphs is the reachability problem. In its simplest form it requires us to determine whether there exists a path in the given graph $G = (V, E)$ such that this path starts at vertex v and ends at vertex u. A more general form is to determine for a given starting vertex $v \in V$ all vertices u such that there is a path from v to u. This latter problem can be solved by starting at vertex v and systematically searching the graph G for vertices that can be reached from v. We describe two search methods for this.

```
1   Algorithm Triple(t)
2   {
3       if t ≠ 0 then
4       {
5           Visit(t);
6           Triple(t → lchild);
7           Visit(t);
8           Triple(t → rchild);
9           Visit(t);
10      }
11  }
```

Algorithm 6.3 Triple-order traversal for Exercise 10

```
1   Algorithm Trip(t);
2   // It is assumed that lchild and rchild fields are > 0.
3   {
4       p := t; q := −1;
5       while (p ≠ −1) do
6       {
7           Visit(p);
8           r := (p → lchild); (p → lchild) := (p → rchild);
9           (p → rchild) := q; q := p; p := r;
10      }
11  }
```

Algorithm 6.4 A nonrecursive algorithm for the triple-order traversal for Exercise 10

6.2.1 Breadth First Search and Traversal

In breadth first search we start at a vertex v and mark it as having been reached (visited). The vertex v is at this time said to be unexplored. A vertex is said to have been explored by an algorithm when the algorithm has visited all vertices adjacent from it. All unvisited vertices adjacent from v are visited next. These are new unexplored vertices. Vertex v has now been explored. The newly visited vertices haven't been explored and are put onto the end of a list of unexplored vertices. The first vertex on this list is the next to be explored. Exploration continues until no unexplored vertex is left. The list of unexplored vertices operates as a queue and can be represented using any of the standard queue representations (see Section 2.1). BFS (Algorithm 6.5) describes, in pseudocode, the details of the search. It makes use of the queue representation given in Section 2.1 (Algorithm 2.3).

Example 6.1 Let us try out the algorithm on the undirected graph of Figure 6.4(a). If the graph is represented by its adjacency lists as in Figure 6.4(c), then the vertices get visited in the order 1, 2, 3, 4, 5, 6, 7, 8. A breadth first search of the directed graph of Figure 6.4(b) starting at vertex 1 results in only the vertices 1, 2, and 3 being visited. Vertex 4 cannot be reached from 1. □

Theorem 6.2 Algorithm BFS visits all vertices reachable from v.

Proof: Let $G = (V, E)$ be a graph (directed or undirected) and let $v \in V$. We prove the theorem by induction on the length of the shortest path from v to every reachable vertex $w \in V$. The length (i.e., number of edges) of the shortest path from v to a reachable vertex w is denoted by $d(v, w)$.

Basis Step. Clearly, all vertices w with $d(v, w) \leq 1$ get visited.

Induction Hypothesis. Assume that all vertices w with $d(v, w) \leq r$ get visited.

Induction Step. We now show that all vertices w with $d(v, w) = r + 1$ also get visited.

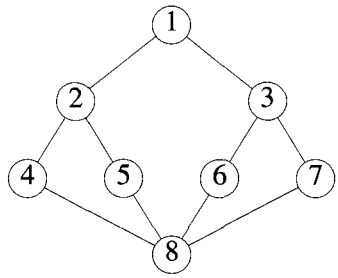
Let w be a vertex in V such that $d(v, w) = r + 1$. Let u be a vertex that immediately precedes w on a shortest v to w path. Then $d(v, u) = r$ and so u gets visited by BFS. We can assume $u \neq v$ and $r \geq 1$. Hence, immediately before u gets visited, it is placed on the queue q of unexplored vertices. The algorithm doesn't terminate until q becomes empty. Hence, u is removed from q at some time and all unvisited vertices adjacent from it get visited in the **for** loop of line 11 of Algorithm 6.5. Hence, w gets visited. □

```
1   Algorithm BFS(v)
2   // A breadth first search of G is carried out beginning
3   // at vertex v. For any node i, visited[i] = 1 if i has
4   // already been visited. The graph G and array visited[ ]
5   // are global; visited[ ] is initialized to zero.
6   {
7       u := v; // q is a queue of unexplored vertices.
8       visited[v] := 1;
9       repeat
10      {
11          for all vertices w adjacent from u do
12          {
13              if (visited[w] = 0) then
14              {
15                  Add w to q; // w is unexplored.
16                  visited[w] := 1;
17              }
18          }
19          if q is empty then return; // No unexplored vertex.
20          Delete u from q; // Get first unexplored vertex.
21      } until(false);
22  }
```

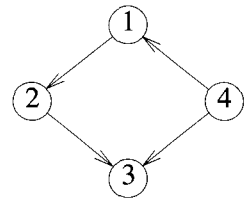
Algorithm 6.5 Pseudocode for breadth first search

Theorem 6.3 Let $T(n,e)$ and $S(n,e)$ be the maximum time and maximum additional space taken by algorithm BFS on any graph G with n vertices and e edges. $T(n,e) = \Theta(n+e)$ and $S(n,e) = \Theta(n)$ if G is represented by its adjacency lists. If G is represented by its adjacency matrix, then $T(n,e) = \Theta(n^2)$ and $S(n,e) = \Theta(n)$.

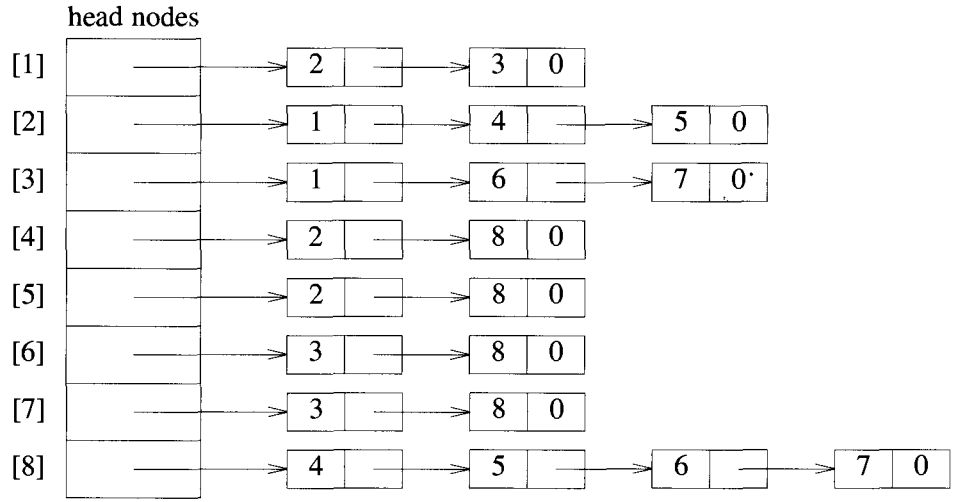
Proof: Vertices get added to the queue only in line 15 of Algorithm 6.5. A vertex w can get onto the queue only if $visited[w] = 0$. Immediately following w's addition to the queue, $visited[w]$ is set to 1 (line 16). Hence, each vertex can get onto the queue at most once. Vertex v never gets onto the queue and so at most $n-1$ additions are made. The queue space needed is at most $n-1$. The remaining variables take $O(1)$ space. Hence, $S(n,e) = O(n)$. If G is an n-vertex graph with v connected to the remaining $n-1$ vertices, then all $n-1$ vertices adjacent from v are on the queue at the same time. Furthermore, $\Theta(n)$ space is needed for the array *visited*. Hence $S(n,e) = \Theta(n)$. This result is independent of whether adjacency matrices or lists are used.



