Chapter 8

Dynamic Programming

In the previous chapter we saw that it is often possible to divide an instance into subinstances, to solve the subinstances (perhaps by dividing them further), and then to combine the solutions of the subinstances so as to solve the original instance. It sometimes happens that the natural way of dividing an instance suggested by the structure of the problem leads us to consider several overlapping subinstances. If we solve each of these independently, they will in turn create a host of identical subinstances. If we pay no attention to this duplication, we are likely to end up with an inefficient algorithm; if, on the other hand, we take advantage of the duplication and arrange to solve each subinstance only once, saving the solution for later use, then a more efficient algorithm will result. The underlying idea of dynamic programming is thus quite simple: avoid calculating the same thing twice, usually by keeping a table of known results that fills up as subinstances are solved.

Divide-and-conquer is a *top-down* method. When a problem is solved by divide-and-conquer, we immediately attack the complete instance, which we then divide into smaller and smaller subinstances as the algorithm progresses. Dynamic programming on the other hand is a *bottom-up* technique. We usually start with the smallest, and hence the simplest, subinstances. By combining their solutions, we obtain the answers to subinstances of increasing size, until finally we arrive at the solution of the original instance.

We begin the chapter with two simple examples of dynamic programming that illustrate the general technique in an uncomplicated setting. The following sections pick up the problems of making change, which we met in Section 6.1, and of filling a knapsack, encountered in Section 6.5.

8.1 Two simple examples

8.1.1 Calculating the binomial coefficient

Consider the problem of calculating the binomial coefficient

$$\binom{n}{k} = \begin{cases} 1 & \text{if k = 0 or k = n} \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if 0 < k < n} \\ 0 & \text{otherwise.} \end{cases}$$

Suppose $0 \leq k \leq n$. If we calculate $\binom{n}{k}$ directly by

function $C(n, k)$
  if $k = 0$ **or** $k = n$ **then return** 1
  else return $C(n-1, k-1) + C(n-1, k)$

many of the values $C(i, j)$, $i < n$, $j < k$, are calculated over and over. For example, the algorithm calculates $C(5,3)$ as the sum of $C(4,2)$ and $C(4,3)$. Both these intermediate results require us to calculate $C(3,2)$. Similarly the value of $C(2,2)$ is used several times. Since the final result is obtained by adding up a number of 1s, the execution time of this algorithm is sure to be in $\Omega\left(\binom{n}{k}\right)$. We met a similar phenomenon before in the algorithm *Fibrec* for calculating the Fibonacci sequence; see Section 2.7.5.

If, on the other hand, we use a table of intermediate results—this is of course *Pascal's triangle*—we obtain a more efficient algorithm; see Figure 8.1. The table should be filled line by line. In fact, it is not even necessary to store the entire table: it suffices to keep a vector of length k, representing the current line, and to update this vector from left to right. Thus to calculate $\binom{n}{k}$ the algorithm takes a time in $\Theta(nk)$ and space in $\Theta(k)$, if we assume that addition is an elementary operation.

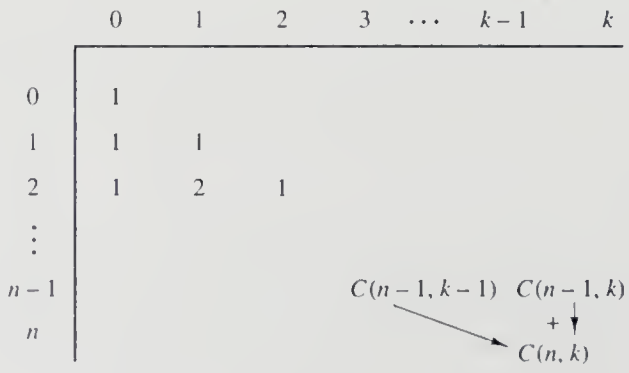

Figure 8.1. Pascal's triangle

8.1.2 The World Series

Imagine a competition in which two teams A and B play not more than $2n-1$ games, the winner being the first team to achieve n victories. We assume that there are no tied games, that the results of each match are independent, and that for any given match there is a constant probability p that team A will be the winner, and hence a constant probability $q = 1 - p$ that team B will win.

Let $P(i, j)$ be the probability that team A will win the series given that they still need i more victories to achieve this, whereas team B still need j more victories if they are to win. For example, before the first game of the series the probability that team A will be the overall winner is $P(n, n)$: both teams still need n victories to win the series. If team A require 0 more victories, then in fact they have already won the series, and so $P(0, i) = 1$, $1 \le i \le n$. Similarly if team B require 0 more victories, then they have already won the series, and the probability that team A will be the overall winners is zero: so $P(i, 0) = 0$, $1 \le i \le n$. Since there cannot be a situation where both teams have won all the matches they need, $P(0, 0)$ is meaningless. Finally, since team A win any given match with probability p and lose it with probability q,

$$P(i, j) = pP(i-1, j) + qP(i, j-1), \quad i \ge 1, \quad j \ge 1.$$

Thus we can compute $P(i, j)$ as follows.

```
function P(i, j)
    if i = 0 then return 1
    else if j = 0 then return 0
    else return pP(i − 1, j)+qP(i, j − 1)
```

Let $T(k)$ be the time needed in the worst case to calculate $P(i, j)$, where $k = i + j$. With this method, we see that

$$\begin{aligned} T(1) &= c \\ T(k) &\le 2T(k-1) + d, \quad k > 1 \end{aligned}$$

where c and d are constants. Rewriting $T(k-1)$ in terms of $T(k-2)$, and so on, we find

$$\begin{aligned} T(k) &\le 4T(k-2) + 2d + d, \quad k > 2 \\ &\vdots \\ &\le 2^{k-1}T(1) + (2^{k-2} + 2^{k-3} + \cdots + 2 + 1)d \\ &= 2^{k-1}c + (2^{k-1} - 1)d \\ &= 2^k(c/2 + d/2) - d. \end{aligned}$$

$T(k)$ is therefore in $O(2^k)$, which is $O(4^n)$ if $i = j = n$. In fact, if we look at the way the recursive calls are generated, we find the pattern shown in Figure 8.2, which is identical to that obtained in the naive calculation of the binomial coefficient. To see this, imagine that any call $P(m, n)$ in the figure is replaced by $C(m + n, n)$.

Thus $P(i, j)$ is replaced by $C(i + j, j)$, $P(i - 1, j)$ by $C(i + j - 1, j)$, and $P(i, j - 1)$ by $C(i + j - 1, j - 1)$. Now the pattern of calls shown by the arrows corresponds to the calculation

$$C(i + j, j) = C(i + j - 1, j) + C(i + j - 1, j - 1)$$

of a binomial coefficient. The total number of recursive calls is therefore exactly $2\binom{i+j}{j} - 2$; see Problem 8.1. To calculate the probability $P(n, n)$ that team A will win given that the series has not yet started, the required time is thus in $\Omega\left(\binom{2n}{n}\right)$.

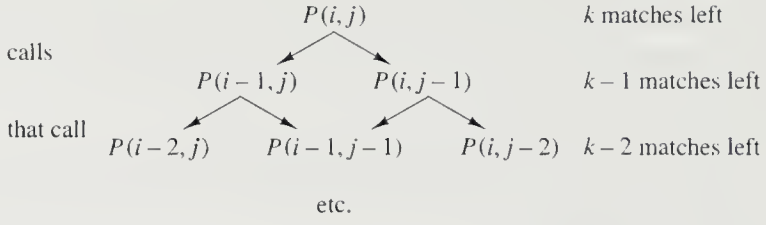

Figure 8.2. Recursive calls made by a call on $P(i, j)$

Problem 1.42 asks the reader to show that $\binom{2n}{n} \geq 4^n/(2n + 1)$. Combining these results, we see that the time required to calculate $P(n, n)$ is in $O(4^n)$ and in $\Omega(4^n/n)$. The method is therefore not practical for large values of n. (Although sporting competitions with $n > 4$ are the exception, this problem does have other applications!)

To speed up the algorithm, we proceed more or less as with Pascal's triangle: we declare an array of the appropriate size and then fill in the entries. This time, however, instead of filling the array line by line, we work diagonal by diagonal. Here is the algorithm to calculate $P(n, n)$.

```
function series(n, p)
    array P[0..n, 0..n]
    q ← 1 − p
    {Fill from top left to main diagonal}
    for s ← 1 to n do
        P[0, s] ← 1;  P[s, 0] ← 0
        for k ← 1 to s − 1 do
            P[k, s − k] ← pP[k − 1, s − k] + qP[k, s − k − 1]
    {Fill from below main diagonal to bottom right}
    for s ← 1 to n do
        for k ← 0 to n − s do
            P[s + k, n − k] ← pP[s + k − 1, n − k] + qP[s + k, n − k − 1]
    return P[n, n]
```

Since the algorithm has to fill an $n \times n$ array, and since a constant time is required to calculate each entry, its execution time is in $\Theta(n^2)$. As with Pascal's triangle, it is easy to implement this algorithm so that storage space in $\Theta(n)$ is sufficient.

8.2 Making change (2)

Recall that the problem is to devise an algorithm for paying a given amount to a customer using the smallest possible number of coins. In Section 6.1 we described a greedy algorithm for this problem. Unfortunately, although the greedy algorithm is very efficient, it works only in a limited number of instances. With certain systems of coinage, or when coins of a particular denomination are missing or in short supply, the algorithm may either find a suboptimal answer, or not find an answer at all.

For example, suppose we live where there are coins for 1, 4 and 6 units. If we have to make change for 8 units, the greedy algorithm will propose doing so using one 6-unit coin and two 1-unit coins, for a total of three coins. However it is clearly possible to do better than this: we can give the customer his change using just two 4-unit coins. Although the greedy algorithm does not find this solution, it is easily obtained using dynamic programming.

As in the previous section, the crux of the method is to set up a table containing useful intermediate results that are then combined into the solution of the instance under consideration. Suppose the currency we are using has available coins of n different denominations. Let a coin of denomination i, $1 \leq i \leq n$, have value d_i units. We suppose, as is usual, that each $d_i > 0$. For the time being we shall also suppose that we have an unlimited supply of coins of each denomination. Finally suppose we have to give the customer coins worth N units, using as few coins as possible.

To solve this problem by dynamic programming, we set up a table $c[1..n, 0..N]$, with one row for each available denomination and one column for each amount from 0 units to N units. In this table $c[i, j]$ will be the minimum number of coins required to pay an amount of j units, $0 \leq j \leq N$, using only coins of denominations 1 to i, $1 \leq i \leq n$. The solution to the instance is therefore given by $c[n, N]$ if all we want to know is how many coins are needed. To fill in the table, note first that $c[i, 0]$ is zero for every value of i. After this initialization, the table can be filled either row by row from left to right, or column by column from top to bottom. To pay an amount j using coins of denominations 1 to i, we have in general two choices. First, we may choose not to use any coins of denomination i, even though this is now permitted, in which case $c[i, j] = c[i-1, j]$. Alternatively, we may choose to use at least one coin of denomination i. In this case, once we have handed over the first coin of this denomination, there remains to be paid an amount of $j - d_i$ units. To pay this takes $c[i, j - d_i]$ coins, so $c[i, j] = 1 + c[i, j - d_i]$. Since we want to minimize the number of coins used, we choose whichever alternative is the better. In general therefore

$$c[i, j] = \min(c[i-1, j], 1 + c[i, j - d_i]).$$

When $i = 1$ one of the elements to be compared falls outside the table. The same is true when $j < d_1$. It is convenient to think of such elements as having the value $+\infty$. If $i = 1$ and $j < d_1$, then both elements to be compared fall outside the table. In this case we set $c[i, j]$ to $+\infty$ to indicate that it is impossible to pay an amount j using only coins of type 1.

Figure 8.3 illustrates the instance given earlier, where we have to pay 8 units with coins worth 1, 4 and 6 units. For example, $c[3,8]$ is obtained in this case as the smaller of $c[2,8]= 2$ and $1 + c[3,8 - d_3]= 1 + c[3,2]= 3$. The entries elsewhere in the table are obtained similarly. The answer to this particular instance is that we can pay 8 units using only two coins. In fact the table gives us the solution to our problem for all the instances involving a payment of 8 units or less.

Amount:	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	1	2	3	1	2	3	4	2
$d_3 = 6$	0	1	2	3	1	2	1	2	2

Figure 8.3. Making change using dynamic programming

Here is a more formal version of the algorithm.

```
function coins(N)
    {Gives the minimum number of coins needed to make
     change for N units. Array d[1..n] specifies the coinage:
     in the example there are coins for 1, 4 and 6 units.}
    array d[1..n]= [1,4,6]
    array c[1..n,0..N]
    for i ← 1 to n do c[i,0]← 0
    for i ← 1 to n do
        for j ← 1 to N do
            c[i,j]← if i = 1 and j < d[i] then +∞
                    else if i = 1 then 1 + c[1,j − d[1]]
                    else if j < d[i] then c[i − 1,j]
                    else min(c[i − 1,j],1 + c[i,j − d[i]])
    return c[n,N]
```

If an unlimited supply of coins with a value of 1 unit is available, then we can always find a solution to our problem. If this is not the case, there may be values of N for which no solution is possible. This happens for instance if all the coins represent an even number of units, and we are required to pay an odd number of units. In such instances the algorithm returns the artificial result $+\infty$. Problem 8.9 invites the reader to modify the algorithm to handle a situation where the supply of coins of a particular denomination is limited.

Although the algorithm appears only to say how many coins are required to make change for a given amount, it is easy once the table c is constructed to discover exactly which coins are needed. Suppose we are to pay an amount j using coins of denominations $1, 2, \ldots, i$. Then the value of $c[i,j]$ says how many coins are needed. If $c[i,j]= c[i-1,j]$, no coins of denomination i are necessary, and we move up to $c[i-1,j]$ to see what to do next; if $c[i,j]= 1 + c[i,j-d_i]$, then we hand over one coin of denomination i, worth d_i, and move left to $c[i,j-d_i]$ to see what to do next. If $c[i-1,j]$ and $1 + c[i,j-d_i]$ are both equal to $c[i,j]$, we

may choose either course of action. Continuing in this way, we eventually arrive back at $c[0,0]$, and now there remains nothing to pay. This stage of the algorithm is essentially a greedy algorithm that bases its decisions on the information in the table, and never has to backtrack.

Analysis of the algorithm is straightforward. To see how many coins are needed to make change for N units when n different denominations are available, the algorithm has to fill up an $n \times (N+1)$ array, so the execution time is in $\Theta(nN)$. To see which coins should be used, the search back from $c[n,N]$ to $c[0,0]$ makes $n-1$ steps to the row above (corresponding to not using a coin of the current denomination) and $c[n,N]$ steps to the left (corresponding to handing over a coin). Since each of these steps can be made in constant time, the total time required is in $\Theta(n + c[n,N])$.

8.3 The principle of optimality

The solution to the problem of making change obtained by dynamic programming seems straightforward, and does not appear to hide any deep theoretical considerations. However it is important to realize that it relies on a useful principle called the *principle of optimality*, which in many settings appears so natural that it is invoked almost without thinking. This principle states that in an optimal sequence of decisions or choices, each subsequence must also be optimal. In our example, we took it for granted, when calculating $c[i,j]$ as the lesser of $c[i-1,j]$ and $1 + c[i,j-d_i]$, that if $c[i,j]$ is the optimal way of making change for j units using coins of denominations 1 to i, then $c[i-1,j]$ and $c[i,j-d_i]$ must also give the *optimal* solutions to the instances they represent. In other words, although the only value in the table that really interests us is $c[n,N]$, we took it for granted that all the other entries in the table must also represent optimal choices: and rightly so, for in this problem the principle of optimality applies.

Although this principle may appear obvious, it does not apply to every problem we might encounter. When the principle of optimality does *not* apply, it will probably not be possible to attack the problem in question using dynamic programming. This is the case, for instance, when a problem concerns the optimal use of limited resources. Here the optimal solution to an instance may not be obtained by combining the optimal solutions to two or more subinstances, if the resources used in these subsolutions add up to more than the total resources available.

For example, if the shortest route from Montreal to Toronto passes through Kingston, then that part of the journey from Montreal to Kingston must also follow the shortest possible route, as must the part of the journey from Kingston to Toronto. Thus the principle of optimality applies. However if the fastest way to drive from Montreal to Toronto takes us through Kingston, it does not necessarily follow that it is best to drive as fast as possible from Montreal to Kingston, and then to drive as fast as possible from Kingston to Toronto. If we use too much petrol on the first half of the trip, we may have to fill up somewhere on the second half, losing more time than we gained by driving hard. The sub-trips from Montreal to Kingston, and from Kingston to Toronto, are not independent, since they share a resource, so choosing an optimal solution for one sub-trip may prevent our using an optimal solution for the other. In this situation, the principle of optimality does not apply.