(a) Undirected graph G



(b) Directed graph



(c) Adjacency list for G

Figure 6.4 Example graphs and adjacency lists

```
1   Algorithm BFT(G, n)
2   // Breadth first traversal of G
3   {
4       for i := 1 to n do  // Mark all vertices unvisited.
5           visited[i] := 0;
6       for i := 1 to n do
7           if (visited[i] = 0) then BFS(i);
8   }
```

Algorithm 6.6 Breadth first graph traversal

If adjacency lists are used, then all vertices adjacent from u can be determined in time $d(u)$, where $d(u)$ is the degree of u if G is undirected and $d(u)$ is the out-degree of u if G is directed. Hence, when vertex u is being explored, the time for the **for** loop of line 11 of Algorithm 6.5 is $\Theta(d(u))$. Since each vertex in G can be explored at most once, the total time for the **repeat** loop of line 9 is $O(\sum d(u)) = O(e)$. Then $visited[i]$ has to be initialized to 0, $1 \leq i \leq n$. This takes $O(n)$ time. The total time is therefore $O(n+e)$. If adjacency matrices are used, then it takes $\Theta(n)$ time to determine all vertices adjacent from u and the time becomes $O(n^2)$. If G is a graph such that all vertices are reachable from v, then all vertices get explored and the time is at least $O(n+e)$ and $O(n^2)$ respectively. Hence, $T(n,e) = \Theta(n+e)$ when adjacency lists are used, and $T(n,e) = \Theta(n^2)$ when adjacency matrices are used. □

If BFS is used on a connected undirected graph G, then all vertices in G get visited and the graph is traversed. However, if G is not connected, then at least one vertex of G is not visited. A complete traversal of the graph can be made by repeatedly calling BFS each time with a new unvisited starting vertex. The resulting traversal algorithm is known as breadth first traversal (BFT) (see Algorithm 6.6). The proof of Theorem 6.3 can be used for BFT too to show that the time and additional space required by BFT on an n-vertex e-edge graph are $\Theta(n+e)$ and $\Theta(n)$ respectively if adjacency lists are used. If adjacency matrices are used, then the bounds are $\Theta(n^2)$ and $\Theta(n)$ respectively.

6.2.2 Depth First Search and Traversal

A depth first search of a graph differs from a breadth first search in that the exploration of a vertex v is suspended as soon as a new vertex is reached. At

```
1   Algorithm DFS(v)
2   // Given an undirected (directed) graph G = (V, E) with
3   // n vertices and an array visited[ ] initially set
4   // to zero, this algorithm visits all vertices
5   // reachable from v. G and visited[ ] are global.
6   {
7       visited[v] := 1;
8       for each vertex w adjacent from v do
9       {
10          if (visited[w] = 0) then DFS(w);
11      }
12  }
```

Algorithm 6.7 Depth first search of a graph

this time the exploration of the new vertex u begins. When this new vertex has been explored, the exploration of v continues. The search terminates when all reached vertices have been fully explored. This search process is best described recursively as in Algorithm 6.7.

Example 6.2 A depth first search of the graph of Figure 6.4(a) starting at vertex 1 and using the adjacency lists of Figure 6.4(c) results in the vertices being visited in the order 1, 2, 4, 8, 5, 6, 3, 7. □

One can easily prove that DFS visits all vertices reachable from vertex v. If $T(n, e)$ and $S(n, e)$ represent the maximum time and maximum additional space taken by DFS for an n-vertex e-edge graph, then $S(n, e) = \Theta(n)$ and $T(n, e) = \Theta(n + e)$ if adjacency lists are used and $T(n, e) = \Theta(n^2)$ if adjacency matrices are used (see the exercises).

A depth first traversal of a graph is carried out by repeatedly calling DFS, with a new unvisited starting vertex each time. The algorithm for this (DFT) differs from BFT only in that the call to BFS(i) is replaced by a call to DFS(i). The exercises contain some problems that are solved best by BFS and others that are solved best by DFS. Later sections of this chapter also discuss graph problems solved best by DFS.

BFS and DFS are two fundamentally different search methods. In BFS a node is fully explored before the exploration of any other node begins. The next node to explore is the first unexplored node remaining. The exercises examine a search technique (D-search) that differs from BFS only in that

the next node to explore is the most recently reached unexplored node. In DFS the exploration of a node is suspended as soon as a new unexplored node is reached. The exploration of this new node is immediately begun.

EXERCISES

1. Devise an algorithm using the idea of BFS to find a shortest (directed) cycle containing a given vertex v. Prove that your algorithm finds a shortest cycle. What are the time and space requirements of your algorithm?

2. Show that DFS visits all vertices in G reachable from v.

3. Prove that the bounds of Theorem 6.3 hold for DFS.

4. It is easy to see that for any graph G, both DFS and BFS will take almost the same amount of time. However, the space requirements may be considerably different.

   (a) Give an example of an n-vertex graph for which the depth of recursion of DFS starting from a particular vertex v is $n-1$ whereas the queue of BFS has at most one vertex at any given time if BFS is started from the same vertex v.

   (b) Give an example of an n-vertex graph for which the queue of BFS has $n-1$ vertices at one time whereas the depth of recursion of DFS is at most one. Both searches are started from the same vertex.

5. Another way to search a graph is D-search. This method differs from BFS in that the next vertex to explore is the vertex most recently added to the list of unexplored vertices. Hence, this list operates as a stack rather than a queue.

   (a) Write an algorithm for D-search.

   (b) Show that D-search starting from vertex v visits all vertices reachable from v.

   (c) What are the time and space requirements of your algorithm?

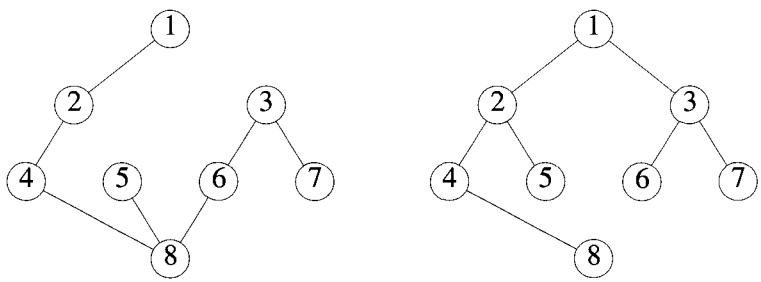
6.3 CONNECTED COMPONENTS AND SPANNING TREES

If G is a connected undirected graph, then all vertices of G will get visited on the first call to BFS (Algorithm 6.5). If G is not connected, then at

least two calls to BFS will be needed. Hence, BFS can be used to determine whether G is connected. Furthermore, all newly visited vertices on a call to BFS from BFT represent the vertices in a connected component of G. Hence the connected components of a graph can be obtained using BFT. For this, BFS can be modified so that all newly visited vertices are put onto a list. Then the subgraph formed by the vertices on this list make up a connected component. Hence, if adjacency lists are used, a breadth first traversal will obtain the connected components in $\Theta(n+e)$ time.

BFT can also be used to obtain the reflexive transitive closure matrix of an undirected graph G. If A^* is this matrix, then $A^*(i,j) = 1$ iff either $i = j$ or $i \neq j$ and i and j are in the same connected component. We can set up in $\Theta(n+e)$ time an array *connec* such that *connec*$[i]$ is the index of the connected component containing vertex i, $1 \leq i \leq n$. Hence, we can determine whether $A^*(i,j)$, $i \neq j$, is 1 or 0 by simply seeing whether *connec*$[i]$ = *connec*$[j]$. The reflexive transitive closure matrix of an undirected graph G with n vertices and e edges can therefore be computed in $\Theta(n^2)$ time and $\Theta(n)$ space using either adjacency lists or matrices (the space count does not include the space needed for A^* itself).

As a final application of breadth first search, consider the problem of obtaining a spanning tree for an undirected graph G. The graph G has a spanning tree iff G is connected. Hence, BFS easily determines the existence of a spanning tree. Furthermore, consider the set of edges (u,w) used in the **for** loop of line 11 of algorithm BFS to reach unvisited vertices w. These edges are called *forward edges*. Let t denote the set of these forward edges. We claim that if G is connected, then t is a spanning tree of G. For the graph of Figure 6.4(a) the set of edges t will be all edges in G except $(5,8)$, $(6,8)$, and $(7,8)$ (see Figure 6.5(b)). Spanning trees obtained using a breadth first search are called *breadth first spanning trees*.



(a) DFS(1) spanning tree (b) BFS(1) spanning tree

Figure 6.5 DFS and BFS spanning trees for the graph of Figure 6.4(a)

Theorem 6.4 Modify algorithm BFS by adding on the statements $t := \emptyset$; and $t := t \cup \{(u, w)\}$; to lines 8 and 16, respectively. Call the resulting algorithm BFS*. If BFS* is called so that v is any vertex in a connected undirected graph G, then on termination, the edges in t form a spanning tree of G.

Proof: We have already seen that if G is a connected graph on n vertices, then all n vertices will get visited. Also, each of these, except the start vertex v, will get onto the queue once (line 15). Hence, t will contain exactly $n-1$ edges. All these edges are distinct. The $n-1$ edges in t will therefore define an undirected graph on n vertices. This graph will be connected since it contains a path from the start vertex v to every other vertex (and so there is a path between each two vertices). A simple proof by induction shows that every connected graph on n vertices with exactly $n-1$ edges is a tree. Hence t is a spanning tree of G. □

As in the case of BFT, the connected components of a graph can be obtained using DFT. Similarly, the reflexive transitive closure matrix of an undirected graph can be found using DFT. If DFS (Algorithm 6.7) is modified by adding $t := \emptyset$; and $t := t \cup \{(v, w)\}$; to line 7 and the **if** statement of line 10, respectively, then when DFS terminates, the edges in t define a spanning tree for the undirected graph G if G is connected. A spanning tree obtained in this manner is called a *depth first spanning tree*. For the graph of Figure 6.4(a) the spanning tree obtained will include all edges in G except for $(2,5)$, $(8,7)$, and $(1,3)$ (see Figure 6.5(a)). Hence, DFS and BFS are equally powerful for the search problems discussed so far.

EXERCISES

1. Show that for any undirected graph $G = (V, E)$, a call to BFS(v) with $v \in V$ results in visiting all the vertices in the connected component containing v.

2. Rewrite BFS and BFT so that all the connected components of the undirected graph G get printed out. Assume that G is input in adjacency list form.

3. Prove that if G is a connected undirected graph with n vertices and $n-1$ edges, then G is a tree.

4. Present a D-search-based algorithm that produces a spanning tree for an undirected connected graph.

5. (a) The *radius* of a tree is its depth. Show that the forward edges used in BFS(v) define a spanning tree with root v having minimum radius among all spanning trees, for the undirected connected graph G having root v.

(b) Using the result of part (a), write an algorithm to find a minimum-radius spanning tree for G. What are the time and space requirements of your algorithm?

6. The *diameter* of a tree is the maximum distance between any two vertices. Let d be the diameter of a minimum-diameter spanning tree for an undirected connected graph G. Let r be the radius of a minimum-radius spanning tree for G.

   (a) Show that $2r - 1 \leq d \leq 2r$.
   (b) Write an algorithm to find a minimum-diameter spanning tree for G. (*Hint*: Use breadth-first search followed by some local modification.)
   (c) Prove that your algorithm is correct.
   (d) What are the time and space requirements of your algorithm?

7. A *bipartite graph* $G = (V, E)$ is an undirected graph whose vertices can be partitioned into two disjoint sets V_1 and $V_2 = V - V_1$ with the properties that no two vertices in V_1 are adjacent in G and no two vertices in V_2 are adjacent in G. The graph G of Figure 6.4(a) is bipartite. A possible partitioning of V is $V_1 = \{1, 4, 5, 6, 7\}$ and $V_2 = \{2, 3, 8\}$. Write an algorithm to determine whether a graph G is bipartite. If G is bipartite, your algorithm should obtain a partitioning of the vertices into two disjoint sets V_1 and V_2 satisfying the properties above. Show that if G is represented by its adjacency lists, then this algorithm can be made to work in time $O(n + e)$, where $n = |V|$ and $e = |E|$.

8. Write an algorithm to find the reflexive transitive closure matrix A^* of a directed graph G. Show that if G has n vertices and e edges and is represented by its adjacency lists, then this can be done in time $\Theta(n^2 + ne)$. How much space does your algorithm take in addition to that needed for G and A^*? (*Hint*: Use either BFS or DFS.)

9. Input is an undirected connected graph $G(V, E)$ each one of whose edges has the same weight w (w being a real number). Give an $O(|E|)$ time algorithm to find a minimum-cost spanning tree for G. What is the weight of this tree?

10. Given are a directed graph $G(V, E)$ and a node $v \in V$. Write an efficient algorithm to decide whether there is a directed path from v to every other node in the graph. What is the worst-case run time of your algorithm?

11. Design an algorithm to decide whether a given undirected graph $G(V, E)$ contains a cycle of length 4. The running time of the algorithm should be $O(|V|^3)$.

12. Let $G(V, E)$ be a binary tree with n nodes. The *distance* between two vertices in G is the length of the path connecting these two vertices. The problem is to construct an $n \times n$ matrix whose ijth entry is the distance between v_i and v_j. Design an $O(n^2)$ time algorithm to construct such a matrix. Assume that the tree is given in the adjacency-list representation.

13. Present an $O(|V|)$ time algorithm to check whether a given undirected graph $G(V, E)$ is a tree. The graph G is given in the form of an adjacency list.

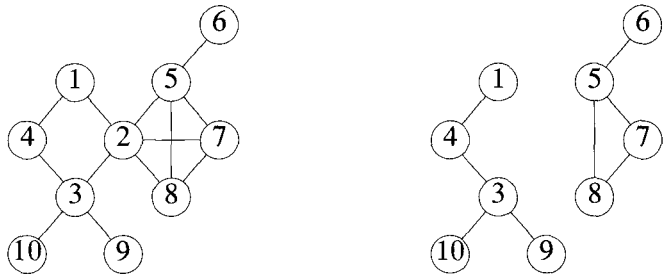
6.4 BICONNECTED COMPONENTS AND DFS

In this section, by "graph" we always mean an undirected graph. A vertex v in a connected graph G is an *articulation point* if and only if the deletion of vertex v together with all edges incident to v disconnects the graph into two or more nonempty components.

Example 6.3 In the connected graph of Figure 6.6(a) vertex 2 is an articulation point as the deletion of vertex 2 and edges $(1,2), (2,3), (2,5), (2,7)$, and $(2,8)$ leaves behind two disconnected nonempty components (Figure 6.6(b)). Graph G of Figure 6.6(a) has only two other articulation points: vertex 5 and vertex 3. Note that if any of the remaining vertices is deleted from G, then exactly one component remains. □

A graph G is *biconnected* if and only if it contains no articulation points. The graph of Figure 6.6(a) is not biconnected. The graph of Figure 6.7 is biconnected. The presence of articulation points in a connected graph can be an undesirable feature in many cases. For example, if G represents a communication network with the vertices representing communication stations and the edges communication lines, then the failure of a communication station i that is an articulation point would result in the loss of communication to points other than i too. On the other hand, if G has no articulation point, then if any station i fails, we can still communicate between every two stations not including station i.

In this section we develop an efficient algorithm to test whether a connected graph is biconnected. For the case of graphs that are not biconnected, this algorithm will identify all the articulation points. Once it has been determined that a connected graph G is not biconnected, it may be desirable to determine a set of edges whose inclusion makes the graph biconnected. Determining such a set of edges is facilitated if we know the maximal subgraphs of G that are biconnected. $G' = (V', E')$ is a maximal biconnected subgraph of G if and only if G has no biconnected subgraph $G'' = (V'', E'')$



(a) Graph G (b) Result of deleting vertex 2

Figure 6.6 An example graph

such that $V' \subseteq V''$ and $E' \subset E''$. A maximal biconnected subgraph is a *biconnected component.*

The graph of Figure 6.7 has only one biconnected component (i.e., the entire graph). The biconnected components of the graph of Figure 6.6(a) are shown in Figure 6.8.

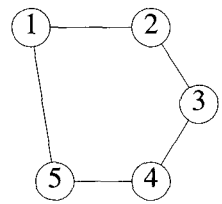


Figure 6.7 A biconnected graph

It is relatively easy to show that

Lemma 6.1 Two biconnected components can have at most one vertex in common and this vertex is an articulation point. □

Hence, no edge can be in two different biconnected components (as this would require two common vertices). The graph G can be transformed into a biconnected graph by using the edge addition scheme of Algorithm 6.8.

Since every biconnected component of a connected graph G contains at least two vertices (unless G itself has only one vertex), it follows that the v_i of line 5 exists.

Example 6.4 Using the above scheme to transform the graph of Figure 6.6(a) into a biconnected graph requires us to add edges $(4, 10)$ and $(10, 9)$ (corresponding to the articulation point 3), edge $(1, 5)$ (corresponding to the articulation point 2), and edge $(6, 7)$ (corresponding to point 5). □

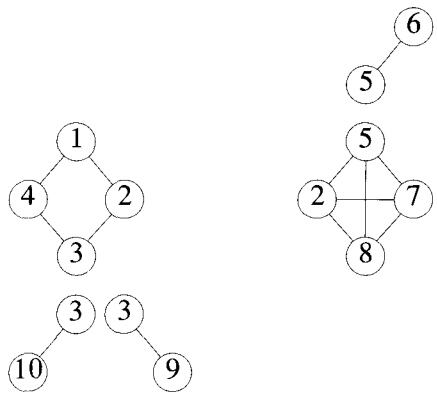


Figure 6.8 Biconnected components of graph of Figure 6.6(a)

Note that once the edges (v_i, v_{i+1}) of line 6 (Algorithm 6.8) are added, vertex a is no longer an articulation point. Hence following the addition

```
1  for each articulation point a do
2  {
3      Let B_1, B_2, ..., B_k be the biconnected
4      components containing vertex a;
5      Let v_i, v_i ≠ a, be a vertex in B_i, 1 ≤ i ≤ k;
6      Add to G the edges (v_i, v_{i+1}), 1 ≤ i < k;
7  }
```

Algorithm 6.8 Scheme to construct a biconnected graph

of the edges corresponding to all articulation points, G has no articulation points and so is biconnected. If G has p articulation points and b biconnected components, then the scheme of Algorithm 6.8 introduces exactly $b - p$ new edges into G. One can show that this scheme may use more than the minimum number of edges needed to make G biconnected (see the exercises).