For a second example, consider the problem of finding not the shortest, but the longest simple route between two cities, using a given set of roads. A *simple* route is one that never visits the same spot twice, so this condition rules out infinite routes round and round a loop. If we know that the longest simple route from Montreal to Toronto passes through Kingston, it does *not* follow that it can be obtained by taking the longest simple route from Montreal to Kingston, and then the longest simple route from Kingston to Toronto. It is too much to expect that when these two simple routes are spliced together, the resulting route will also be simple. Once again, the principle of optimality does not apply.

Nevertheless, the principle of optimality applies more often than not. When it does, it can be restated as follows: the optimal solution to any nontrivial instance of a problem is a combination of optimal solutions to *some* of its subinstances. The difficulty in turning this principle into an algorithm is that it is not usually obvious which subinstances are relevant to the instance under consideration. Coming back to the example of finding the shortest route, how can we tell whether the subinstance consisting of finding the shortest route from Montreal to Ottawa is relevant when we want the shortest route from Montreal to Toronto? This difficulty prevents our using an approach similar to divide-and-conquer starting from the original instance and recursively finding optimal solutions to the relevant subinstances, and only to these. Instead, dynamic programming efficiently solves every subinstance to figure out which ones are in fact relevant; only then are these combined into an optimal solution to the original instance.

8.4 The knapsack problem (2)

As in Section 6.5, we are given a number of objects and a knapsack. This time, however, we suppose that the objects may *not* be broken into smaller pieces, so we may decide either to take an object or to leave it behind, but we may not take a fraction of an object. For $i = 1, 2, \ldots, n$, suppose that object i has a positive weight w_i and a positive value v_i. The knapsack can carry a weight not exceeding W. Our aim is again to fill the knapsack in a way that maximizes the value of the included objects, while respecting the capacity constraint. Let x_i be 0 if we elect not to take object i, or 1 if we include object i. In symbols the new problem may be stated as:

$$\text{maximize } \sum_{i=1}^{n} x_i v_i \qquad \text{subject to } \sum_{i=1}^{n} x_i w_i \leq W$$

where $v_i > 0$, $w_i > 0$ and $x_i \in \{0, 1\}$ for $1 \leq i \leq n$. Here the conditions on v_i and w_i are constraints on the instance; those on x_i are constraints on the solution. Since the problem closely resembles the one in Section 6.5, it is natural to enquire first whether a slightly modified version of the greedy algorithm we used before will still work. Suppose then that we adapt the algorithm in the obvious way, so that it looks at the objects in order of decreasing value per unit weight. If the knapsack is not full, the algorithm should select a *complete* object if possible before going on to the next.

Unfortunately the greedy algorithm turns out not to work when x_i is required to be 0 or 1. For example, suppose we have three objects available, the first of which

weighs 6 units and has a value of 8, while the other two weigh 5 units each and have a value of 5 each. If the knapsack can carry 10 units, then the optimal load includes the two lighter objects for a total value of 10. The greedy algorithm, on the other hand, would begin by choosing the object that weighs 6 units, since this is the one with the greatest value per unit weight. However if objects cannot be broken the algorithm will be unable to use the remaining capacity in the knapsack. The load it produces therefore consists of just one object with a value of only 8.

To solve the problem by dynamic programming, we set up a table $V[1..n, 0..W]$, with one row for each available object, and one column for each weight from 0 to W. In the table, $V[i, j]$ will be the maximum value of the objects we can transport if the weight limit is j, $0 \leq j \leq W$, and if we only include objects numbered from 1 to i, $1 \leq i \leq n$. The solution of the instance can therefore be found in $V[n, W]$.

The parallel with the problem of making change is close. As there, the principle of optimality applies. We may fill in the table either row by row or column by column. In the general situation, $V[i, j]$ is the larger (since we are trying to maximize value) of $V[i-1, j]$ and $V[i-1, j-w_i]+v_i$. The first of these choices corresponds to not adding object i to the load. The second corresponds to choosing object i, which has for effect to increase the value of the load by v_i and to reduce the capacity available by w_i. Thus we fill in the entries in the table using the general rule

$$V[i, j] = \max(V[i-1, j], V[i-1, j-w_i]+v_i).$$

For the out-of-bounds entries we define $V[0, j]$ to be 0 when $j \geq 0$, and we define $V[i, j]$ to be $-\infty$ for all i when $j < 0$. The formal statement of the algorithm, which closely resembles the function *coins* of the previous section, is left as an exercise for the reader; see Problem 8.11.

Figure 8.4 gives an example of the operation of the algorithm. In the figure there are five objects, whose weights are respectively 1, 2, 5, 6 and 7 units, and whose values are 1, 6, 18, 22 and 28. Their values per unit weight are thus 1.00, 3.00, 3.60, 3.67 and 4.00. If we can carry a maximum of 11 units of weight, then the table shows that we can compose a load whose value is 40.

Weight limit:	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1, v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2, v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5, v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6, v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5 = 7, v_5 = 28$	0	1	6	7	7	18	22	28	29	34	35	40

Figure 8.4. The knapsack using dynamic programming

Just as for the problem of making change, the table V allows us to recover not only the value of the optimal load we can carry, but also its composition. In our example, we begin by looking at $V[5, 11]$. Since $V[5, 11] = V[4, 11]$ but $V[5, 11] \neq V[4, 11-w_5]+v_5$, an optimal load cannot include object 5. Next $V[4, 11] \neq V[3, 11]$

but $V[4,11] = V[3,11 - w_4] + v_4$, so an optimal load must include object 4. Now $V[3,5] \neq V[2,5]$ but $V[3,5] = V[2,5 - w_3] + v_3$, so we must include object 3. Continuing thus, we find that $V[2,0] = V[1,0]$ and $V[1,0] = V[0,0]$, so the optimal load includes neither object 2 nor object 1. In this instance, therefore, there is only one optimal load, consisting of objects 3 and 4.

In this example the greedy algorithm would first consider object 5, since this has the greatest value per unit weight. The knapsack can carry one such object. Next the greedy algorithm would consider object 4, whose value per unit weight is next highest. This object cannot be included in the load without violating the capacity constraint. Continuing in this way, the greedy algorithm would look at objects 3, 2 and 1, in that order, finally ending up with a load consisting of objects 5, 2 and 1, for a total value of 35. Once again we see that the greedy algorithm does not work when objects cannot be broken.

Analysis of the dynamic programming algorithm is straightforward, and closely parallels the analysis of the algorithm for making change. We find that a time in $\Theta(nW)$ is necessary to construct the table V, and that the composition of the optimal load can then be determined in a time in $O(n + W)$.

8.5 Shortest paths

Let $G = \langle N, A \rangle$ be a directed graph; N is the set of nodes and A is the set of edges. Each edge has an associated nonnegative length. We want to calculate the length of the shortest path between each pair of nodes. Compare this to Section 6.4 where we were looking for the length of the shortest paths from one particular node, the source, to all the others.

As before, suppose the nodes of G are numbered from 1 to n, so $N = \{1, 2, \ldots, n\}$, and suppose a matrix L gives the length of each edge, with $L[i,i] = 0$ for $i = 1, 2, \ldots, n$, $L[i,j] \geq 0$ for all i and j, and $L[i,j] = \infty$ if the edge (i,j) does not exist.

The principle of optimality applies: if k is a node on the shortest path from i to j, then the part of the path from i to k, and the part from k to j, must also be optimal.

We construct a matrix D that gives the length of the shortest path between each pair of nodes. The algorithm initializes D to L, that is, to the direct distances between nodes. It then does n iterations. After iteration k, D gives the length of the shortest paths that only use nodes in $\{1, 2, \ldots, k\}$ as intermediate nodes. After n iterations, D therefore gives the length of the shortest paths using any of the nodes in N as an intermediate node, which is the result we want. At iteration k, the algorithm must check for each pair of nodes (i,j) whether or not there exists a path from i to j passing through node k that is better than the present optimal path passing only through nodes in $\{1, 2, \ldots, k-1\}$. If D_k represents the matrix D after the k–th iteration (so $D_0 = L$), the necessary check can be implemented by

$$D_k[i,j] = \min(D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j]),$$

where we use the principle of optimality to compute the length of the shortest path from i to j passing through k. We have also tacitly used the fact that an optimal path through k does not visit k twice.

At the k–th iteration the values in the k–th row and the k–th column of D do not change, since $D[k,k]$ is always zero. It is therefore not necessary to protect these values when updating D. This allows us to get away with using only a single $n \times n$ matrix D, whereas at first sight it might seem necessary to use two such matrices, one containing the values of D_{k-1} and the other the values of D_k, or even a matrix $n \times n \times n$.

The algorithm, known as *Floyd's algorithm*, follows.

```
function Floyd(L[1..n, 1..n]): array [1..n, 1..n]
    array D[1..n, 1..n]
    D ← L
    for k ← 1 to n do
        for i ← 1 to n do
            for j ← 1 to n do
                D[i, j] ← min(D[i, j], D[i, k]+D[k, j])
    return D
```

Figure 8.5 gives an example of the way the algorithm works.

$$D_0 = L = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix}$$

$$D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \qquad D_2 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \qquad D_4 = \begin{pmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

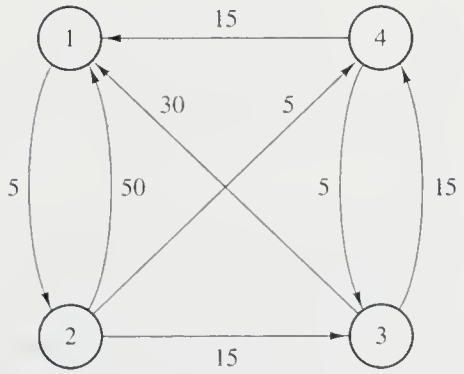

Figure 8.5. Floyd's algorithm at work

It is obvious that this algorithm takes a time in $\Theta(n^3)$. We can also use Dijkstra's algorithm to solve the same problem; see Section 6.4. In this case we have to apply the algorithm n times, each time choosing a different node as the source. If we use the version of Dijkstra's algorithm that works with a matrix of distances, the total computation time is in $n \times \Theta(n^2)$, that is, in $\Theta(n^3)$. The order is the same as for Floyd's algorithm, but the simplicity of the latter means that it will probably have a smaller hidden constant and thus be faster in practice. Compilers are good at optimizing **for**-loops, too. On the other hand, if we use the version of Dijkstra's algorithm that works with a heap, and hence with lists of the distances to adjacent nodes, the total time is in $n \times \Theta((a + n)\log n)$, that is, in $\Theta((an + n^2)\log n)$, where a is the number of edges in the graph. If the graph is sparse ($a \ll n^2$), it may be preferable to use Dijkstra's algorithm n times; if the graph is dense ($a \approx n^2$), it is better to use Floyd's algorithm.

We usually want to know where the shortest path goes, not just its length. In this case we use a second matrix P, all of whose elements are initialized to 0. The innermost loop of the algorithm becomes

if $D[i,k]+D[k,j]< D[i,j]$ **then** $D[i,j] \leftarrow D[i,k]+D[k,j]$
$P[i,j] \leftarrow k$

When the algorithm stops, $P[i,j]$ contains the number of the last iteration that caused a change in $D[i,j]$. To recover the shortest path from i to j, look at $P[i,j]$. If $P[i,j]= 0$, then $D[i,j]$ never changed, and the shortest path is directly along the edge (i,j); otherwise, if $P[i,j]= k$, the shortest path from i to j passes through k. Look recursively at $P[i,k]$ and $P[k,j]$ to find any other intermediate nodes along the shortest path.

If we take the graph of Figure 8.5 as an example, P becomes

$$P = \begin{pmatrix} 0 & 0 & 4 & 2 \\ 4 & 0 & 4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}.$$

Since $P[1,3]= 4$, the shortest path from 1 to 3 passes through 4. Looking now at $P[1,4]$ and $P[4,3]$, we discover that between 1 and 4 we have to go via 2, but that from 4 to 3 we proceed directly. Finally we see that the trips from 1 to 2, and from 2 to 4, are also direct. The shortest path from 1 to 3 is thus $1,2,4,3$.

If we allow edges in the graph to have negative lengths, the notion of "shortest path" loses much of its meaning: if the graph includes a cycle whose total length is negative, then the more often we go round the negative cycle, the shorter our path will be! Problem 8.17 asks what happens to Floyd's algorithm if we give it a graph with negative edges, but no negative cycles. Even if a graph has negative cycles, it still makes sense to ask for the shortest simple paths. (Remember that a simple path is one that never visits the same node twice.) No efficient algorithm is known for finding shortest simple paths in graphs that may have edges of negative length. The situation is the same for the problem of finding longest simple paths, mentioned in Section 8.3: no efficient algorithm is known. These problems are both $\mathcal{NP}$–complete; see Chapter 12.

8.6 Chained matrix multiplication

Recall that the product C of a $p \times q$ matrix A and a $q \times r$ matrix B is the $p \times r$ matrix given by

$$c_{ij} = \sum_{k=1}^{q} a_{ik} b_{kj}, \qquad 1 \le i \le p, 1 \le j \le r.$$

Algorithmically, we can express this as

```
for i ← 1 to p do
    for j ← 1 to r do
        C[i, j] ← 0
        for k ← 1 to q do
            C[i, j] ← C[i, j] + A[i, k]B[k, j]
```

from which it is clear that a total of pqr scalar multiplications are required to calculate the matrix product using this algorithm. (In this section we shall not consider the possibility of using a better matrix multiplication algorithm, such as Strassen's algorithm, described in Section 7.6.)

Suppose now we want to calculate the product of more than two matrices. Matrix multiplication is associative, so we can compute the matrix product

$$M = M_1 M_2 \cdots M_n$$

in a number of ways, which all give the same answer:

$$\begin{aligned} M &= (\cdots((M_1 M_2) M_3) \cdots M_n) \\ &= (M_1 (M_2 (M_3 \cdots (M_{n-1} M_n) \cdots))) \\ &= (\cdots((M_1 M_2)(M_3 M_4)) \cdots), \end{aligned}$$

and so on. However matrix multiplication is not commutative, so we are not allowed to change the order of the matrices in these arrangements.

The choice of a method of computation can have a considerable influence on the time required. Suppose, for example, that we want to calculate the product $ABCD$ of four matrices, where A is 13×5, B is 5×89, C is 89×3, and D is 3×34. To measure the efficiency of the different methods, we count the number of scalar multiplications involved. As programmed above, there will be an equal number of scalar additions, plus some housekeeping, so the number of scalar multiplications is a good indicator of overall efficiency. For instance, using $M = ((AB)C)D$, we calculate successively

AB	5785 multiplications
$(AB)C$	3471 multiplications
$((AB)C)D$	1326 multiplications

for a total of 10 582 scalar multiplications. There are five essentially different ways of calculating the product in this case: when the product is expressed as $(AB)(CD)$, we do not differentiate between the method that calculates AB first and CD second,

and the one that starts with CD and then calculates AB, since they both require the same number of multiplications. For each of these five methods, here is the corresponding number of scalar multiplications:

$((AB)C)D$	10582
$(AB)(CD)$	54201
$(A(BC))D$	2856
$A((BC)D)$	4055
$A(B(CD))$	26418

The most efficient method is almost 19 times faster than the slowest.

To find directly the best way to calculate the product, we could simply parenthesize the expression in every possible fashion and count each time how many scalar multiplications are required. Let $T(n)$ be the number of essentially different ways to parenthesize a product of n matrices. Suppose we decide to make the first cut between the i–th and the $(i+1)$–st matrices of the product, thus:

$$M = (M_1 M_2 \cdots M_i)(M_{i+1} M_{i+2} \cdots M_n).$$

There are now $T(i)$ ways to parenthesize the left-hand term and $T(n-i)$ ways to parenthesize the right-hand term. Any of the former may be combined with any of the latter, so for this particular value of i there are $T(i)T(n-i)$ ways of parenthesizing the whole expression. Since i can take any value from 1 to $n-1$, we obtain finally the following recurrence for $T(n)$:

$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i).$$

Adding the obvious initial condition $T(1) = 1$, we can use the recurrence to calculate any required value of T. The following table gives some values of $T(n)$.

n	1	2	3	4	5	10	15
$T(n)$	1	1	2	5	14	4862	2674440

The values of $T(n)$ are called the *Catalan numbers*.

For each way that parentheses can be inserted in the expression for M, it takes a time in $\Omega(n)$ to count the number of scalar multiplications required (at least, if we do not try to be subtle). Since $T(n)$ is in $\Omega(4^n/n^2)$ (combine the results of Problems 8.24 and 1.42), finding the best way to calculate M using the direct approach requires a time in $\Omega(4^n/n)$. This method is therefore impracticable for large values of n: there are too many ways in which parentheses can be inserted for us to look at them all.

A little experimenting shows that none of the obvious greedy algorithms will allow us to compute matrix products in an optimal way; see Problem 8.20. Fortunately, the principle of optimality applies to this problem. For instance, if the best way of multiplying all the matrices requires us to make the first cut between the i–th and the $(i+1)$–st matrices of the product, then both the subproducts $M_1 M_2 \cdots M_i$ and $M_{i+1} M_{i+2} \cdots M_n$ must also be calculated in an optimal way. This suggests

that we should consider using dynamic programming. We construct a table m_{ij}, $1 \le i \le j \le n$, where m_{ij} gives the optimal solution—that is, the required number of scalar multiplications—for the part $M_i M_{i+1} \cdots M_j$ of the required product. The solution to the original problem is thus given by m_{1n}.

Suppose the dimensions of the matrices are given by a vector $d[0..n]$ such that the matrix M_i, $1 \le i \le n$, is of dimension $d_{i-1} \times d_i$. We build the table m_{ij} diagonal by diagonal: diagonal s contains the elements m_{ij} such that $j - i = s$. The diagonal $s = 0$ therefore contains the elements m_{ii}, $1 \le i \le n$, corresponding to the "products" M_i. Here there is no multiplication to be done, so $m_{ii} = 0$ for every i. The diagonal $s = 1$ contains the elements $m_{i,i+1}$ corresponding to products of the form $M_i M_{i+1}$. Here we have no choice but to compute the product directly, which we can do using $d_{i-1} d_i d_{i+1}$ scalar multiplications, as we saw at the beginning of the section. Finally when $s > 1$ the diagonal s contains the elements $m_{i,i+s}$ corresponding to products of the form $M_i M_{i+1} \cdots M_{i+s}$. Now we have a choice: we can make the first cut in the product after any of the matrices $M_i, M_{i+1}, \ldots, M_{i+s-1}$. If we make the cut after M_k, $i \le k < i + s$, we need m_{ik} scalar multiplications to calculate the left-hand term, $m_{k+1,i+s}$ to calculate the right-hand term, and then $d_{i-1} d_k d_{i+s}$ to multiply the two resulting matrices to obtain the final result. To find the optimum, we choose the cut that minimizes the required number of scalar multiplications.

Summing up, we fill the table m_{ij} using the following rules for $s = 0, 1, \ldots, n-1$.

$$\begin{aligned}
s = 0 : \quad & m_{ii} = 0 && i = 1, 2, \ldots, n \\
s = 1 : \quad & m_{i,i+1} = d_{i-1} d_i d_{i+1} && i = 1, 2, \ldots, n-1 \\
1 < s < n : \quad & m_{i,i+s} = \min_{i \le k < i+s} (m_{ik} + m_{k+1,i+s} + d_{i-1} d_k d_{i+s}) \\
& && i = 1, 2, \ldots, n-s
\end{aligned}$$

It is only for clarity that the second case need be written out explicitly, as it falls under the general case with $s = 1$.

To apply this to the example, we want to calculate the product $ABCD$ of four matrices, where A is 13×5, B is 5×89, C is 89×3, and D is 3×34. The vector d is therefore $(13, 5, 89, 3, 34)$. For $s = 1$, we find $m_{12} = 5785$, $m_{23} = 1335$ and $m_{34} = 9078$. Next, for $s = 2$ we obtain

$$\begin{aligned}
m_{13} &= \min(m_{11} + m_{23} + 13 \times 5 \times 3, m_{12} + m_{33} + 13 \times 89 \times 3) \\
&= \min(1530, 9256) = 1530 \\
m_{24} &= \min(m_{22} + m_{34} + 5 \times 89 \times 34, m_{23} + m_{44} + 5 \times 3 \times 34) \\
&= \min(24208, 1845) = 1845.
\end{aligned}$$

Finally for $s = 3$

$$\begin{aligned}
m_{14} = \min(&m_{11} + m_{24} + 13 \times 5 \times 34, && \{k = 1\} \\
&m_{12} + m_{34} + 13 \times 89 \times 34, && \{k = 2\} \\
&m_{13} + m_{44} + 13 \times 3 \times 34) && \{k = 3\} \\
= \min(&4055, 54201, 2856) = 2856.
\end{aligned}$$

The complete array m is shown in Figure 8.6.

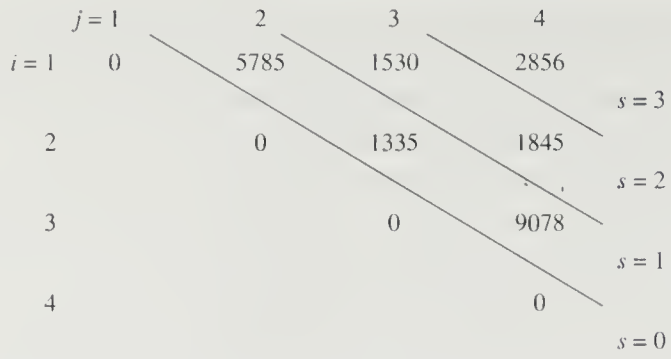

Figure 8.6. An example of the chained matrix multiplication algorithm

Once again, we usually want to know not just the number of scalar multiplications necessary to compute the product M, but also how to perform this computation efficiently. As in Section 8.5, we do this by adding a second array to keep track of the choices we have made. Let this new array be *bestk*. Now when we compute m_{ij}, we save in *bestk*$[i, j]$ the value of k that corresponds to the minimum term among those compared. When the algorithm stops, *bestk*$[1, n]$ tells us where to make the first cut in the product. Proceeding recursively on both the terms thus produced, we can reconstruct the optimal way of parenthesizing M. Problems 8.21 and 8.22 invite you to fill in the details.

For $s > 0$ there are $n - s$ elements to be computed in the diagonal s; for each of these we must choose between s possibilities given by the different values of k. The execution time of the algorithm is therefore in the exact order of

$$\begin{aligned}\sum_{s=1}^{n-1}(n-s)s &= n\sum_{s=1}^{n-1}s - \sum_{s=1}^{n-1}s^2 \\ &= n^2(n-1)/2 - n(n-1)(2n-1)/6 \\ &= (n^3-n)/6,\end{aligned}$$

where we used Propositions 1.7.14 and 1.7.15 to evaluate the sums. The execution time of the algorithm is thus in $\Theta(n^3)$, better algorithms exist.

8.7 Approaches using recursion

Although dynamic programming algorithms, such as the one just given for calculating m, are efficient, there is something unsatisfactory about the bottom-up approach. A top-down method, whether it be divide-and-conquer, stepwise refinement, or recursion, seems more natural, especially to one who has been taught always to develop programs in this way. A better reason is that the bottom-up approach leads us to compute values that might be completely irrelevant. It is tempting, therefore, to see whether we can achieve the same efficiency in a top-down version of the algorithm.

We illustrate this with the matrix multiplication problem described in the previous section. One simple line of attack is to replace the table m by a function *fm*, which is calculated as required. In other words, we would like to find a function

fm such that $fm(i,j) = m_{ij}$ for $1 \le i \le j \le n$, but that can be calculated recursively, unlike the table m, which we calculated bottom-up.

Writing such a function is simple: all we have to do is to program the rules for calculating m.

```
function fm(i, j)
    if i = j then {only one matrix is involved}
        return 0
    m ← ∞
    for k ← i to j − 1 do
        m ← min(m, fm(i, k)+fm(k + 1, j)+d[i − 1]d[k]d[j])
    return m
```

Here the global array $d[0..n]$ gives the dimensions of the matrices involved, exactly as before. For all the relevant values of k the intervals $[i..k]$ and $[k+1..j]$ concerned in the recursive calls involve less matrices than $[i..j]$. However each recursive call still involves at least one matrix (provided of course $i \le j$ on the original call). Eventually therefore the recursion will stop. To find how many scalar multiplications are needed to calculate $M = M_1 M_2 \ldots M_n$, we simply call $fm(1, n)$.

To analyse this algorithm, let $T(s)$ be the time required to execute a call of $fm(i, i+s)$, where s is the number of matrix multiplications involved in the corresponding product. This is the same s used previously to number the diagonals of the table m. Clearly $T(0) = c$ for some constant c. When $s > 0$, we have to choose the smallest among s terms, each of the form

$$fm(i,k)+fm(k+1,i+s)+d[i-1]d[k]d[i+s], \quad i \le k < i+s.$$

Let b be a constant such that we can execute two scalar multiplications, six scalar additions, a comparison with the previous value of the minimum, and any necessary housekeeping in a time b. Now the time required to evaluate one of these terms is $T(k-i)+T(i+s-k-1)+b$. The total time $T(s)$ required to evaluate $fm(i, i+s)$ is therefore

$$T(s) = \sum_{k=i}^{i+s-1} (T(k-i)+T(i+s-k-1)+b).$$

Writing $m = k - i$, this becomes

$$\begin{aligned} T(s) &= \sum_{m=0}^{s-1} (T(m)+T(s-1-m)+b) \\ &= sb + 2\sum_{m=0}^{s-1} T(m). \end{aligned}$$

Since this implies that $T(s) \ge 2T(s-1)$, we see immediately that $T(s) \ge 2^s T(0)$, so the algorithm certainly takes a time in $\Omega(2^n)$ to find the best way to multiply n matrices. It therefore cannot be competitive with the algorithm using dynamic programming, which takes a time in $\Theta(n^3)$.

The algorithm can be speeded up if we are clever enough to avoid recursive calls when $d[i-1]d[k]d[i+s]$ is already greater than the previous value of the minimum. The improvement will depend on the instance, but is most unlikely to make an algorithm that takes exponential time competitive with one that only takes polynomial time.

To find an upper bound on the time taken by the recursive algorithm, we use constructive induction. Looking at the form of the recursion for T, and remembering that $T(s) \geq 2^s T(0)$, it seems possible that T might be bounded by a power of some constant larger than 2: so let us try proving $T(s) \leq a3^s$, for some appropriate constant a. Take this as the induction hypothesis, and assume it is true for all $m < s$. On substituting in the recurrence we obtain

$$\begin{aligned} T(s) &\leq sd + 2\sum_{m=0}^{s-1} a3^m \\ &= sd + a3^s - a, \end{aligned}$$

where we used Proposition 1.7.10 to compute the sum. Unfortunately, this does not allow us to conclude that $T(s) \leq a3^s$. However, if we adopt the tactic recommended in Section 1.6.4 and strengthen the induction hypothesis, things work better. As the strengthened induction hypothesis, suppose $T(m) \leq a3^m - b$ for $m < s$, where b is a new unknown constant. Now when we substitute into the recurrence we obtain

$$\begin{aligned} T(s) &\leq sd + 2\sum_{m=0}^{s-1} (a3^m - b) \\ &= s(d-2b) + a3^s - a. \end{aligned}$$

This is sufficient to ensure that $T(s) \leq a3^s - b$ provided $b \geq d/2$ and $a \geq b$. To start the induction, we require that $T(0) \leq a - b$, which is satisfied provided $a \geq T(0) + b$. Summing up, we have proved that $T(s) \leq a3^m - b$ for all s provided $b \geq d/2$ and $a \geq T(0) + b$. The time taken by the recursive algorithm to find the best way of computing a product of n matrices is therefore in $O(3^n)$.

We conclude that a call on the recursive function $fm(1, n)$ is faster than naively trying all possible ways to parenthesize the desired product, which, as we saw, takes a time in $\Omega(4^n/n)$. However it is slower than the dynamic programming algorithm described previously. This illustrates a point made earlier in this chapter. To decide the best way to parenthesize the product $ABCDEFG$, say, fm recursively solves 12 subinstances, including the overlapping $ABCDEF$ and $BCDEFG$, both of which recursively solve $BCDEF$ from scratch. It is this duplication of effort that makes fm inefficient.

8.8 Memory functions

The algorithm in the previous section is not the first we have seen where a simple recursive formulation of a solution leads to an inefficient program, as common subinstances are solved independently more than once. Dynamic programming

allows us to avoid this at the cost of complicating the algorithm. In dynamic programming, too, we may have to solve some irrelevant subinstances, since it is only later that we know exactly which subsolutions are needed. A top-down, recursive algorithm does not have this drawback. Can we perhaps combine the advantages of both techniques, and retain the simplicity of a recursive formulation without losing the efficiency offered by dynamic programming?

One easy way of doing this that works in many situations is to use a *memory function*. To the recursive program we add a table of the necessary size. Initially, all the entries in this table hold a special value to show they have not yet been calculated. Thereafter, whenever we call the function, we first look in the table to see whether it has already been evaluated with the same set of parameters. If so, we return the value in the table. If not, we go ahead and calculate the function. Before returning the calculated value, however, we save it at the appropriate place in the table. In this way it is never necessary to calculate the function twice for the same values of its parameters.

For the recursive algorithm *fm* of Section 8.7, let *mtab* be a table whose entries are all initialized to -1 (since the number of scalar multiplications required to compute a matrix product cannot be negative). The following reformulation of the function *fm*, which uses the table *mtab* as a global variable, combines the clarity of a recursive formulation with the efficiency of dynamic programming.

```
function fm-mem(i, j)
    if i = j then return 0
    if mtab[i, j] ≥ 0 then return mtab[i, j]
    m ← ∞
    for k ← i to j − 1 do
        m ← min(m, fm-mem(i, k) + fm-mem(k + 1, j)
                                + d[i − 1]d[k]d[j])
    mtab[i, j] ← m
    return m
```

As pointed out in Section 8.7, this function may be speeded up by avoiding the recursive calls if $d[i-1]d[k]d[j]$ is already larger than the previous value of m.

We sometimes have to pay a price for using this technique. We saw in Section 8.1.1, for instance, that we can calculate a binomial coefficient $\binom{n}{k}$ using a time in $\Theta(nk)$ and space in $\Theta(k)$. Implemented using a memory function, the calculation takes the same amount of time but needs space in $\Omega(nk)$; see Problem 8.26.

If we use a little more space—the space needed is only multiplied by a constant factor—we can avoid the initialization time needed to set all the entries of the table to some special value. This can be done using *virtual initialization*, described in Section 5.1. This is particularly desirable when only a few values of the function are to be calculated, but we do not know in advance which ones. For an example, see Section 9.1.

8.9 Problems

Problem 8.1. Prove that the total number of recursive calls made during the computation of $C(n,k)$ using the algorithm of Section 8.1.1 is exactly $2\binom{n}{k} - 2$.

Problem 8.2. Calculating the Fibonacci sequence affords another example of the kind of technique introduced in Section 8.1. Which algorithm in Section 2.7.5 uses dynamic programming?

Problem 8.3. Prove that the time needed to calculate $P(n,n)$ using the function P of Section 8.1.2 is in $\Theta(4^n/\sqrt{n})$.

Problem 8.4. Using the algorithm *series* of Section 8.1.2, calculate the probability that team A will win the series if $p = 0.45$ and if four victories are needed to win.

Problem 8.5. Repeat the previous problem with $p = 0.55$. What should be the relation between the answers to the two problems?

Problem 8.6. As in Problem 8.4, calculate the probability that team A will win the series if $p = 0.45$ and if four victories are needed to win. This time, however, calculate the required probability directly as the probability that team A will win 4 or more out of a series of 7 games. (Playing extra games after team A have won the series cannot change the result.)

Problem 8.7. Adapt algorithm *series* of Section 8.1.2 to the case where team A win any given match with probability p and lose it with probability q, but there is also a probability r that the match is tied, so it counts as a win for nobody. Assume that n victories are still required to win the series. Of course we must have $p + q + r = 1$.

Problem 8.8. Show that storage space in $\Theta(n)$ is sufficient to implement the algorithm *series* of Section 8.1.2.

Problem 8.9. Adapt the algorithm *coins* of Section 8.2 so it will work correctly even when the number of coins of a particular denomination is limited.

Problem 8.10. Rework the example illustrated in Figure 8.4, but renumbering the objects in the opposite order (so $w_1 = 7$, $v_1 = 28$, ..., $w_5 = 1$, $v_5 = 1$). Which elements of the table should remain unchanged?

Problem 8.11. Write out the algorithm for filling the table V as described in Section 8.4.

Problem 8.12. When $j < w_i$ in the algorithm for filling the table V described in Section 8.4, we take $V[i-1, j-w_i]$ to be $-\infty$. Can the finished table contain entries that are $-\infty$? If so, what do they indicate? If not, why not?

Problem 8.13. There may be more than one optimal solution to an instance of the knapsack problem. Using the table V described in Section 8.4, can you find all possible optimal solutions to an instance, or only one? If so, how? If not, why not?

Problem 8.14. An instance of the knapsack problem described in Section 8.4 may have several different optimal solutions. How would you discover this? Does the table V allow you to recover more than one solution in this case?