Now, let us attack the problem of identifying the articulation points and biconnected components of a connected graph G with $n \geq 2$ vertices. The problem is efficiently solved by considering a depth first spanning tree of G.

Figure 6.9(a) and (b) shows a depth first spanning tree of the graph of Figure 6.6(a). In each figure there is a number outside each vertex. These numbers correspond to the order in which a depth first search visits these vertices and are referred to as the *depth first numbers (dfns)* of the vertex. Thus, $dfn[1] = 1$, $dfn[4] = 2$, $dfn[6] = 8$, and so on. In Figure 6.9(b) solid edges form the depth first spanning tree. These edges are called *tree edges.* Broken edges (i.e., all the remaining edges) are called *back edges.*

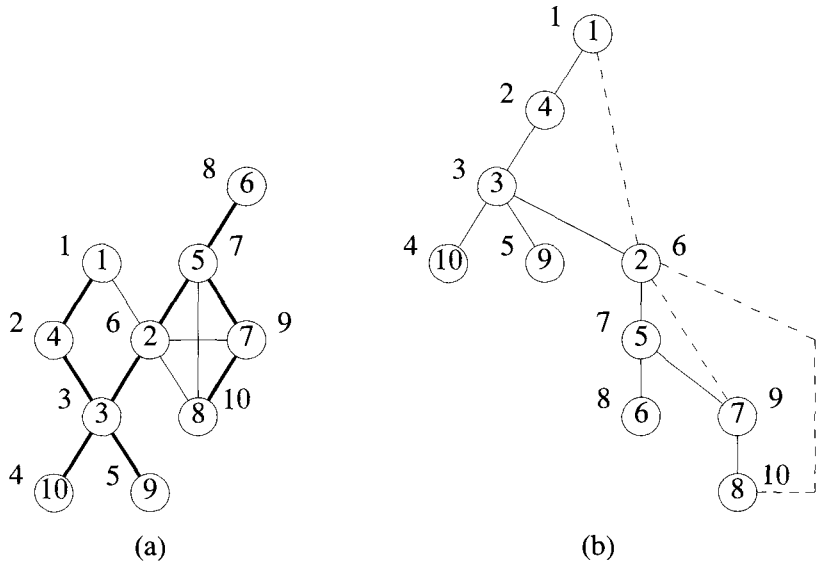


Figure 6.9 A depth first spanning tree of the graph of Figure 6.6(a)

Depth first spanning trees have a property that is very useful in identifying articulation points and biconnected components

Lemma 6.2 If (u, v) is any edge in G, then relative to the depth first spanning tree t, either u is an ancestor of v or v is an ancestor of u. So, there are no cross edges relative to a depth first spanning tree ((u, v) is a *cross* edge relative to t if and only if u is not an ancestor of v and v is not an ancestor of u).

Proof: To see this, assume that $(u, v) \in E(G)$ and (u, v) is a cross edge. Then (u, v) cannot be a tree edge as otherwise u is the parent of v or vice versa. So, (u, v) must be a back edge. Without loss of generality, we can assume $dfn[u] < dfn[v]$. Since vertex u is visited first, its exploration cannot be complete until vertex v is visited. From the definition of depth first search, it follows that u is an ancestor of all the vertices visited until u is completely explored. Hence u is an ancestor of v in t and (u, v) cannot be a cross edge. □

We make the following observation

Lemma 6.3 The root node of a depth first spanning tree is an articulation point iff it has at least two children. Furthermore, if u is any other vertex, then it is not an articulation point iff from every child w of u it is possible to reach an ancestor of u using only a path made up of descendents of w and a back edge. □

Note that if this cannot be done for some child w of u, then the deletion of vertex u leaves behind at least two nonempty components (one containing the root and the other containing vertex w). This observation leads to a simple rule to identify articulation points. For each vertex u, define $L[u]$ as follows:

$$L[u] = \min \{dfn[u], \min \{L[w] \mid w \text{ is a child of } u\}, \min \{dfn[w] \mid (u, w) \text{ is a back edge}\}\}$$

It should be clear that $L[u]$ is the lowest depth first number that can be reached from u using a path of descendents followed by at most one back edge. From the preceding discussion it follows that if u is not the root, then u is an articulation point iff u has a child w such that $L[w] \geq dfn[u]$.

Example 6.5 For the spanning tree of Figure 6.9(b) the L values are $L[1 : 10] = \{1, 1, 1, 1, 6, 8, 6, 6, 5, 4\}$. Vertex 3 is an articulation point as child 10 has $L[10] = 4$ and $dfn[3] = 3$. Vertex 2 is an articulation point as child 5 has $L[5] = 6$ and $dfn[2] = 6$. The only other articulation point is vertex 5; child 6 has $L[6] = 8$ and $dfn[5] = 7$. □

$L[u]$ can be easily computed if the vertices of the depth first spanning tree are visited in postorder. Thus, to determine the articulation points,

it is necessary to perform a depth first search of the graph G and visit the nodes in the resulting depth first spanning tree in postorder. It is possible to do both these functions in parallel. Pseudocode Art (Algorithm 6.9) carries out a depth first search of G. During this search each newly visited vertex gets assigned its depth first number. At the same time, $L[i]$ is computed for each vertex in the tree. This algorithm assumes that the connected graph G and the arrays dfn and L are global. In addition, it is assumed that the variable num is also global. It is clear from the algorithm that when vertex u has been explored and a return made from the function, then $L[u]$ has been correctly computed. Note that in the **else** clause of line 15, if $w \neq v$, then either (u, w) is a back edge or $dfn[w] > dfn[u] \geq L[u]$. In either case, $L[u]$ is correctly updated. The initial call to Art is Art$(1, 0)$. Note dfn is initialized to zero before invoking Art.

```
1   Algorithm Art(u, v)
2   // u is a start vertex for depth first search. v is its parent if any
3   // in the depth first spanning tree. It is assumed that the global
4   // array dfn is initialized to zero and that the global variable
5   // num is initialized to 1. n is the number of vertices in G.
6   {
7       dfn[u] := num; L[u] := num; num := num + 1;
8       for each vertex w adjacent from u do
9       {
10          if (dfn[w] = 0) then
11          {
12              Art(w, u); // w is unvisited.
13              L[u] := min(L[u], L[w]);
14          }
15          else if (w ≠ v) then L[u] := min(L[u], dfn[w]);
16      }
17  }
```

Algorithm 6.9 Pseudocode to compute dfn and L

Once $L[1:n]$ has been computed, the articulation points can be identified in $O(e)$ time. Since Art has a complexity $O(n+e)$, where e is the number of edges in G, the articulation points of G can be determined in $O(n+e)$ time.

Now, what needs to be done to determine the biconnected components of G? If following the call to Art (line 12) $L[w] \geq dfn[u]$, then we know that u is either the root or an articulation point. Regardless of whether u is not the root or is the root and has one or more children, the edge (u, w) together with

all edges (both tree and back) encountered during this call to Art (except for edges in other biconnected components contained in subtree w) forms a biconnected component. A formal proof of this statement appears in the proof of Theorem 6.5. The modified algorithm appears as Algorithm 6.10.

```
1     Algorithm BiComp(u, v)
2     // u is a start vertex for depth first search. v is its parent if
3     // any in the depth first spanning tree. It is assumed that the
4     // global array dfn is initially zero and that the global variable
5     // num is initialized to 1. n is the number of vertices in G.
6     {
7         dfn[u] := num; L[u] := num; num := num + 1;
8         for each vertex w adjacent from u do
9         {
9.1           if ((v ≠ w) and (dfn[w] < dfn[u]))then
9.2               add (u, w) to the top of a stack s;
10            if (dfn[w] = 0) then
11            {
11.1              if (L[w] ≥ dfn[u]) then
11.2              {
11.3                  write ("New bicomponent");
11.4                  repeat
11.5                  {
11.6                      Delete an edge from the top of stack s;
11.7                      Let this edge be (x, y);
11.8                      write (x, y);
11.9                  } until (((x, y) = (u, w)) or ((x, y) = (w, u)));
11.10             }
12                BiComp(w, u); // w is unvisited.
13                L[u] := min(L[u], L[w]);
14            }
15            else if (w ≠ v) then L[u] := min(L[u], dfn[w]);
16        }
17    }
```

Algorithm 6.10 Pseudocode to determine bicomponents

One can verify that the computing time of Algorithm 6.10 remains $O(n+e)$. The following theorem establishes the correctness of the algorithm. Note that when G has only one vertex, it has no edges so the algorithm generates

no output. In this case G does have a biconnected component, namely its single vertex. This case can be handled separately.