Problem 8.15. In Section 8.4 we assumed that we had available n objects numbered 1 to n. Suppose instead that we have n *types* of object available, with an adequate supply of each type. Formally, this simply replaces the constraint that x_i must be 0 or 1 by the looser constraint that x_i must be a nonnegative integer. Adapt the dynamic programming algorithm of Section 8.4 so it will handle this new problem.

Problem 8.16. Adapt your algorithm of Problem 8.15 so it will work even when the number of objects of a given type is limited.

Problem 8.17. Does Floyd's algorithm (see Section 8.5) work on a graph that has some edges whose lengths are negative, but that does not include a negative cycle? Justify your answer.

Problem 8.18. (Warshall's algorithm) As for Floyd's algorithm (see Section 8.5) we are concerned with finding paths in a graph. In this case, however, the length of the edges is of no interest; only their existence is important. Let the matrix L be such that $L[i, j] = \mathit{true}$ if the edge (i, j) exists, and $L[i, j] = \mathit{false}$ otherwise. We want to find a matrix D such that $D[i, j] = \mathit{true}$ if there exists at least one path from i to j, and $D[i, j] = \mathit{false}$ otherwise. Adapt Floyd's algorithm for this slightly different case.
Note: We are looking for the *reflexive transitive closure* of the graph in question.

Problem 8.19. Find a significantly better algorithm for the preceding problem in the case when the matrix L is symmetric, that is, when $L[i, j] = L[j, i]$.

Problem 8.20. We (vainly) hope to find a greedy algorithm for the chained matrix multiplication problem; see Section 8.6. Suppose we are to calculate

$$M = M_1 M_2 \cdots M_n,$$

where matrix M_i is $d_{i-1} \times d_i$, $1 \leq i \leq n$. For each of the following suggested techniques, provide a counterexample where the technique does not work.

(a) First multiply the matrices M_i and M_{i+1} whose common dimension d_i is smallest, and continue in the same way.

(b) First multiply the matrices M_i and M_{i+1} whose common dimension d_i is largest, and continue in the same way.

(c) First multiply the matrices M_i and M_{i+1} that minimize the product $d_{i-1}d_i d_{i+1}$, and continue in the same way.

(d) First multiply the matrices M_i and M_{i+1} that maximize the product $d_{i-1}d_i d_{i+1}$, and continue in the same way.

Problem 8.21. Write out in detail the algorithm for calculating the values of m_{ij} described in Section 8.6.

Problem 8.22. Adapt your algorithm for the previous problem so that not only does it calculate m_{ij}, but it also says how the matrix product should be calculated to achieve the optimal value of m_{1n}.

Problem 8.23. What is wrong with the following simple argument? "The algorithm for calculating the values of m given in Section 8.6 has essentially to fill in the entries in just over half of an $n \times n$ table. Its execution time is thus clearly in $\Theta(n^2)$."

Problem 8.24. Let $T(n)$ be a Catalan number; see Section 8.6. Prove that

$$T(n) = \frac{1}{n}\binom{2n-2}{n-1}.$$

Problem 8.25. Prove that the number of ways to cut an n-sided convex polygon into $n-2$ triangles using diagonal lines that do not cross is $T(n-1)$, the $(n-1)$–st Catalan number; see Section 8.6. For example, a hexagon can be cut in 14 different ways, as shown in Figure 8.7.

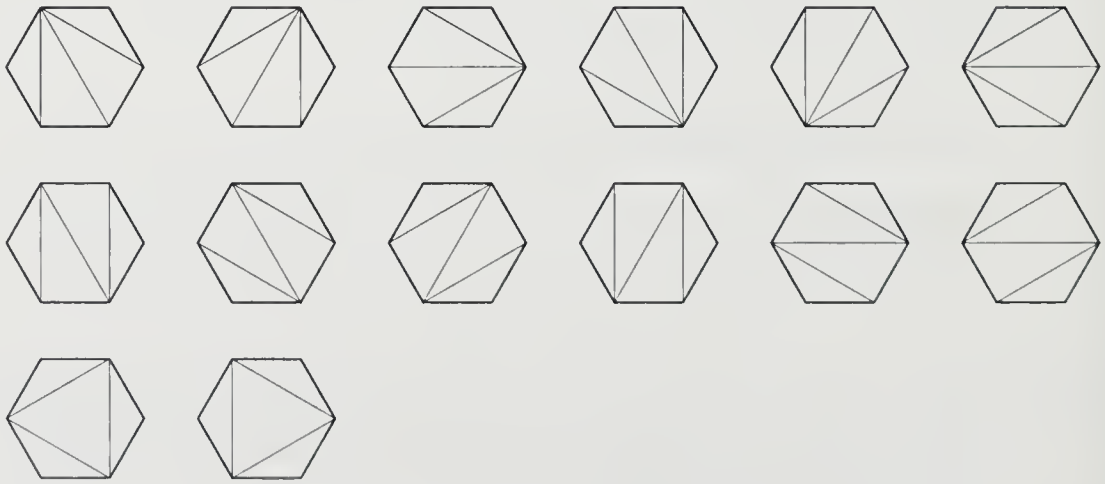

Figure 8.7. Cutting a hexagon into triangles

Problem 8.26. Show how to calculate (i) a binomial coefficient, and (ii) the function *series*(n, p) of Section 8.1.2 using a memory function.

Problem 8.27. Show how to solve (i) the problem of making change, and (ii) the knapsack problem of Section 8.4 using a memory function.

Problem 8.28. Consider the alphabet $\sum = \{a, b, c\}$. The elements of $\sum$ have the following multiplication table, where the rows show the left-hand symbol and the columns show the right-hand symbol.

	a	b	c
a	b	b	a
b	c	b	a
c	a	c	c

Thus $ab = b$, $ba = c$, and so on. Note that the multiplication defined by this table is neither commutative nor associative.
Find an efficient algorithm that examines a string $x = x_1x_2 \cdots x_n$ of characters of $\sum$ and decides whether or not it is possible to parenthesize x in such a way that the value of the resulting expression is a. For instance, if $x = bbbba$, your algorithm should return "yes" because $(b(bb))(ba) = a$. This expression is not unique. For example, $(b(b(b(ba)))) = a$ as well. In terms of n, the length of the string x, how much time does your algorithm take?

Problem 8.29. Modify your algorithm from the previous problem so it returns the number of different ways of parenthesizing x to obtain a.

Problem 8.30. Let u and v be two strings of characters. We want to transform u into v with the smallest possible number of operations of the following three types: delete a character, add a character, or change a character. For instance, we can transform $abbac$ into $abcbc$ in three stages:

$$\begin{aligned} abbac &\rightarrow abac \quad \text{(delete } b\text{)} \\ &\rightarrow ababc \quad \text{(add } b\text{)} \\ &\rightarrow abcbc \quad \text{(change } a \text{ into } c\text{)}. \end{aligned}$$

Show that this transformation is not optimal.
Write a dynamic programming algorithm that finds the minimum number of operations needed to transform u into v and tells us what these operations are. As a function of the lengths of u and v, how much time does your algorithm take?

Problem 8.31. You have n objects that you wish to put in order using the relations "$<$" and "$=$". For example, with three objects 13 different orderings are possible.

$$\begin{array}{lllll} a = b = c & a = b < c & a < b = c & a < b < c & a < c < b \\ a = c < b & b < a = c & b < a < c & b < c < a & b = c < a \\ c < a = b & c < a < b & c < b < a & & \end{array}$$

Give a dynamic programming algorithm that can calculate, as a function of n, the number of different possible orderings. Your algorithm should take a time in $O(n^2)$ and space in $O(n)$.

Problem 8.32. There are n trading posts along a river. At any of the posts you can rent a canoe to be returned at any other post downstream. (It is next to impossible to paddle against the current.) For each possible departure point i and each possible arrival point j the cost of a rental from i to j is known. However, it can happen

that the cost of renting from i to j is higher than the total cost of a series of shorter rentals. In this case you can return the first canoe at some post k between i and j and continue your journey in a second canoe. There is no extra charge for changing canoes in this way.
Give an efficient algorithm to determine the minimum cost of a trip by canoe from each possible departure point i to each possible arrival point j. In terms of n, how much time is needed by your algorithm?

Problem 8.33. When we discussed binary search trees in Section 5.5, we mentioned that it is a good idea to keep them balanced. This is true provided all the nodes are equally likely to be accessed. If some nodes are more often accessed than others, however, an unbalanced tree may give better average performance. For example, the tree shown in Figure 8.8 is better than the one in Figure 5.9 if we are interested in minimizing the average number of comparisons with the tree and if the nodes are accessed with the following probabilities.

Node	6	12	18	20	27	34	35
Probability	0.2	0.25	0.05	0.1	0.05	0.3	0.05

More generally, suppose we have an ordered set $c_1 < c_2 < \cdots < c_n$ of n distinct keys. The probability that a request refers to key c_i is p_i, $1 \le i \le n$. Suppose for simplicity that every request refers to a key in the search tree, so $\sum_{i=1}^{n} p_i = 1$. Recall that the depth of the root of a tree is 0, the depth of its children is 1, and so on. If key c_i is held in a node at depth d_i, then $d_i + 1$ comparisons are necessary to find it. For a given tree the average number of comparisons needed is thus

$$C = \sum_{i=1}^{n} p_i(d_i + 1).$$

For example, the average number of comparisons needed with the tree in Figure 8.8 is

$$0.3 + (0.25 + 0.05)\times 2 + (0.2 + 0.1)\times 3 + (0.05 + 0.05)\times 4 = 2.2.$$

(a) Compute the average number of comparisons needed with the tree in Figure 5.9 and verify that the tree in Figure 8.8 is better.

(b) The tree in Figure 8.8 was obtained from the given probabilities using a simple algorithm. Can you guess what this is?

(c) Find yet another search tree for the same set of keys that is even more efficient on the average than Figure 8.8. What conclusion about algorithm design does this reinforce ?

Problem 8.34. Continuing Problem 8.33, design a dynamic programming algorithm to find an optimal binary search tree for a set of keys with given probabilities of access. How much time does your algorithm take as a function of the number of keys? Apply your algorithm to the instance given in Problem 8.33.
Hint: In any search tree where the nodes holding keys $c_i, c_{i+1}, \ldots, c_j$ form a subtree, let C_{ij} be the minimum average number of accesses made to these nodes. In particular, $C_{1,n}$ is the average number of accesses caused by a query to an optimal binary

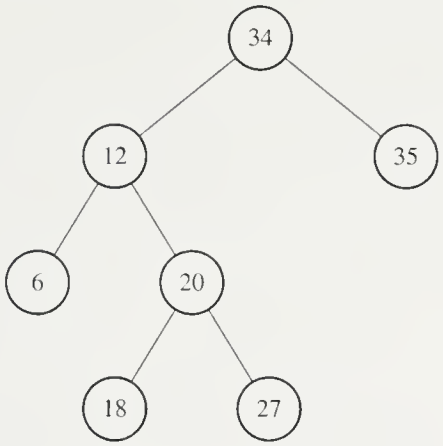

Figure 8.8. Another search tree

search tree and $C_{ii} = p_i$ for each i, $1 \le i \le n$. Invoke the principle of optimality to argue that

$$C_{ij} = \min_{i \le k \le j} (C_{i,k-1} + C_{k+1,j}) + \sum_{k=i}^{j} p_k$$

when $i < j$. Give a dynamic programming algorithm to compute C_{ij} for all $0 \le i \le j \le n$, and an algorithm to find the optimal binary search tree from the C_{ij}'s.

Problem 8.35. Solve Problem 8.34 again. This time your algorithm to compute an optimal binary search tree for a set of n keys must run in a time in $O(n^2)$.
Hint: First prove that $r_{i,j-1} \le r_{ij} \le r_{i+1,j}$ for every $1 \le i \le j \le n$, where r_{ij} is the root of an optimal search subtree containing $c_i, c_{i+1}, \ldots, c_j$ for $1 \le i \le j \le n$ (ties are broken arbitrarily) and $r_{i,i-1} = i$, $1 \le i \le n$.

Problem 8.36. As a function of n, how many binary search trees are there for n distinct keys?

Problem 8.37. Recall that Ackermann's function $A(m, n)$, defined in Problem 5.38, grows extremely rapidly. Give a dynamic programming algorithm to calculate it. Your algorithm must consist simply of two nested loops and recursion is not allowed. Moreover, you are restricted to using a space in $O(m)$ to calculate $A(m, n)$. However you may suppose that a word of storage can hold an arbitrarily large integer.
Hint: Use two arrays $value[0..m]$ and $index[0..m]$ and make sure that $value[i] = A(i, index[i])$ at the end of each trip round the inner loop.

8.10 References and further reading

Several books cover dynamic programming. We mention only Bellman (1957), Bellman and Dreyfus (1962), Nemhauser (1966) and Laurière (1979).

The algorithm in Section 8.2 for making change is discussed in Wright (1975) and Chang and Korsh (1976). For more examples of solving knapsack problems using dynamic programming, see Hu (1981).

The algorithm in Section 8.5 for calculating all shortest paths is due to Floyd (1962). A theoretically more efficient algorithm is known: Fredman (1976) shows how to solve the problem in a time in $O(n^3 \sqrt[3]{\log \log n / \log n})$. The solution to Problem 8.18 is supplied by the algorithm in Warshall (1962). Both Floyd's and Warshall's algorithms are essentially the same as the earlier one in Kleene (1956) to determine the regular expression corresponding to a given finite automaton; see Hopcroft and Ullman (1979). All these algorithms with the exception of Fredman's are unified in Tarjan (1981).

The algorithm in Section 8.6 for chained matrix multiplication is described in Godbole (1973); a more efficient algorithm, able to solve the problem in a time in $O(n \log n)$, can be found in Hu and Shing (1982, 1984). Catalan numbers are discussed in many places, including Sloane (1973) and Purdom and Brown (1985). Memory functions are introduced in Michie (1968); for further details see Marsh (1970).

Problem 8.25 is discussed in Sloane (1973). A solution to Problem 8.30 is given in Wagner and Fischer (1974). Problem 8.31 suggested itself to the authors while grading an exam including a question resembling Problem 3.21: we were curious to know what proportion of all the possible answers was represented by the 69 different answers suggested by the students!

Problem 8.34 on the construction of optimal binary search trees comes from Gilbert and Moore (1959), where it is extended to the possibility that the requested key may not be in the tree. The improvement considered in Problem 8.35 comes from Knuth (1971, 1973) but a simpler and more general solution is given by Yao (1980), who also gives a sufficient condition for certain dynamic programming algorithms that run in cubic time to be transformable automatically into quadratic-time algorithms. The optimal search tree for the 31 most common words in English is compared in Knuth (1973) with the tree obtained using the obvious greedy algorithm suggested in Problem 8.33(b).

Important dynamic programming algorithms we have not mentioned include the one in Kasimi (1965) and Younger (1967) that takes cubic time to carry out the systematic analysis of any context-free language (see Hopcroft and Ullman 1979) and the one in Held and Karp (1962) that solves the travelling salesperson problem (see Sections 12.5.2 and 13.1.2) in a time in $O(n^2 2^n)$, much better than the time in $\Omega(n!)$ required by the naive algorithm.

Chapter 9

Exploring graphs

A great many problems can be formulated in terms of graphs. We have seen, for instance, the shortest route problem and the problem of the minimum spanning tree. To solve such problems, we often need to look at all the nodes, or all the edges, of a graph. Sometimes the structure of the problem is such that we need only visit some of the nodes, or some of the edges. Up to now the algorithms we have seen have implicitly imposed an order on these visits: it was a case of visiting the nearest node, or the shortest edge, and so on. In this chapter we introduce some general techniques that can be used when no particular order of visits is required.

9.1 Graphs and games: An introduction

Consider the following game. It is one of the many variants of Nim, also known as the Marienbad game. Initially there is a heap of matches on the table between two players. The first player may remove as many matches as he likes, except that he must take at least one and he must leave at least one. There must therefore be at least two matches in the initial heap. Thereafter, each player in turn must remove at least one match and at most twice the number of matches his opponent just took. The player who removes the last match wins. There are no draws.

Suppose that at some stage in this game you find yourself in front of a pile of five matches. Your opponent has just picked up two matches, and now it is your turn to play. You may take one, two, three or four matches: however you may not take all five, since the rules forbid taking more than twice what your opponent just took. What should you do?

Most people playing this kind of game begin to run through the possibilities in their heads: "If I take four matches, that leaves just one for my opponent, which he can take and win; if I take three, that leaves two for my opponent, and again he

can take them and win; if I take two, the same thing happens; but if I take just one, then he will have four matches in front of him of which he can only take one or two. In this case he doesn't win at once, so it is certainly my best move." By looking just one move ahead in this simple situation the player can determine what to do next. In a more complicated example, he may have several possible moves. To choose the best it may be necessary to consider not just the situation after his own move, but to look further ahead to see how his opponent can counter each possible move. And then he might have to think about his own move after each possible counter, and so on.

To formalize this process of looking ahead, we represent the game by a directed graph. Each node in the graph corresponds to a position in the game, and each edge corresponds to a move from one position to another. (In some contexts, for example in the reports of chess games, a move consists of an action by one player together with the reply from his opponent. In such contexts the term *half-move* is used to denote the action by just one player. In this book we stick to the simpler terminology and call each player's action a *move*.) A position in the game is not specified merely by the number of matches that remain on the table. It is also necessary to know the upper limit on the number of matches that may be taken on the next move. However it is not necessary to know whose turn it is to play, since the rules are the same for both players (unlike games such as 'fox and geese', where the players have different aims and forces). The nodes of the graph corresponding to this game are therefore pairs $\langle i, j \rangle$. In general, $\langle i, j \rangle$, $1 \leq j \leq i$, indicates that i matches remain on the table, and that any number of them between 1 and j may be taken on the next move. The edges leaving this position, that is, the moves that can be made, go to the j nodes $\langle i-k, \min(2k, i-k) \rangle$, $1 \leq k \leq j$. The node corresponding to the initial position in a game with n matches is $\langle n, n-1 \rangle$, $n \geq 2$. The position $\langle 0, 0 \rangle$ loses the game: if a player is in this position when it is his turn to move, his opponent has just taken the last match and won.

Figure 9.1 shows part of the graph corresponding to this game. In fact, it is the part of the graph needed by the player in the example above who faces a heap of five matches of which he may take four: this is the position $\langle 5, 4 \rangle$. No positions of the form $\langle i, 0 \rangle$ appear except for the losing position $\langle 0, 0 \rangle$. Such positions cannot be reached in the course of a game, so they are of no interest. Similarly nodes $\langle i, j \rangle$ with j odd and $j < i - 1$ cannot be reached from any initial position, so they too are omitted. As we explain in a moment, the square nodes represent losing positions and the round nodes are winning positions. The heavy edges correspond to winning moves: in a winning position, choose one of the heavy edges to win. There are no heavy edges leaving a losing position, corresponding to the fact that such positions offer no winning move. We observe that the player who must move first in a game with two, three or five matches has no winning strategy, whereas he does have such a strategy in the game with four matches.

To decide which are the winning positions and which the losing positions, we start at the losing position $\langle 0, 0 \rangle$ and work back. This node has no successor, and a player who finds himself in this position loses the game. In any of the nodes $\langle 1, 1 \rangle$, $\langle 2, 2 \rangle$ or $\langle 3, 3 \rangle$, a player can make a move that puts his opponent in the losing position. These three nodes are therefore winning nodes. From $\langle 2, 1 \rangle$ the

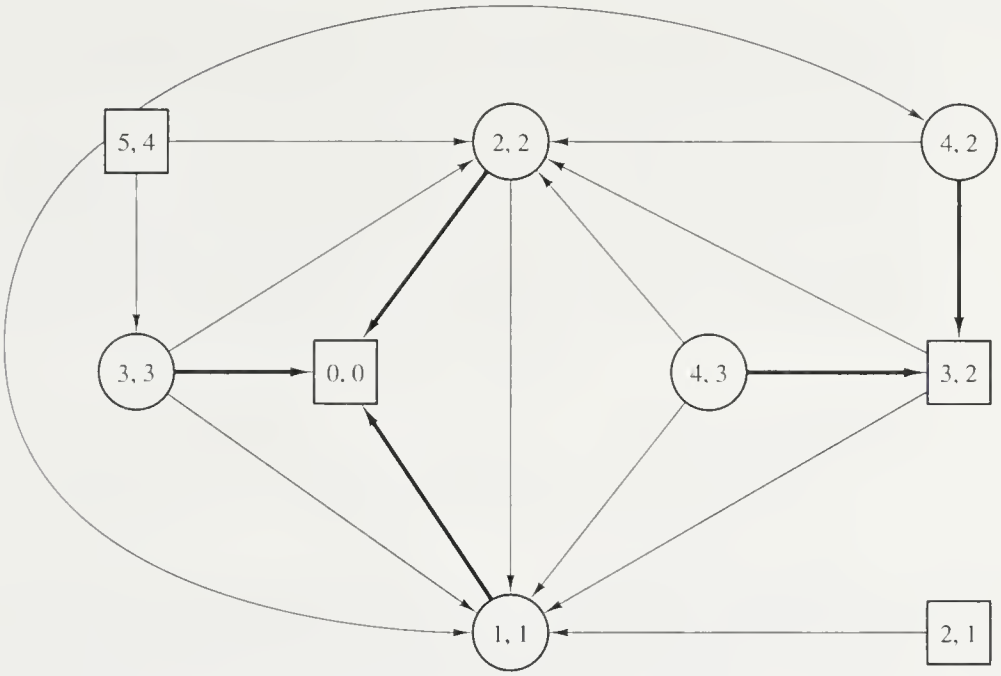

Figure 9.1. Part of a game graph

only possible move is to $\langle 1,1\rangle$. In position $\langle 2,1\rangle$ a player is therefore bound to put his opponent in a winning position, so $\langle 2,1\rangle$ itself is a losing position. A similar argument applies to the losing position $\langle 3,2\rangle$. Two moves are possible, but they both leave the opponent in a winning position, so $\langle 3,2\rangle$ itself is a losing position. From either $\langle 4,2\rangle$ or $\langle 4,3\rangle$ there is a move available that puts the opponent in a losing position, namely $\langle 3,2\rangle$; hence both these nodes are winning positions. Finally the four possible moves from $\langle 5,4\rangle$ all leave the opponent in a winning position, so $\langle 5,4\rangle$ is a losing position.

On a larger graph this process of labelling winning and losing positions can be continued backwards as required. The rules we have been applying can be summed up as follows: a position is a winning position if *at least one* of its successors is a losing position, for then the player can move to put his opponent in a losing position; a position is a losing position if *all* its successors are winning positions, for then the player cannot avoid leaving his opponent in a winning position. The following algorithm therefore determines whether a position is winning or losing.

```
function recwin(i, j)
    {Returns true if and only if node ⟨i, j⟩ is winning;
     we assume 0 ≤ j ≤ i}
    for k ← 1 to j do
        if not recwin(i − k, min(2k, i − k))
            then return true
    return false
```

This algorithm suffers from the same defect as algorithm *Fibrec* in Section 2.7.5: it calculates the same value over and over. For instance, *recwin*$(5,4)$ returns *false*, having called successively *recwin*$(4,2)$, *recwin*$(3,3)$, *recwin*$(2,2)$ and *recwin*$(1,1)$, but *recwin*$(3,3)$ too calls *recwin*$(2,2)$ and *recwin*$(1,1)$.

There are two obvious approaches to removing this inefficiency. The first, using dynamic programming, requires us to create a Boolean array G such that $G[i,j] = true$ if and only if $\langle i,j \rangle$ is a winning position. As usual with dynamic programming, we proceed in a bottom-up fashion, calculating $G[r,s]$ for $1 \leq s \leq r < i$, as well as the values of $G[i,s]$ for $1 \leq s < j$, before calculating $G[i,j]$.

```
procedure dynwin(n)
    {For each 1 ≤ j ≤ i ≤ n, sets G[i, j] to true
     if and only if position ⟨i, j⟩ is winning}
    G[0,0] ← false
    for i ← 1 to n do
        for j ← 1 to i do
            k ← 1
            while k < j and G[i − k, min(2k, i − k)] do
                k ← k + 1
            G[i, j] ← not G[i − k, min(2k, i − k)]
```

In this context dynamic programming leads us to calculate wastefully some entries of the array G that are never needed. For instance, we know $\langle 15,14 \rangle$ is a winning position as soon as we discover that its second successor $\langle 13,4 \rangle$ is a losing position. It is no longer of interest to know whether the next successor $\langle 12,6 \rangle$ is a winning or a losing position. In fact, only 28 nodes are really useful when we calculate $G[15,14]$, although the dynamic programming algorithm determines 121 of them. About half this work can be avoided if we do not calculate $G[i,j]$ when j is odd and $j < i - 1$, since these nodes are never of interest, but there is no "bottom-up" reason for not calculating $G[12,6]$. As usual, things get worse as the instance gets bigger: to solve the game with 248 matches it is sufficient to explore 1000 nodes, yet *dynwin* looks at more than 30 times this number.

The recursive algorithm given previously is inefficient because it recalculates the same value several times. Because of its top-down nature, however, it never calculates an unnecessary value. A solution that combines the advantages of both algorithms consists of using a memory function; see Section 8.8. This involves remembering which nodes have already been visited during the recursive computation using a global Boolean array *known*$[0..n, 0..n]$, where n is an upper bound on the number of matches to be used. The necessary initializations are as follows.

```
G[0,0] ← false; known[0,0] ← true
    for i ← 1 to n do
        for j ← 1 to i do
            known[i, j] ← false
```

Thereafter, to discover whether $\langle i, j \rangle$ is a winning or a losing position, call the following function.

```
function nim(i, j)
    {For each 1 ≤ j ≤ i ≤ n, returns true
     if and only if position ⟨i, j⟩ is winning}
    if known[i, j] then return G[i, j]
    known[i, j] ← true
    for k ← 1 to j do
        if not nim(i − k, min(2k, i − k)) then
            G[i, j] ← true
            return true
    G[i, j] ← false
    return false
```

At first sight there is no particular reason to favour this approach over dynamic programming, because in any case we have to take the time to initialize the whole array $known[0..n, 0..n]$. However, virtual initialization (described in Section 5.1) allows us to avoid this, and to obtain a worthwhile gain in efficiency.

The game we have considered up to now is so simple that it can be solved without using the associated graph; see Problem 9.5. However the same principles apply to many other games of strategy. As before, a node of a directed graph corresponds to a particular position in the game, and an edge corresponds to a legal move between two positions. The graph is infinite if there is no a priori limit on the number of positions possible in the game. For simplicity, we shall suppose that the game is played by two players who move in turn, that the rules are the same for both players (we say the game is *symmetric*), and that chance plays no part in the outcome (the game is *deterministic*). The ideas we present can easily be adapted to more general contexts. We further suppose that no instance of the game can last forever and that no position in the game offers an infinite number of legal moves to the player whose turn it is. In particular, some positions in the game, called the *terminal positions*, offer no legal moves, and hence some nodes in the graph have no successors.

To determine a winning strategy in a game of this kind, we attach to each node of the graph a label chosen from the set *win*, *lose* and *draw*. The label refers to the situation of a player about to move in the corresponding position, assuming neither player will make an error. The labels are assigned systematically as follows. (In the simple example given earlier, no draws were possible, so the label *draw* was never used, and the rules stated there are incomplete.)

1. Label the terminal positions. The labels assigned depend on the game in question. For most games, if you find yourself in a terminal position, then there is no legal move you can make, and you have lost. However this is not always the case. If you cannot move because of a stalemate in chess, for example, the game is a draw. Also many games of the Nim family come in pairs, one where

the player who takes the last match wins, and one (called the *misère* version of the game) where the player who takes the last match loses.

2. A nonterminal position is a winning position if *at least one* of its successors is a losing position, for the player whose turn it is can leave his opponent in this losing position.

3. A nonterminal position is a losing position if *all* its successors are winning positions, for the player whose turn it is cannot avoid leaving his opponent in one of these winning positions.

4. Any other nonterminal position leads to a draw. In this case the successors must include at least one draw, possibly with some winning positions as well. The player whose turn it is can avoid leaving his opponent in a winning position, but cannot force him into a losing position.

Once these labels are assigned, a winning strategy can be read off from the graph.

In principle, this technique applies even to a game as complex as chess. At first sight, the graph associated with chess appears to contain cycles, since if two positions u and v of the pieces differ only by the legal move of a rook, say, the king not being in check, then we can move equally well from u to v and from v to u. However this problem disappears on closer examination. In the variant of Nim used as an example above, a position is defined not just by the number of matches on the table, but also by an invisible item of information giving the number of matches that can be picked up on the next move. Similarly, a position in chess is not defined simply by the position of the pieces. We also need to know whose turn it is to move, which rooks and kings have moved since the beginning of the game (to know if it is legal to castle), and whether some pawn has just moved two squares forward (to know if a capture *en passant* is possible). There are also rules explicitly designed to prevent a game dragging on forever. For example, a game is declared to be a draw after a certain number of moves in which no irreversible action (the movement of a pawn, or a capture) has taken place. Thanks to these and similar rules, there are no cycles in the graph corresponding to chess. However, they force us to include such items as the number of moves since the last irreversible action in the information defining a position.

Adapting the general rules given above, we can label each node as being a winning position for White, a winning position for Black, or a draw. Once constructed, this graph allows us in principle to play a perfect game of chess, that is, to win whenever it is possible, and to lose only when it is inevitable. Unfortunately—or perhaps fortunately for the game of chess—the graph contains so many nodes that it is out of the question to explore it completely, even with the fastest existing computers. The best we can do is to explore the graph near the current position, to see how the situation might develop, just like the novice who reasons, "If I do this, he will reply like that, and then I can do this", and so on. Even this technique is not without its subtleties, however. Should we look at *all* the possibilities offered by the current position, and then, for each of these, *all* the possibilities of reply?

Or should we rather pick a promising line of attack and follow it up for several moves to see where it leads? Different search strategies may lead to quite different results, as we describe shortly.

If we cannot hope to explore the whole graph for the game of chess, then we cannot hope to construct it and store it either. The best we can expect is to construct parts of the graph as we go along, saving them if they are interesting and throwing them away otherwise. Thus throughout this chapter we use the word "graph" in two different ways.

On the one hand, a graph may be a data structure in the storage of a computer. In this case, the nodes are represented by a certain number of bytes, and the edges are represented by pointers. The operations to be carried out are quite concrete: to "mark a node" means to change a bit in storage, to "find a neighbouring node" means to follow a pointer, and so on.

At other times, the graph exists only implicitly, as when we explore the abstract graph corresponding to the game of chess. This graph never really exists in the storage of the machine. Most of the time, all we have is a representation of the current position (that is, of the node we are in the process of visiting, for, as we saw, nodes correspond to positions of the pieces plus some extra information), and possibly representations of a small number of other positions. Of course we also know the rules of the game in question. In this case to "mark a node" means to take any appropriate measures that enable us to recognize a position we have already seen, or to avoid arriving at the same position twice. To "find a neighbouring node" means to change the current position by making a single legal move, for if it is possible to get from one position to another by making a single move, then an edge exists in the implicit graph between the two corresponding nodes.

Exactly similar considerations apply when we explore any large graph, as we shall see particularly in Section 9.6. However, whether the graph is a data structure or merely an abstraction that we can never manipulate as a whole, the techniques used to traverse it are essentially the same. In this chapter we therefore do not distinguish the two cases.

9.2 Traversing trees

We shall not spend long on detailed descriptions of how to explore a tree. We simply remind the reader that in the case of binary trees three techniques are often used. If at each node of the tree we visit first the node itself, then all the nodes in the left subtree, and finally all the nodes in the right subtree, we are traversing the tree in *preorder*; if we visit first the left subtree, then the node itself, and finally the right subtree, we are traversing the tree in *inorder*; and if we visit first the left subtree, then the right subtree, and lastly the node itself, we are traversing the tree in *postorder*. Preorder and postorder generalize in an obvious way to nonbinary trees.

These three techniques explore the tree from left to right. Three corresponding techniques explore the tree from right to left. Implementation of any of these techniques using recursion is straightforward.

Lemma 9.2.1 *For each of the six techniques mentioned, the time $T(n)$ needed to explore a binary tree containing n nodes is in $\Theta(n)$.*

Proof Suppose that visiting a node takes a time in $O(1)$, that is, the time required is bounded above by some constant c. Without loss of generality we may suppose that $c \geq T(0)$. Suppose further that we are to explore a tree containing n nodes, $n > 0$; one of these nodes is the root, so if g of them lie in the left subtree, then there are $n - g - 1$ in the right subtree. Then

$$T(n) \leq \max_{0 \leq g \leq n-1} (T(g) + T(n-g-1) + c), \qquad n > 0.$$

This is true whatever the order in which the left and right subtrees and the root are explored. We prove by constructive induction that $T(n) \leq an + b$, where a and b are appropriate constants, as yet unknown. If we choose $b \geq c$ the hypothesis is true for $n = 0$, because $c \geq T(0)$. For the induction step, let $n > 0$, and suppose the hypothesis is true for all m, $0 \leq m \leq n - 1$. Then

$$\begin{aligned} T(n) &\leq \max_{0 \leq g \leq n-1} (T(g) + T(n-g-1) + c) \\ &\leq \max_{0 \leq g \leq n-1} (ag + b + a(n-g-1) + b + c) \\ &\leq an + 3b - a. \end{aligned}$$

Hence provided we choose $a \geq 2b$ we have $T(n) \leq an + b$, so the hypothesis is also true for $m = n$. This proves that $T(n) \leq an + b$ for every $n \geq 0$, and therefore $T(n)$ is in $O(n)$.