Theorem 6.5 Algorithm 6.10 correctly generates the biconnected components of the connected graph G when G has at least two vertices.

Proof: This can be shown by induction on the number of biconnected components in G. Clearly, for all biconnected graphs G, the root u of the depth first spanning tree has only one child w. Furthermore, w is the only vertex for which $L[w] \geq dfn[u]$ in line 11.1 of Algorithm 6.10. By the time w has been explored, all edges in G have been output as one biconnected component.

Now assume the algorithm works correctly for all connected graphs G with at most m biconnected components. We show that it also works correctly for all connected graphs with $m+1$ biconnected components. Let G be any such graph. Consider the first time that $L[w] \geq dfn[u]$ in line 11.1. At this time no edges have been output and so all edges in G incident to the descendents of w are on the stack and are above the edge (u, w). Since none of the descendents of u is an articulation point and u is, it follows that the set of edges above (u, w) on the stack forms a biconnected component together with the edge (u, w). Once these edges have been deleted from the stack and output, the algorithm behaves essentially as it would on the graph G', obtained by deleting from G the biconnected component just output. The behavior of the algorithm on G differs from that on G' only in that during the completion of the exploration of vertex u, some edges (u, r) such that (u, r) is in the component just output may be considered. However, for all such edges, $dfn[r] \neq 0$ and $dfn[r] > dfn[u] \geq L[u]$. Hence, these edges only result in a vacuous iteration of the **for** loop of line 8 and do not materially affect the algorithm.

One can easily establish that G' has at least two vertices. Since in addition G' has exactly m biconnected components, it follows from the induction hypothesis that the remaining components are correctly generated. □

It should be noted that the algorithm described above will work with any spanning tree relative to which the given graph has no cross edges. Unfortunately, graphs can have cross edges relative to breadth first spanning trees. Hence, algorithm Art cannot be adapted to BFS.

EXERCISES

1. For the graphs of Figure 6.10 identify the articulation points and draw the biconnected components.

2. Show that if G is a connected undirected graph, then no edge of G can be in two different biconnected components.

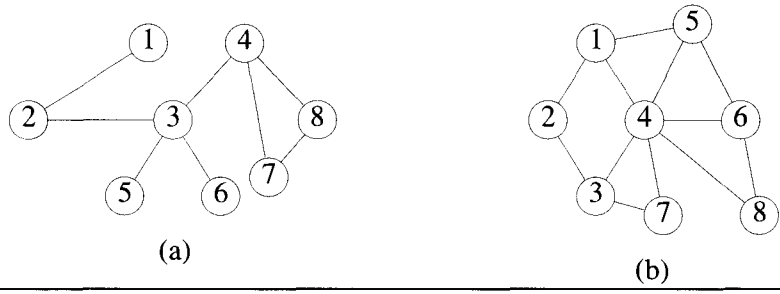


Figure 6.10 Graphs for Exercise 1

3. Let $G_i = (V_i, E_i), 1 \leq i \leq k$, be the biconnected components of a connected graph G. Show that

   (a) If $i \neq j$, then $V_i \cap V_j$ contains at most one vertex.

   (b) Vertex v is an articulation point of G iff $\{v\} = V_i \cap V_j$ for some i and j, $i \neq j$.

4. Show that the scheme of Algorithm 6.8 may use more than the minimum number of edges needed to make G biconnected.

5. Let G be a connected undirected graph. Write an algorithm to find the minimum number of edges that have to be added to G so that G becomes biconnected. Your algorithm should output such a set of edges. What are the time and space requirements of your algorithm?

6. Show that if t is a breadth first spanning tree for an undirected connected graph G, then G may have cross edges relative to t.

7. Prove that a nonroot vertex u is an articulation point iff $L[w] \geq dfn[u]$ for some child w of u.

8. Prove that in BiComp (Algorithm 6.10) if either $v = w$ or $dfn[w] > dfn[u]$, then edge (u, w) is either already on the stack of edges or has been output as part of a biconnected component.

9. Let $G(V, E)$ be any connected undirected graph. A *bridge* of G is defined to be an edge of G which when removed from G, will make it disconnected. Present an $O(|E|)$ time algorithm to find all the bridges of G.

10. Let $S(V, T)$ be any DFS tree for a given connected undirected graph $G(V, E)$. Prove that a leaf of S can not be an articulation point of G.

11. Prove or disprove: "An undirected graph $G(V, E)$ is biconnected if and only if for each pair of distinct vertices v and w in V there are two distinct paths from v to w that have no vertices in common except v and w."

6.5 REFERENCES AND READINGS

Several applications of depth first search to graph problems are given in "Depth first search and linear graph algorithms," by R. Tarjan, *SIAM Journal on Computing* 1, no. 2 (1972): 146–160.

The $O(n + e)$ depth first algorithm for biconnected components is due to R. Tarjan and appears in the preceding paper. This paper also contains an $O(n + e)$ algorithm to find the strongly connected components of a directed graph.

An $O(n + e)$ algorithm to find a smallest set of edges that, when added to a graph G, produces a biconnected graph has been given by A. Rosenthal and A. Goldner.

For an extensive coverage on graph algorithms see:

Data Structures and Network Algorithms, by R. E. Tarjan, Society for Industrial and Applied Mathematics, 1983.

Algorithmic Graph Theory, by A. Gibbons, Cambridge University Press, 1985.

Algorithmic Graph Theory and Perfect Graphs, by M. Golumbic, Academic Press, 1980.

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). □

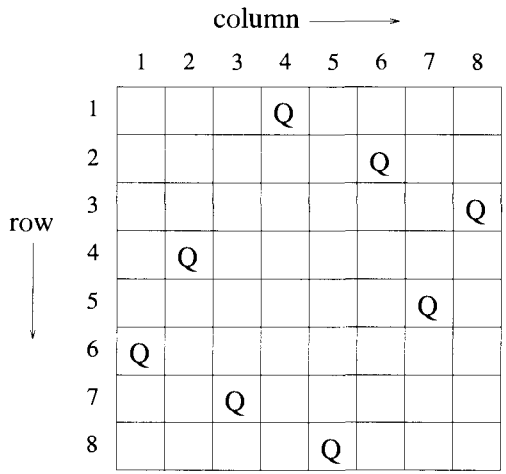


Figure 7.1 One solution to the 8-queens problem

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. □

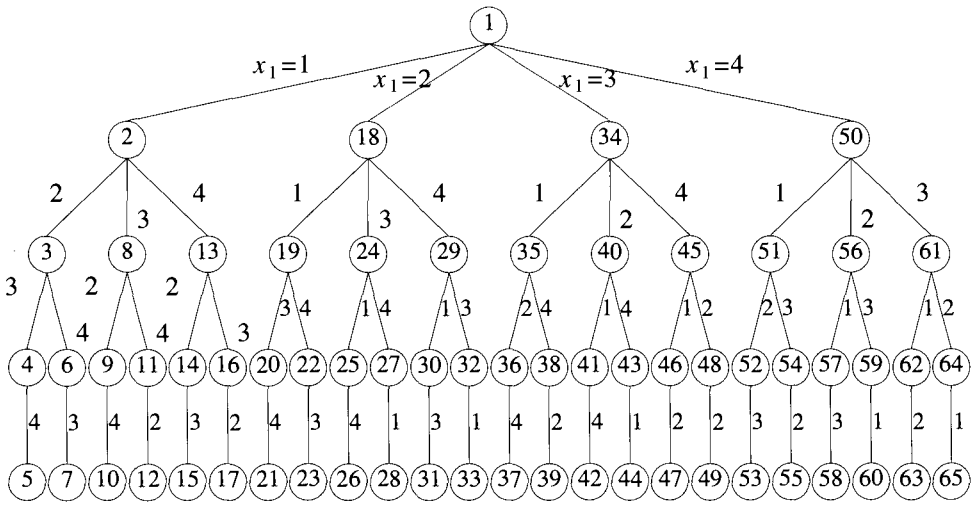


Figure 7.2 Tree organization of the 4-queens solution space. Nodes are numbered as in depth first search.

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. □

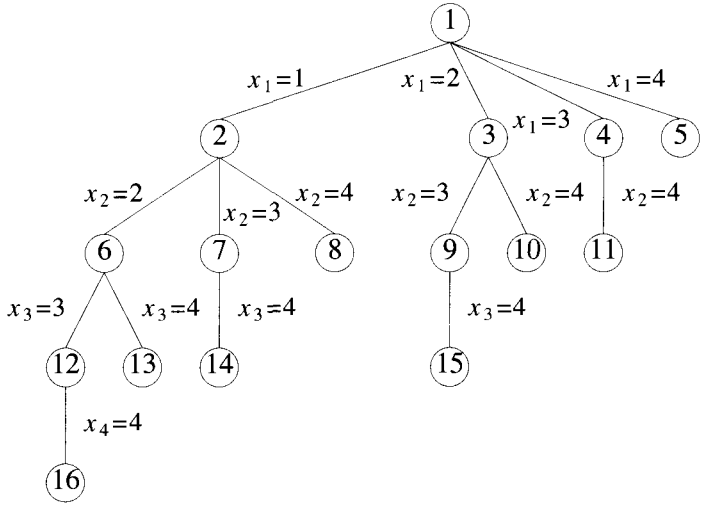


Figure 7.3 A possible solution space organization for the sum of subsets problem. Nodes are numbered as in breadth-first search.

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*.

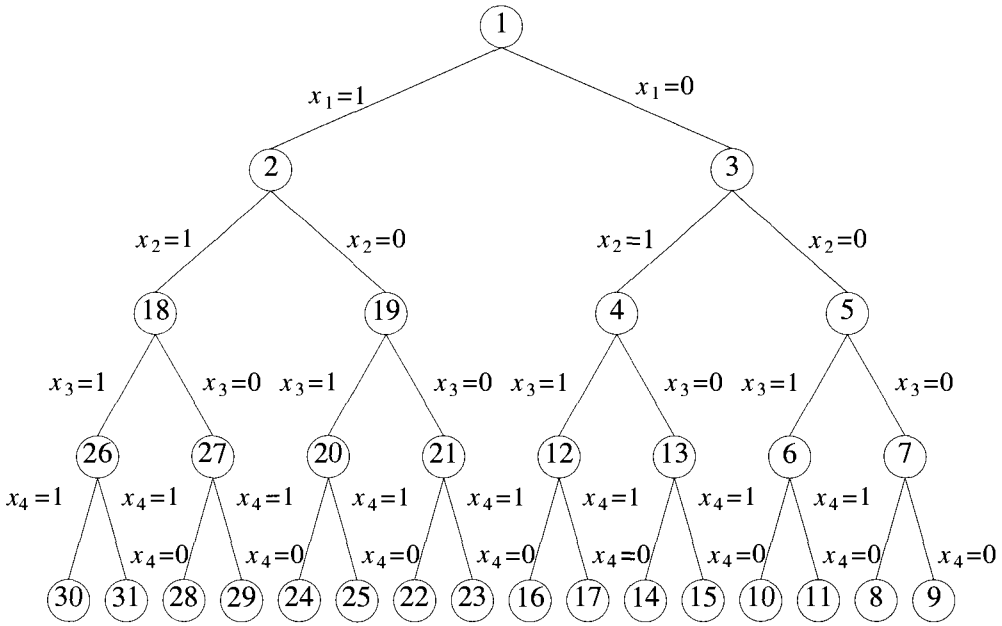


Figure 7.4 Another possible organization for the sum of subsets problems. Nodes are numbered as in D-search.

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. $T(x_1, x_2, \ldots, x_n) = \emptyset$.

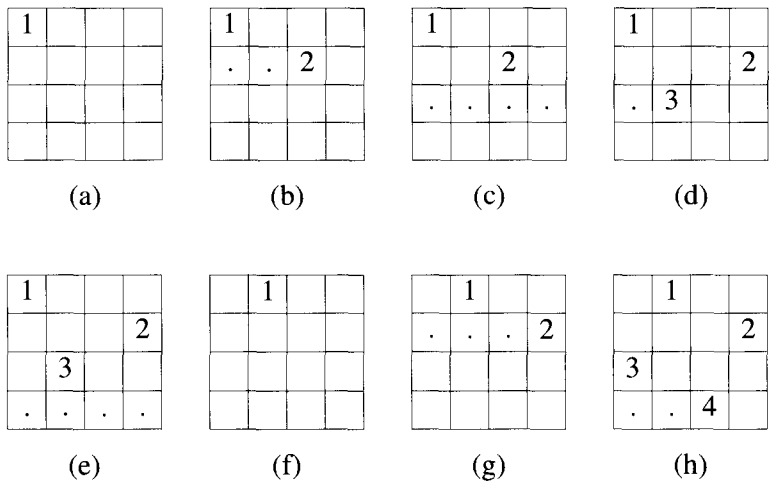


Figure 7.5 Example of a backtrack solution to the 4-queens problem

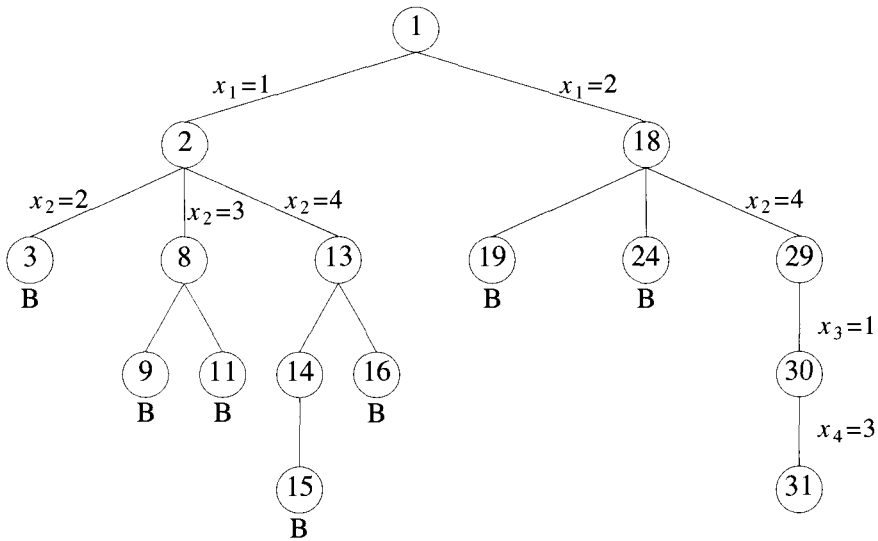


Figure 7.6 Portion of the tree of Figure 7.2 that is generated during backtracking

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

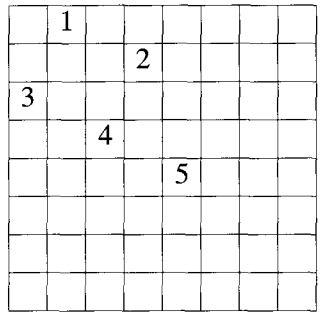
1. Algorithm NQueens can be made more efficient by redefining the function Place(k, i) so that it either returns the next legitimate column on which to place the kth queen or an illegal value. Rewrite both functions (Algorithms 7.4 and 7.5) so they implement this alternate strategy.

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.

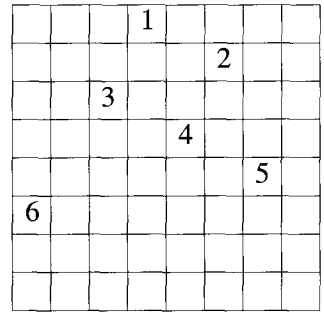
   Observe that for finding inequivalent solutions the algorithm need only set $x[1] = 2, 3, \ldots, \lceil n/2 \rceil$.

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

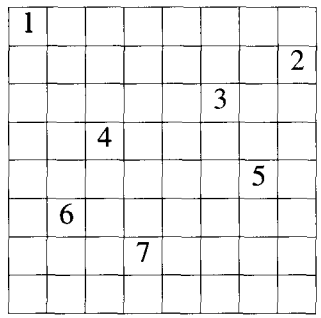
   (b) Run the n-queens program devised above for $n = 8$, 9, and 10. Tabulate the number of solutions your program finds for each value of n.