On the other hand it is clear that $T(n)$ is in $\Omega(n)$ since each of the n nodes is visited. Therefore $T(n)$ is in $\Theta(n)$. ■

9.2.1 Preconditioning

If we have to solve several similar instances of the same problem, it may be worthwhile to invest some time in calculating auxiliary results that can thereafter be used to speed up the solution of each instance. This is *preconditioning*. Informally, let a be the time it takes to solve a typical instance when no auxiliary information is available, let b be the time it takes to solve a typical instance when auxiliary information *is* available, and let p be the time needed to calculate this extra information. To solve n typical instances takes time na without preconditioning and time $p + nb$ if preconditioning is used. Provided $b < a$, it is advantageous to use preconditioning when $n > p/(a - b)$.

From this point of view, the dynamic programming algorithm for making change given in Section 8.2 may be seen as an example of preconditioning. Once the necessary values c_{nj} have been calculated, we can make change quickly whenever this is required.

Even when there are few instances to be solved, precomputation of auxiliary information may be useful. Suppose we occasionally have to solve one instance

from some large set of possible instances. When a solution is needed it must be provided very rapidly, for example to ensure sufficiently fast response for a real-time application. In this case it may well be impractical to calculate ahead of time and to store the solutions to all the relevant instances. On the other hand, it may be possible to calculate and store sufficient auxiliary information to speed up the solution of whatever instance comes along. Such an application of preconditioning may be of practical importance even if only one crucial instance is solved in the whole lifetime of the system: this may be just the instance that enables us, for example, to stop a runaway reactor.

As a second example of this technique we use the problem of determining ancestry in a rooted tree. Let T be a rooted tree, not necessarily binary. We say that a node v of T is an *ancestor* of node w if v lies on the path from w to the root of T. In particular, every node is its own ancestor, and the root is an ancestor of every node. (Those with a taste for recursive definitions may prefer the following: every node is its own ancestor, and, recursively, it is an ancestor of all the nodes of which its children are ancestors.) The problem is thus, given a pair of nodes (v, w) from T, to determine whether or not v is an ancestor of w. If T contains n nodes, any direct solution of this instance takes a time in $\Omega(n)$ in the worst case. However it is possible to precondition T in a time in $\Theta(n)$ so we can subsequently solve any particular instance of the ancestry problem in constant time.

We illustrate this using the tree in Figure 9.2. It contains 13 nodes. To precondition the tree, we traverse it first in preorder and then in postorder, numbering the nodes sequentially as we visit them. For a node v, let $prenum[v]$ be the number assigned to v when we traverse the tree in preorder, and let $postnum[v]$ be the number assigned during the traversal in postorder. In Figure 9.2 these numbers appear to the left and the right of the node, respectively.

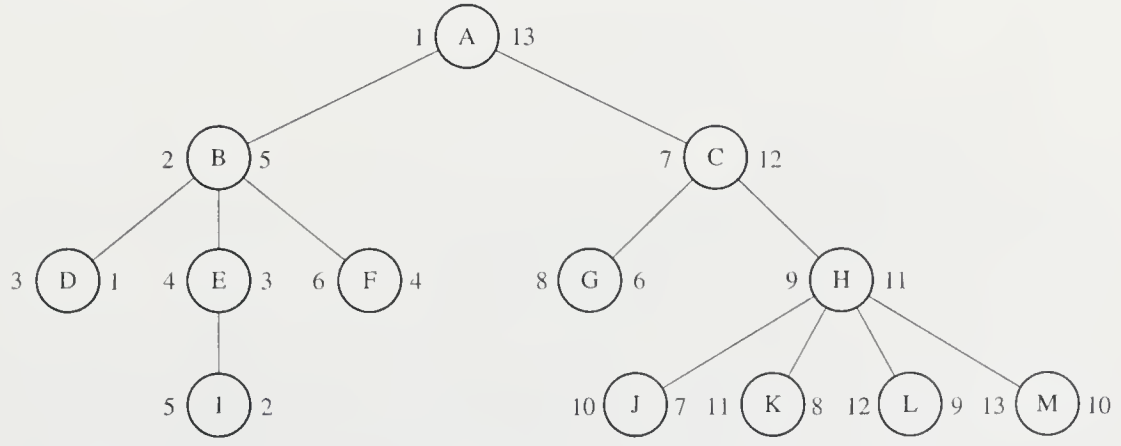

Figure 9.2. A rooted tree with preorder and postorder numberings

Let v and w be two nodes in the tree. In preorder we first number a node and then number its subtrees from left to right. Thus

$$prenum[v] \leq prenum[w] \iff \begin{array}{l} v \text{ is an ancestor of } w \textbf{ or} \\ v \text{ is to the left of } w \text{ in the tree.} \end{array}$$

In postorder we first number the subtrees of a node from left to right, and then we number the node itself. Thus

$$postnum[v] \geq postnum[w] \iff \begin{array}{l} v \text{ is an ancestor of } w \textbf{ or} \\ v \text{ is to the right of } w \text{ in the tree.} \end{array}$$

It follows that

$$\begin{array}{c} prenum[v] \leq prenum[w] \textbf{ and } postnum[v] \geq postnum[w] \\ \iff v \text{ is an ancestor of } w. \end{array}$$

Once the values of *prenum* and *postnum* have been calculated in a time in $\Theta(n)$, the required condition can be checked in a time in $\Theta(1)$.

9.3 Depth-first search: Undirected graphs

Let $G = \langle N, A \rangle$ be an undirected graph all of whose nodes we wish to visit. Suppose it is somehow possible to mark a node to show it has already been visited.

To carry out a *depth-first* traversal of the graph, choose any node $v \in N$ as the starting point. Mark this node to show it has been visited. Next, if there is a node adjacent to v that has not yet been visited, choose this node as a new starting point and call the depth-first search procedure recursively. On return from the recursive call, if there is another node adjacent to v that has not been visited, choose this node as the next starting point, call the procedure recursively once again, and so on. When all the nodes adjacent to v are marked, the search starting at v is finished. If there remain any nodes of G that have not been visited, choose any one of them as a new starting point, and call the procedure yet again. Continue thus until all the nodes of G are marked. Here is the recursive algorithm.

```
procedure dfsearch(G)
    for each v ∈ N do mark[v] ← not-visited
    for each v ∈ N do
        if mark[v] ≠ visited then dfs(v)

procedure dfs(v)
    {Node v has not previously been visited}
    mark[v] ← visited
    for each node w adjacent to v do
        if mark[w] ≠ visited then dfs(w)
```

The algorithm is called depth-first search because it initiates as many recursive calls as possible before it ever returns from a call. The recursion stops only when exploration of the graph is blocked and can go no further. At this point the recursion "unwinds" so alternative possibilities at higher levels can be explored. If the graph corresponds to a game, this may be thought of intuitively as a search that explores the result of one particular strategy as many moves ahead as possible before looking around to see what alternative tactics might be available.

Consider for example the graph in Figure 9.3. If we suppose that the neighbours of a given node are examined in numerical order, and that node 1 is the first starting point, depth-first search of the graph progresses as follows:

1. *dfs*(1) initial call
2. *dfs*(2) recursive call
3. *dfs*(3) recursive call
4. *dfs*(6) recursive call
5. *dfs*(5) recursive call; progress is blocked
6. *dfs*(4) a neighbour of node 1 has not been visited
7. *dfs*(7) recursive call
8. *dfs*(8) recursive call; progress is blocked
9. there are no more nodes to visit

How much time is needed to explore a graph with n nodes and a edges? Since each node is visited exactly once, there are n calls of the procedure *dfs*. When we visit a node, we look at the mark on each of its neighbouring nodes. If the graph is represented so as to make the lists of adjacent nodes directly accessible (type *lisgraph* of Section 5.4), this work is proportional to a in total. The algorithm therefore takes a time in $\Theta(n)$ for the procedure calls and a time in $\Theta(a)$ to inspect the marks. The execution time is thus in $\Theta(\max(a, n))$.

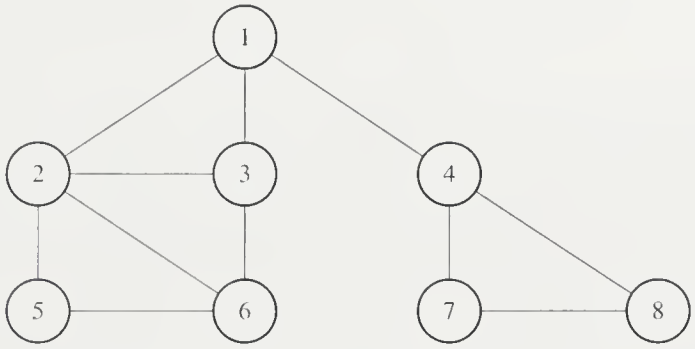

Figure 9.3. An undirected graph

Depth-first traversal of a connected graph associates a spanning tree to the graph. Call this tree T. The edges of T correspond to the edges used to traverse the graph; they are directed from the first node visited to the second. Edges not used in the traversal of the graph have no corresponding edge in T. The initial starting point of the exploration becomes the root of the tree. For example, the edges used in the depth-first search of the graph in Figure 9.3 described above are $\{1,2\}$, $\{2,3\}$, $\{3,6\}$, $\{6,5\}$, $\{1,4\}$, $\{4,7\}$ and $\{7,8\}$. The corresponding directed edges, $(1,2)$, $(2,3)$, and so on, form a spanning tree for this graph. The root of the tree is node 1. The tree is illustrated in Figure 9.4. The broken lines in this figure correspond to edges of G not used in the depth-first search. It is easy to show that an edge of G with no corresponding edge in T necessarily joins some node v to one of its ancestors in T; see Problem 9.17.

If the graph being explored is not connected, a depth-first search associates to it not merely a single tree, but rather a forest of trees, one for each connected component of the graph. A depth-first search also provides a way to number the

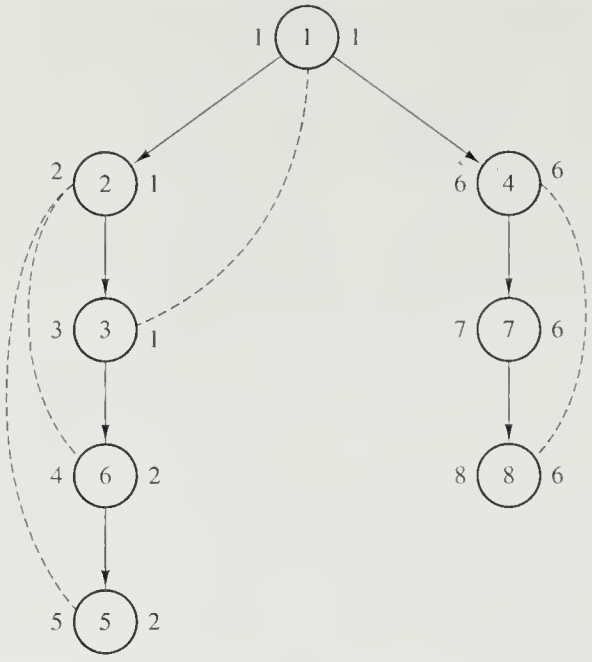

Figure 9.4. A depth-first search tree; *prenum* on the left, *highest* on the right

nodes of the graph being visited. The first node visited—the root of the tree—is numbered 1, the second is numbered 2, and so on. In other words, the nodes of the associated tree are numbered in preorder. To implement this, add the following two statements at the beginning of the procedure *dfs*:

$$pnum \leftarrow pnum + 1$$
$$prenum[v] \leftarrow pnum$$

where *pnum* is a global variable initialized to zero. For example, the depth-first search of the graph in Figure 9.3 described above numbers the nodes of the graph as follows.

node	1	2	3	4	5	6	7	8
prenum	1	2	3	6	5	4	7	8

These are the numbers to the left of each node in Figure 9.4. Of course the tree and the numbering generated by a depth-first search in a graph are not unique, but depend on the chosen starting point and on the order in which neighbours are visited.

9.3.1 Articulation points

A node v of a connected graph is an *articulation point* if the subgraph obtained by deleting v and all the edges incident on v is no longer connected. For example, node 1 is an articulation point of the graph in Figure 9.3; if we delete it, there remain two connected components $\{2,3,5,6\}$ and $\{4,7,8\}$. A graph G is *biconnected* (or *unarticulated*) if it is connected and has no articulation points. It is *bicoherent* (or *isthmus-free*, or *2-edge-connected*) if each articulation point is joined by at least two

edges to each component of the remaining subgraph. These ideas are important in practice. If the graph G represents, say, a telecommunications network, then the fact that it is biconnected assures us that the rest of the network can continue to function even if the equipment in one of the nodes fails. If G is bicoherent, we can be sure the nodes will be able to communicate with one another even if one transmission line stops working.

To see how to find the articulation points of a connected graph G, look again at Figure 9.4. Remember that this figure shows *all* the edges of the graph G of Figure 9.3: those shown as solid lines form a spanning tree T, while the others are shown as broken lines. As we saw, these other edges only go from some node v to its ancestor in the tree, not from one branch to another. To the left of each node v is $preuum[v]$, the number assigned by a preorder traversal of T.

To the right of each node v is a new number that we shall call $highest[v]$. Let w be the highest node in the tree that can be reached from v by following down zero or more solid lines, and then going up at most one broken line. Then define $highest[v]$ to be $preuum[w]$. For instance, from node 7 we can go down one solid line to node 8, then up one broken line to node 4, and this is the highest node we can reach. Since $preuum[4] = 6$, we also have $highest[7] = 6$.

Because the broken lines do not cross from one branch to another, the highest node w reachable in this way must be an ancestor of v. (It cannot lie below v in the tree, because we can always get to v itself by not following any lines at all.) Among the ancestors of v, the highest up the tree is the one with the lowest value of $preuum$. If we have these values, it is therefore not necessary to know the exact level of each node: among the nodes we can reach, we simply choose the one that minimizes $preuum$.

Now consider any node v in T except the root. If v has no children, it cannot be an articulation point of G, for if we delete it the remaining nodes are still connected by the edges left in T. Otherwise, let x be a child of v. Suppose first that $highest[x] < preuum[v]$. This means that from x there is a chain of edges of G, not including the edge $\{v, x\}$ (for we were not allowed to go *up* a solid line), that leads to some node higher up the tree than v. If we delete v, therefore, the nodes in the subtree rooted at x will not be disconnected from the rest of the tree. This is the case with node 3 in the figure, for example. Here $preuum[3] = 3$, and the only child of node 3, namely node 6, has $highest[6] = 2 < preuum[3]$. Therefore if we delete node 3, node 6 and its descendants will still be attached to one of the ancestors of node 3.

If on the other hand $highest[x] \geq preuum[v]$, then no chain of edges from x (again excluding the edge $\{v, x\}$) rejoins the tree higher than v. In this case, should v be deleted, the nodes in the subtree rooted at x will be disconnected from the rest of the tree. Node 4 in the figure illustrates this case. Here $preuum[4] = 6$, and the only child of node 4, namely node 7, has $highest[7] = 6 = preuum[4]$. Therefore if we delete node 4, no path from node 7 or from one of its descendants leads back above the deleted node, so the subtree rooted at node 7 will be detached from the rest of T.

Thus a node v that is not the root of T is an articulation point of G if and only if it has at least one child x with $highest[x] \geq preuum[v]$. As for the root, it is evident

that it is an articulation point of G if and only if it has more than one child; for in this case, since no edges cross from one branch to another, deleting the root disconnects the remaining subtrees of T.

It remains to be seen how to calculate the values of *highest*. Clearly this must be done from the leaves upwards. For example, from node 5 we can stay where we are, or go up to node 2; these are the only possibilities. From node 6 we can stay where we are, go up to node 2, or else go first down to node 5 and then to wherever is reachable from there; and so on. The values of *highest* are therefore calculated in postorder. At a general node v, *highest*$[v]$ is the minimum (corresponding to the highest node) of three kinds of values: *prenum*$[v]$ (we stay where we are), *prenum*$[w]$ for each node w such that there is an edge $\{v, w\}$ in G with no corresponding edge in T (we go up a broken line), and *highest*$[x]$ for every child x of v (we go down a solid line and see where we can get from there).

The complete algorithm for finding the articulation points of an undirected graph G is summarized as follows.

1. Carry out a depth-first search in G, starting from any node. Let T be the tree generated by this search, and for each node v of G, let *prenum*$[v]$ be the number assigned by the search.
2. Traverse T in postorder. For each node v visited, calculate *highest*$[v]$ as the minimum of
   (a) *prenum*$[v]$;
   (b) *prenum*$[w]$ for each node w such that there is an edge $\{v, w\}$ in G with no corresponding edge in T; and
   (c) *highest*$[x]$ for every child x of v.
3. Determine the articulation points of G as follows.
   (a) The root of T is an articulation point if and only if it has more than one child.
   (b) Any other node v is an articulation point if and only if it has a child x such that *highest*$[x] \geq$ *prenum*$[v]$.

It is not difficult to combine steps 1 and 2 of the above algorithm, calculating the values of both *prenum* and *highest* during the depth-first search of G.

9.4 Depth-first search: Directed graphs

The algorithm is essentially the same as for undirected graphs, the difference residing in the interpretation of the word "adjacent". In a directed graph, node w is adjacent to node v if the directed edge (v, w) exists. If (v, w) exists but (w, v) does not, then w is adjacent to v but v is not adjacent to w. With this change of interpretation the procedures *dfs* and *search* from Section 9.3 apply equally well in the case of a directed graph.

The algorithm behaves quite differently, however. Consider a depth-first search of the directed graph in Figure 9.5. If the neighbours of a given node are examined in numerical order, the algorithm progresses as follows:

1. *dfs*(1) initial call
2. *dfs*(2) recursive call
3. *dfs*(3) recursive call; progress is blocked
4. *dfs*(4) a neighbour of node 1 has not been visited
5. *dfs*(8) recursive call
6. *dfs*(7) recursive call; progress is blocked
7. *dfs*(5) new starting point
8. *dfs*(6) recursive call; progress is blocked
9. there are no more nodes to visit

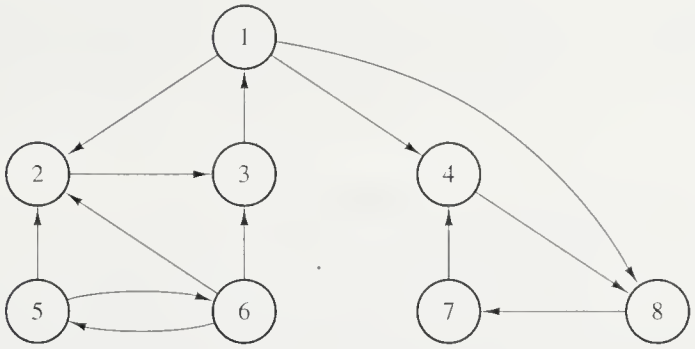

Figure 9.5. A directed graph

An argument identical with the one in Section 9.3 shows that the time taken by this algorithm is also in $\Theta(\max(a, n))$. In this case, however, the edges used to visit all the nodes of a directed graph $G = \langle N, A \rangle$ may form a forest of several trees even if G is connected. This happens in our example: the edges used, namely $(1,2)$, $(2,3)$, $(1,4)$, $(4,8)$, $(8,7)$ and $(5,6)$, form the forest shown by the solid lines in Figure 9.6.

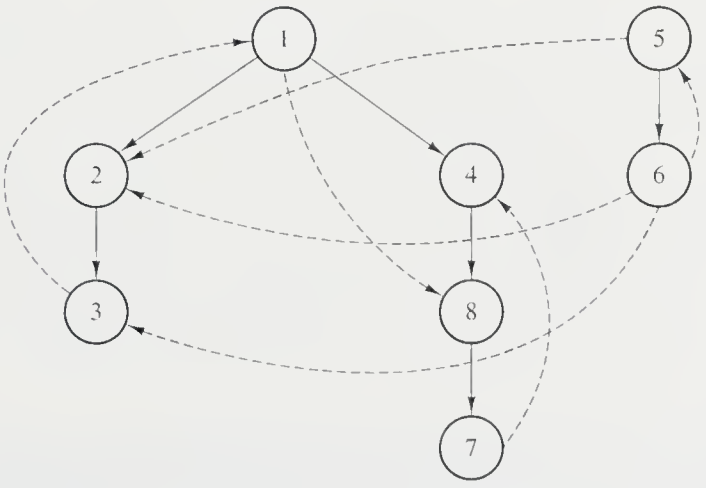

Figure 9.6. A depth-first search forest

Let F be the set of edges in the forest, so that $A \backslash F$ is the set of edges of G that have no corresponding edge in the forest. In the case of an undirected graph, we saw that the edges of the graph with no corresponding edge in the forest necessarily join some node to one of its ancestors. In the case of a directed graph, however, three kinds of edge can appear in $A \backslash F$. These are shown by the broken lines in Figure 9.6.

1. Those like $(3,1)$ or $(7,4)$ lead from a node to one of its ancestors.
2. Those like $(1,8)$ lead from a node to one of its descendants.
3. Those like $(5,2)$ or $(6,3)$ join one node to another that is neither its ancestor nor its descendant. Edges of this type are necessarily directed from right to left.

9.4.1 Acyclic graphs: Topological sorting

Directed acyclic graphs can be used to represent a number of interesting relations. This class includes trees, but is less general than the class of all directed graphs. For example, a directed acyclic graph can be used to represent the structure of an arithmetic expression involving repeated subexpressions: thus Figure 9.7 represents the structure of the expression

$$(a+b)(c+d)+(a+b)(c-d).$$

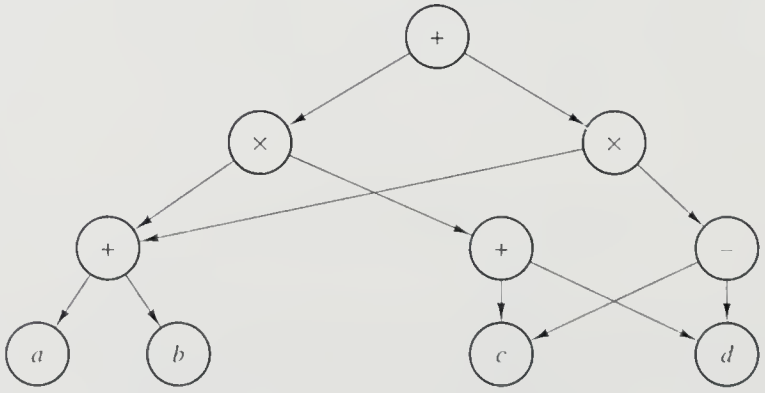

Figure 9.7. A directed acyclic graph

These graphs also offer a natural representation for partial orderings, such as the relation of set-inclusion. Figure 9.8 illustrates part of another partial ordering defined on the positive integers: here there is an edge from node i to node j if and only if i is a proper divisor of j.

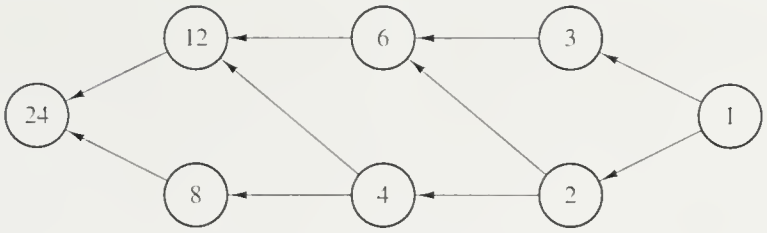

Figure 9.8. Another directed acyclic graph

Finally, directed acyclic graphs are often used to specify how a complex project develops over time. The nodes represent different stages of the project, from the initial state to final completion, and the edges correspond to activities that must be completed to pass from one stage to another. Figure 9.9 gives an example of this type of diagram.

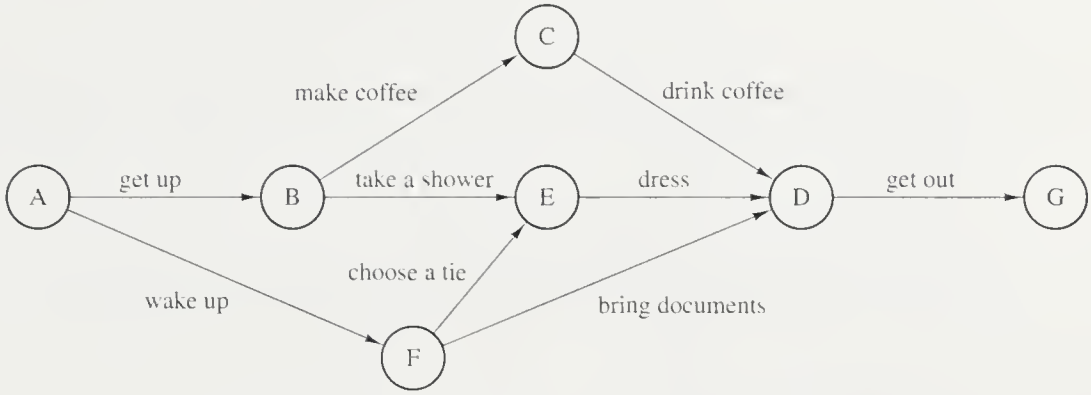

Figure 9.9. Yet another directed acyclic graph

Depth-first search can be used to detect whether a given directed graph is acyclic; see Problem 9.26. It can also be used to determine a *topological ordering* of the nodes of a directed acyclic graph. In this kind of ordering the nodes of the graph are listed in such a way that if there exists an edge (i, j), then node i precedes node j in the list. For example, for the graph of Figure 9.8, the natural order $1, 2, 3, 4, 6, 8, 12, 24$ is adequate; but the order $1, 3, 2, 6, 4, 12, 8, 24$ is also acceptable, as are several others. On the other hand, the order $1, 3, 6, 2, 4, 12, 8, 24$ will not do, because the graph includes an edge $(2, 6)$, and so 2 must precede 6 in the list.

Adapting the procedure *dfs* to make it into a topological sort is simple. Add an extra line

write v

at the end of the procedure *dfs*, run the procedure on the graph concerned, and then reverse the order of the resulting list of nodes.

To see this, consider what happens when we arrive at node v of a directed acyclic graph G using the modified procedure. Some nodes that must follow v in

topological order may already have been visited while following a different path. In this case they are already on the list, as they should be since we reverse the list when the search is finished. Any node that must precede v in topological order either lies along the path we are currently exploring, in which case it is marked as visited but is not yet on the list, or else it has not yet been visited. In either case it will be added to the list after node v (again, correctly, for the list is to be reversed). Now the depth-first search explores the unvisited nodes that can be reached from v by following edges of G. In the topological ordering, these must come after v. Since we intend to reverse the list when the search is finished, adding them to the list during the exploration starting at v, and adding v only when this exploration is finished, gives us exactly what we want.

9.5 Breadth-first search

When a depth-first search arrives at some node v, it next tries to visit some neighbour of v, then a neighbour of this neighbour, and so on. When a breadth-first search arrives at some node v, on the other hand, it first visits all the neighbours of v. Only when this has been done does it look at nodes further away. Unlike depth-first search, breadth-first search is not naturally recursive. To underline the similarities and the differences between the two methods, we begin by giving a nonrecursive formulation of the depth-first search algorithm. Let *stack* be a data type allowing two operations, **push** and **pop**. The type is intended to represent a list of elements to be handled in the order "last come, first served" (often referred to as LIFO, standing for "last in, first out"). The function *top* denotes the element at the top of the stack. Here is the modified depth-first search algorithm.

```
procedure dfs2(v)
    P ← empty-stack
    mark[v] ← visited
    push v onto P
    while P is not empty do
        while there exists a node w adjacent to top(P)
                such that mark[w] ≠ visited
                do mark[w] ← visited
                    push w onto P {w is the new top(P)}
        pop P
```

Modifying the algorithm has not changed its behaviour. All we have done is to make explicit the stacking and unstacking of nodes that in the previous version was handled behind the scenes by the stack mechanism implicit in any recursive language.

For the breadth-first search algorithm, by contrast, we need a type *queue* that allows two operations **enqueue** and **dequeue**. This type represents a list of elements to be handled in the order "first come, first served" (or FIFO, for "first in, first out"). The function *first* denotes the element at the front of the queue. Here now is the breadth-first search algorithm.

```
procedure bfs(v)
    Q ← empty-queue
    mark[v] ← visited
    enqueue v into Q
    while Q is not empty do
        u ← first(Q)
        dequeue u from Q
        for each node w adjacent to u do
            if mark[w] ≠ visited then mark[w] ← visited
              enqueue w into Q
```

In both cases we need a main program to start the search.

```
procedure search(G)
    for each v ∈ N do mark[v] ← not-visited
    for each v ∈ N do
        if mark[v] ≠ visited then {dfs2 or bfs} (v)
```

For example, on the graph of Figure 9.3, if the search starts at node 1, and the neighbours of a node are visited in numerical order, breadth-first search proceeds as follows.

	Node visited	Q
1.	1	2,3,4
2.	2	3,4,5,6
3.	3	4,5,6
4.	4	5,6,7,8
5.	5	6,7,8
6.	6	7,8
7.	7	8
8.	8	—

As for depth-first search, we can associate a tree with a breadth-first search. Figure 9.10 shows the tree generated by the search above. The edges of the graph with no corresponding edge in the tree are represented by broken lines; see Problem 9.30. In general, if the graph G being searched is not connected, the search generates a forest of trees, one for each connected component of G.

It is easy to show that the time required by a breadth-first search is in the same order as that required by a depth-first search, namely $\Theta(\max(a, n))$. If the appropriate interpretation of the word "adjacent" is used, the breadth-first search algorithm—again, exactly like the depth-first search algorithm—can be applied without modification to either directed or undirected graphs; see Problems 9.31 and 9.32.

Breadth-first search is most often used to carry out a partial exploration of an infinite (or unmanageably large) graph, or to find the shortest path from one point to another in a graph. Consider for example the following problem. The value 1 is given. To construct other values, two operations are available: multiplication by 2 and division by 3. For the second operation, the operand must be greater than 2

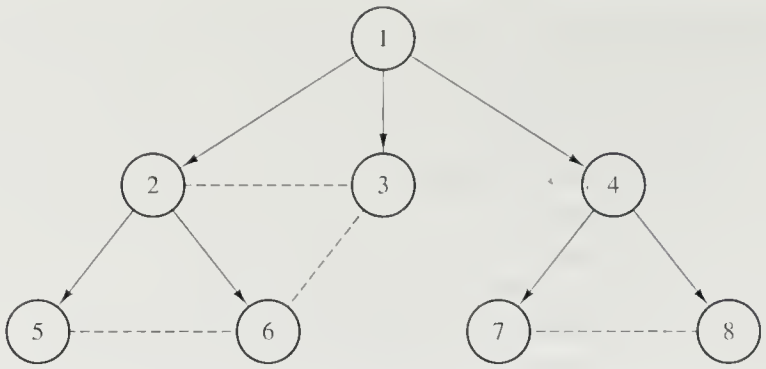

Figure 9.10. A breadth-first search tree

(so we cannot reach 0), and any resulting fraction is dropped. If operations are executed from left to right, we may for instance obtain the value 10 as

$$10 = 1 \times 2 \times 2 \times 2 \times 2 \div 3 \times 2.$$

We want to obtain some specified value n. How should we set about it?

The problem can be represented as a search in the infinite directed graph of Figure 9.11. Here the given value 1 is in the node at top left. Thereafter each node is linked to the values that can be obtained using the two available operations. For example, from the value 16, we can obtain the two new values 32 (by multiplying 16 by 2) and 5 (by dividing 16 by 3, dropping the resulting fraction). For clarity, we have omitted links backwards to values already available, for instance from 8 to 2. These backwards links are nevertheless present in the real graph. The graph is infinite, for a sequence such as $1, 2, \ldots, 256, 512, \ldots$ can be continued indefinitely. It is not a tree, because node 42, for instance, can be reached from both 128 and 21. When the backwards links are included, it is not even acyclic.

To solve a given instance of the problem, that is, to find how to construct a particular value n, we search the graph starting at 1 until we find the value we are looking for. On this infinite graph, however, a depth-first search may not work. Suppose for example that $n = 13$. If we explore the neighbours of a node in the order "first multiplication by 2, then division by 3", a depth-first search visits successively nodes $1, 2, 4, \ldots$, and so on, heading off along the top branch and (since there is always a new neighbour to look at) never encountering a situation that forces it to back up to a previous node. In this case the search certainly fails. If on the other hand we explore the neighbours of a node in the order "first division by 3, then multiplication by 2", the search first runs down to node 12; from there it moves successively to nodes 24, 48, 96, 32, and 64, and from 64 it wanders off into the upper right-hand part of the graph. We may be lucky enough to get back to node 13 and thus find a solution to our problem, but nothing guarantees this. Even if we do, the solution found will be more complex than necessary. If you program this depth-first search on a computer, you will find in fact that you reach the given value 13 after 74 multiplication and division operations.