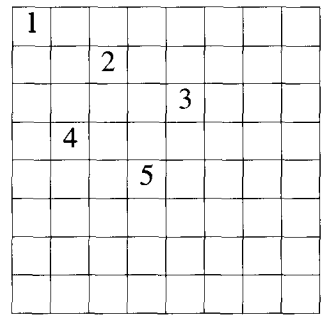
(8,5,4,3,2) = 1649



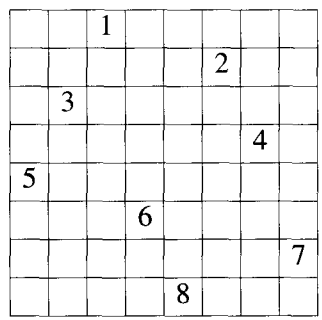
(8,5,3,1,2,1) = 769



(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

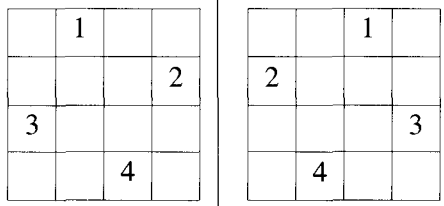


Figure 7.9 Equivalent solutions to the 4-queens problem

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.

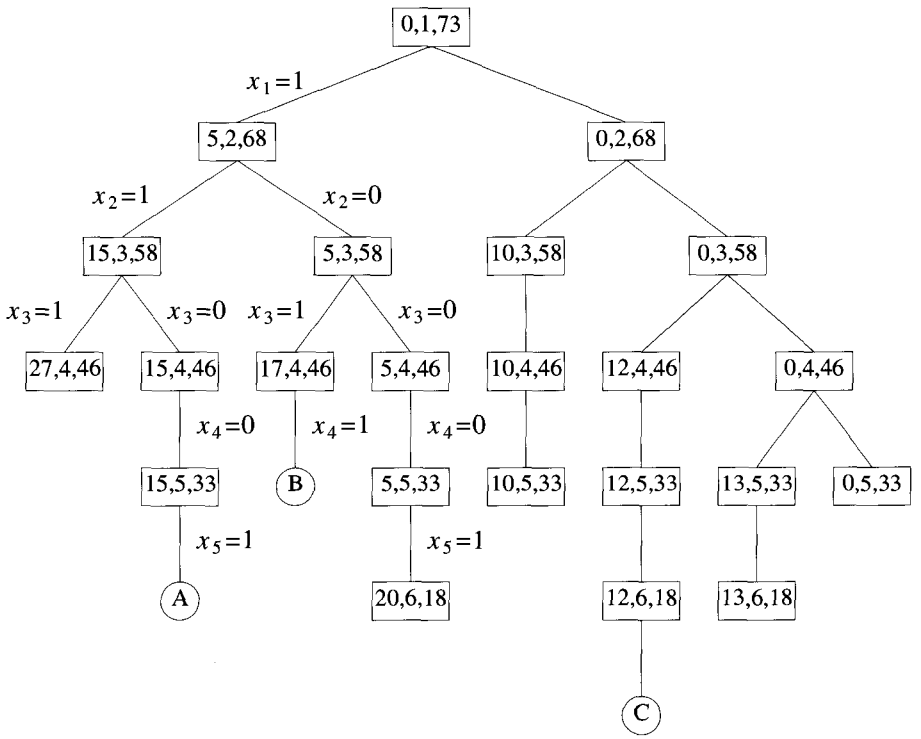


Figure 7.10 Portion of state space tree generated by SumOfSub

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.

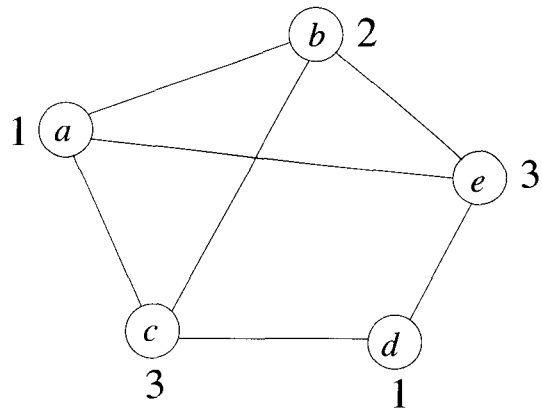


Figure 7.11 An example graph and its coloring

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

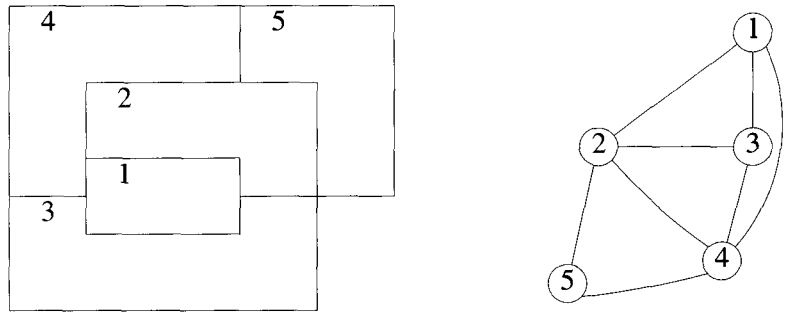


Figure 7.12 A map and its planar graph representation

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

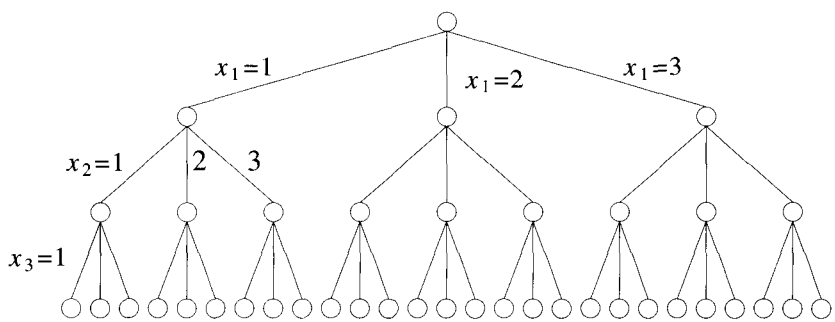


Figure 7.13 State space tree for mColoring when n =3 and m = 3

```
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

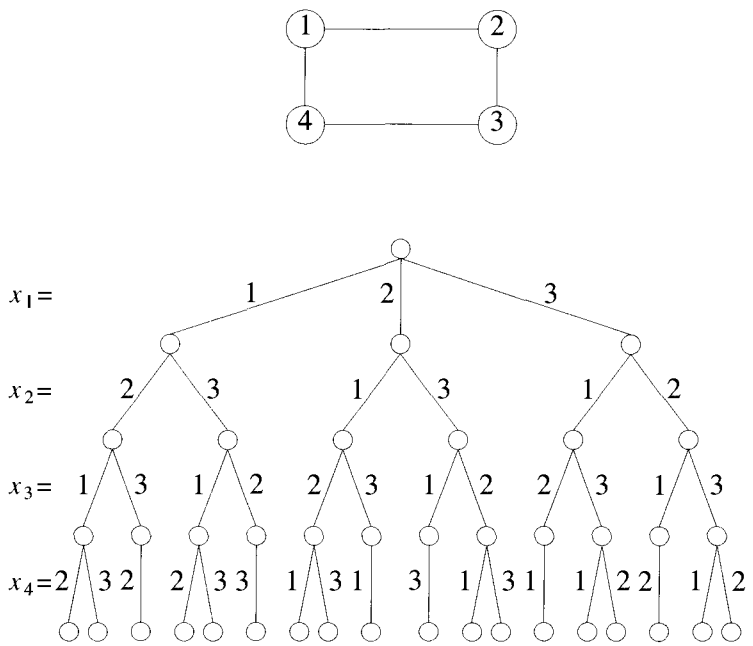


Figure 7.14 A 4-node graph and all possible 3-colorings

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

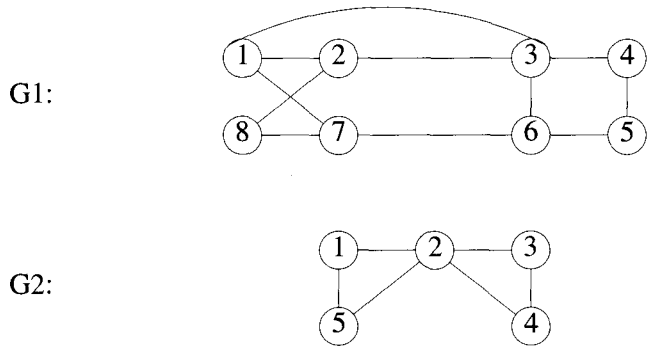


Figure 7.15 Two graphs, one containing a Hamiltonian cycle

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).

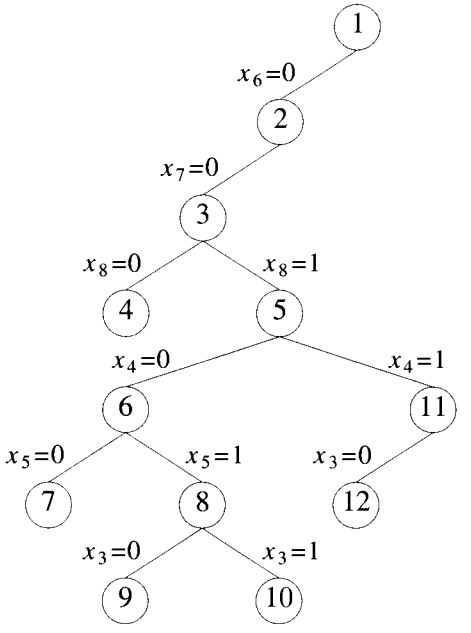


Figure 7.16 Part of the dynamic state space tree generated in Example 7.7

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.

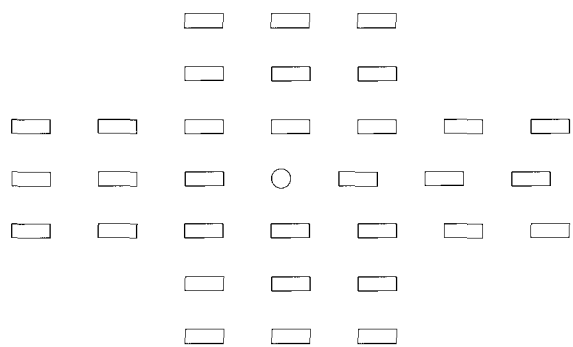


Figure 7.17 A Hi-Q board in its initial state

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

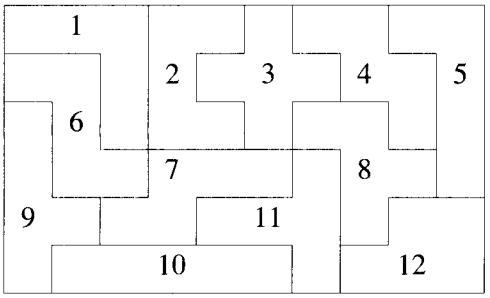


Figure 7.18 A pentomino configuration

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.

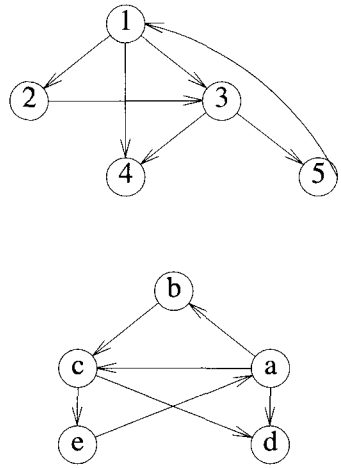


Figure 7.19 Two isomorphic graphs (Exercise 10)

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. □

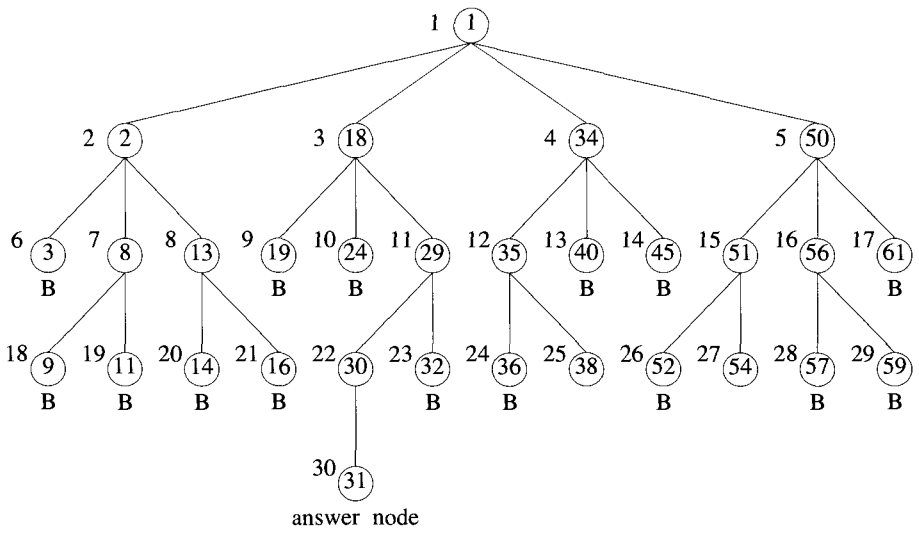


Figure 8.1 Portion of 4-queens state space tree generated by FIFO branch-and-bound

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.

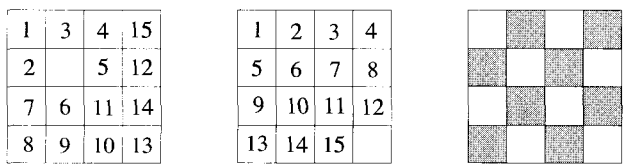
A search strategy that uses a cost function $\hat{c}(x) = f(h(x)) + \hat{g}(x)$ to select the next E-node would always choose for its next E-node a live node with least $\hat{c}(\cdot)$. Hence, such a search strategy is called an LC-search (**L**east **C**ost search). It is interesting to note that BFS 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.

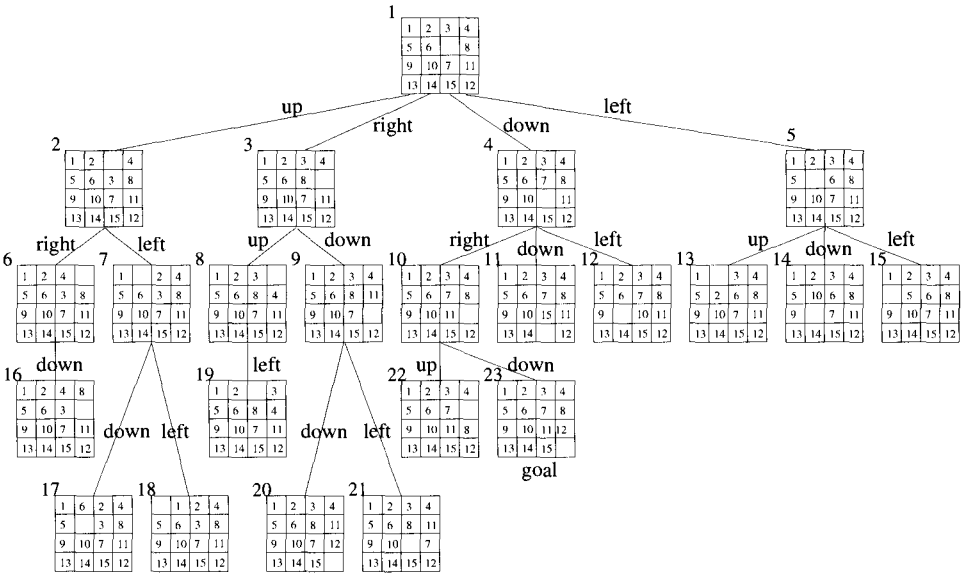


Figure 8.3 Part of the state space tree for the 15-puzzle

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.

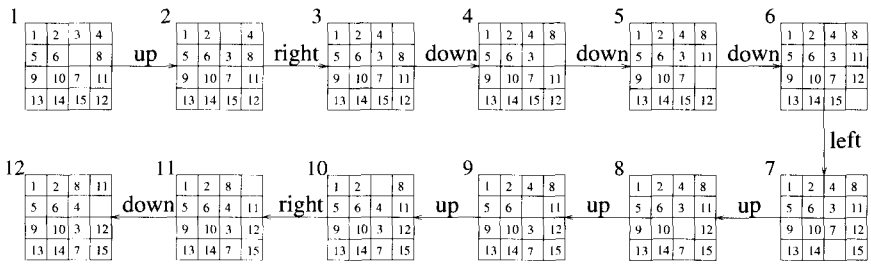


Figure 8.4 First ten steps in a depth first search

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