A breadth-first search, on the other hand, is sure to find a solution to the instance if there is one. If we examine neighbours in the order "first multiplication by 2,

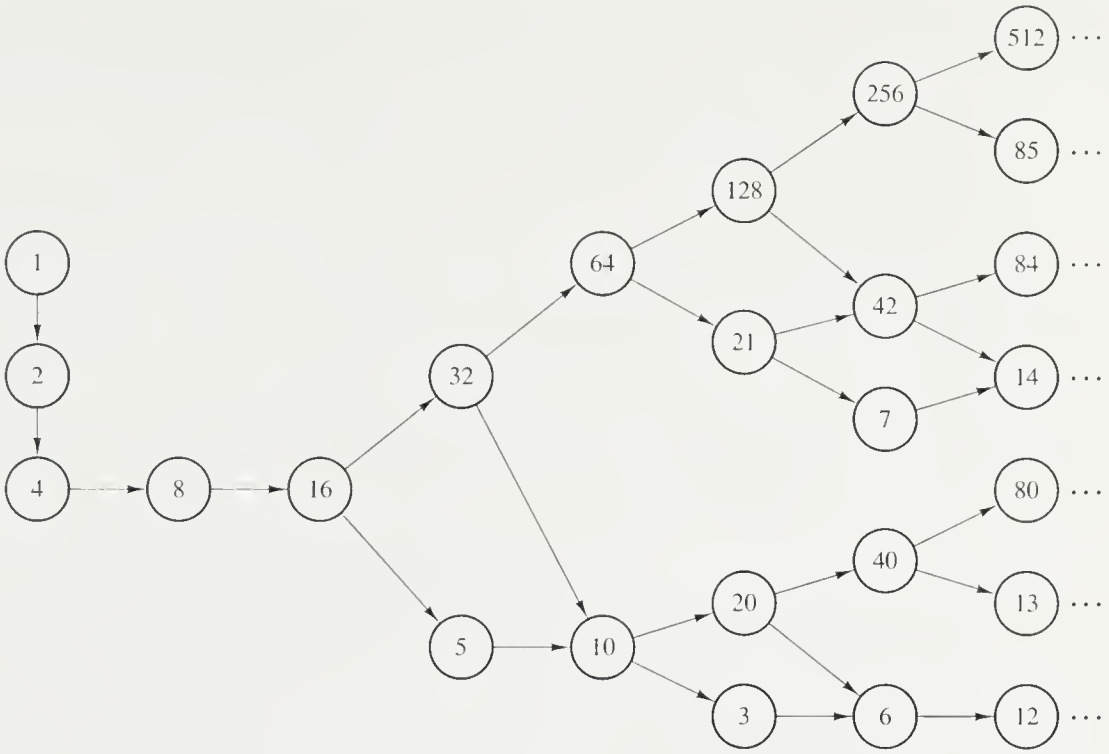

Figure 9.11. Multiplication by 2 and division by 3

then division by 3", a breadth-first search starting at 1 visits successively nodes $1, 2, \ldots, 16, 32, 5, 64, 10, 128, 21, 20, 3$, and so on. Not only are we sure to find the value we seek if it is in the graph, but also the solution obtained will use the smallest number of operations possible. In other words, the path found from 1 to the desired value n will be as short as possible. The same is true if we carry out the search looking at neighbours in the order "first division by 3, then multiplication by 2". Using either of these breadth-first searches, it is easy (even by hand) to discover several ways to produce the value 13 using just 9 operations. For example

$$13 = 1 \times 2 \times 2 \times 2 \times 2 \div 3 \times 2 \times 2 \times 2 \div 3.$$

Of course even a breadth-first search may fail. In our example, it may be that some values n are not present in the graph at all. (Having no idea whether this is true or not, we leave the question as an exercise for the reader.) In this case any search technique is certain to fail for the missing values. If a graph includes one or more nodes with an infinite number of neighbours, but no paths of infinite length, depth-first search may succeed where breadth-first search fails. Nevertheless this situation seems to be less common than its opposite.

9.6 Backtracking

As we saw earlier, various problems can be thought of in terms of abstract graphs. For example, we saw in Section 9.1 that we can use the nodes of a graph to represent positions in a game of chess, and edges to represent legal moves. Often the original

problem translates to searching for a specific node, path or pattern in the associated graph. If the graph contains a large number of nodes, and particularly if it is infinite, it may be wasteful or infeasible to build it explicitly in computer storage before applying one of the search techniques we have encountered so far.

In such a situation we use an *implicit graph*. This is one for which we have available a description of its nodes and edges, so relevant parts of the graph can be built as the search progresses. In this way computing time is saved whenever the search succeeds before the entire graph has been constructed. The economy in storage space can also be dramatic, especially when nodes that have already been searched can be discarded, making room for subsequent nodes to be explored. If the graph involved is infinite, such a technique offers our only hope of exploring it at all. This section and the ones that follow detail some standard ways of organizing searches in an implicit graph.

In its basic form, backtracking resembles a depth-first search in a directed graph. The graph concerned is usually a tree, or at least it contains no cycles. Whatever its structure, the graph exists only implicitly. The aim of the search is to find solutions to some problem. We do this by building partial solutions as the search proceeds; such partial solutions limit the regions in which a complete solution may be found. Generally speaking, when the search begins, nothing is known about the solutions to the problem. Each move along an edge of the implicit graph corresponds to adding a new element to a partial solution, that is, to narrowing down the remaining possibilities for a complete solution. The search is successful if, proceeding in this way, a solution can be completely defined. In this case the algorithm may either stop (if only one solution to the problem is needed), or continue looking for alternative solutions (if we want to look at them all). On the other hand, the search is unsuccessful if at some stage the partial solution constructed so far cannot be completed. In this case the search backs up, exactly like a depth-first search, removing as it goes the elements that were added at each stage. When it gets back to a node with one or more unexplored neighbours, the search for a solution resumes.

9.6.1 The knapsack problem (3)

For a first example illustrating the general principle, we return to the knapsack problem described in Section 8.4. Recall that we are given a certain number of objects and a knapsack. This time, however, instead of supposing that we have n objects available, we shall suppose that we have n *types* of object, and that an adequate number of objects of each type are available. This does not alter the problem in any important way. For $i = 1, 2, \ldots, n$, an object of type i has a positive weight w_i and a positive value v_i. The knapsack can carry a weight not exceeding W. Our aim is to fill the knapsack in a way that maximizes the value of the included objects, while respecting the capacity constraint. We may take an object or to leave it behind, but we may not take a fraction of an object.

Suppose for concreteness that we wish to solve an instance of the problem involving four types of objects, whose weights are respectively 2, 3, 4 and 5 units, and whose values are 3, 5, 6 and 10. The knapsack can carry a maximum of 8

units of weight. This can be done using backtracking by exploring the implicit tree shown in Figure 9.12.

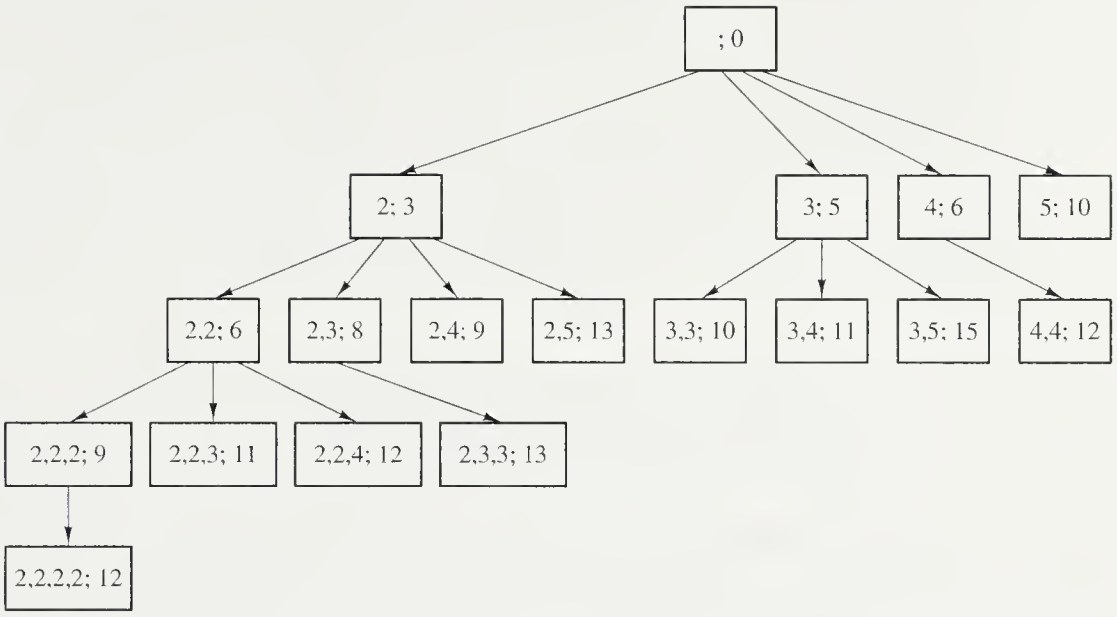

Figure 9.12. The implicit tree for a knapsack problem

Here a node such as $(2,3;8)$ corresponds to a partial solution of our problem. The figures to the left of the semicolon are the weights of the objects we have decided to include, and the figure to the right is the current value of the load. Moving down from a node to one of its children corresponds to deciding which kind of object to put into the knapsack next. Without loss of generality we may agree to load objects into the knapsack in order of increasing weight. This is not essential, and indeed any other order—by decreasing weight, for example, or by value—would work just as well, but it reduces the size of the tree to be searched. For instance, once we have visited node $(2,2,3;11)$ there is no point in later visiting $(2,3,2;11)$.

Initially the partial solution is empty. The backtracking algorithm explores the tree as in a depth-first search, constructing nodes and partial solutions as it goes. In the example, the first node visited is $(2;3)$, the next is $(2,2;6)$, the third is $(2,2,2;9)$ and the fourth $(2,2,2,2;12)$. As each new node is visited, the partial solution is extended. After visiting these four nodes, the depth-first search is blocked: node $(2,2,2,2;12)$ has no unvisited successors (indeed no successors at all), since adding more items to this partial solution would violate the capacity constraint. Since this partial solution may turn out to be the optimal solution to our instance, we memorize it.

The depth-first search now backs up to look for other solutions. At each step back up the tree, the corresponding item is removed from the partial solution. In the example, the search first backs up to $(2,2,2;9)$, which also has no unvisited successors; one step further up the tree, however, at node $(2,2;6)$, two successors

remain to be visited. After exploring nodes $(2,2,3;11)$ and $(2,2,4;12)$, neither of which improves on the solution previously memorized, the search backs up one stage further, and so on. Exploring the tree in this way, $(2,3,3;13)$ is found to be a better solution than the one we have, and later $(3,5;15)$ is found to be better still. Since no other improvement is made before the search ends, this is the optimal solution to the instance.

Programming the algorithm is straightforward, and illustrates the close relation between recursion and depth-first search. Suppose the values of n and W, and of the arrays $w[1..n]$ and $v[1..n]$ for the instance to be solved are available as global variables. The ordering of the types of item is unimportant. Define a function *backpack* as follows.

```
function backpack(i, r)
    {Calculates the value of the best load that can
     be constructed using items of types i to n
     and whose total weight does not exceed r}
    b ← 0
    {Try each allowed kind of item in turn}
    for k ← i to n do
        if w[k] ≤ r then
            b ← max(b, v[k]+backpack(k, r − w[k]))
    return b
```

Now to find the value of the best load, call $backpack(1, W)$. Here each recursive call of *backpack* corresponds to extending the depth-first search one level down the tree, while the **for** loop takes care of examining all the possibilities at a given level. In this version of the program, the composition of the load being examined is given implicitly by the values of k saved on the recursive stack. It is not hard to adapt the program so that it gives the composition of the best load explicitly along with its value; see Problem 9.42.

9.6.2 The eight queens problem

For our second example of backtracking, consider the classic problem of placing eight queens on a chessboard in such a way that none of them threatens any of the others. Recall that a queen threatens the squares in the same row, in the same column, or on the same diagonals.

The most obvious way to solve this problem consists of trying systematically all the ways of placing eight queens on a chessboard, checking each time to see whether a solution has been obtained. This approach is of no practical use, even with a computer, since the number of positions we would have to check is $\binom{64}{8} = 4\,426\,165\,368$. The first improvement we might try consists of never putting more than one queen on any given row. This reduces the computer representation of the chessboard to a vector of eight elements, each giving the position of the queen in the corresponding row. For instance, the vector $(3,1,6,2,8,6,4,7)$ represents the position where the queen on row 1 is in column 3, the queen on row 2 is in column 1, and so on. This particular position is not a solution to our problem since the queens in

rows 3 and 6 are in the same column, and also two pairs of queens lie on the same diagonal. Using this representation, we can write an algorithm using eight nested loops. (For the function *solution*, see Problem 9.43.)

```
program queens1
    for i1 ← 1 to 8 do
        for i2 ← 1 to 8 do
                ...
            for i8 ← 1 to 8 do
                sol ← [i1, i2, ..., i8]
                if solution(sol) then write sol
                                      stop
    write "there is no solution"
```

The number of positions to be considered is reduced to $8^8 = 16\,777\,216$, although in fact the algorithm finds a solution and stops after considering only 1 299 852 positions.

Representing the chessboard by a vector prevents us ever trying to put two queens in the same row. Once we have realized this, it is natural to be equally systematic in our use of the columns. Hence we now represent the board by a vector of eight *different* numbers between 1 and 8, that is, by a permutation of the first eight integers. This yields the following algorithm.

```
program queens2
    sol ← initial-permutation
    while sol ≠ final-permutation and not solution(sol) do
        sol ← next-permutation
    if solution(sol) then write sol
                     else write "there is no solution"
```

There are several natural ways to generate systematically all the permutations of the first n integers. For instance, we can put each value in turn in the leading position and generate recursively, for each leading value, all the permutations of the remaining $n - 1$ elements. The following procedure shows how to do this. Here $T[1..n]$ is a global array initialized to $[1, 2, \ldots, n]$, and the initial call of the procedure is $perm(1)$. This way of generating permutations is itself a kind of backtracking.

```
procedure perm(i)
    if i = n then use(T) {T is a new permutation}
             else for j ← i to n do exchange T[i] and T[j]
                                    perm(i + 1)
                                    exchange T[i] and T[j]
```

This approach reduces the number of possible positions to $8! = 40\,320$. If the preceding algorithm is used to generate the permutations, only 2830 positions are in fact considered before the algorithm finds a solution. It is more complicated to generate permutations rather than all the possible vectors of eight integers between 1

and 8. On the other hand, it is easier to verify in this case whether a given position is a solution. Since we already know that two queens can neither be in the same row nor in the same column, it suffices to verify that they are not on the same diagonal.

Starting from a crude method that put the queens absolutely anywhere on the board, we progressed first to a method that never puts two queens in the same row, and then to a still better method that only considers positions where two queens can neither be in the same row nor in the same column. However, all these algorithms share an important defect: they never test a position to see if it is a solution until all the queens have been placed on the board. For instance, even the best of them makes 720 useless attempts to put the last six queens on the board when it has started by putting the first two on the main diagonal, where of course they threaten one another.

Backtracking allows us to do better than this. As a first step, we reformulate the eight queens problem as a tree searching problem. We say that a vector $V[1..k]$ of integers between 1 and 8 is *k-promising*, for $0 \leq k \leq 8$, if none of the k queens placed in positions $(1, V[1]), (2, V[2]), \ldots, (k, V[k])$ threatens any of the others. Mathematically, a vector V is k-promising if, for every pair of integers i and j between 1 and k with $i \neq j$, we have $V[i] - V[j] \notin \{i - j, 0, j - i\}$. For $k \leq 1$, any vector V is k-promising. Solutions to the eight queens problem correspond to vectors that are 8-promising.

Let N be the set of k-promising vectors, $0 \leq k \leq 8$. Let $G = \langle N, A \rangle$ be the directed graph such that $(U, V) \in A$ if and only if there exists an integer k, $0 \leq k < 8$, such that

- U is k-promising,
- V is $(k + 1)$-promising, and
- $U[i] = V[i]$ for every $i \in [1..k]$.

This graph is a tree. Its root is the empty vector corresponding to $k = 0$. Its leaves are either solutions ($k = 8$) or they are dead ends ($k < 8$) such as $[1, 4, 2, 5, 8]$: in such a position it is impossible to place a queen in the next row without threatening at least one of the queens already on the board. The solutions to the eight queens problem can be obtained by exploring this tree. We do not generate the tree explicitly so as to explore it thereafter, however. Rather, nodes are generated and abandoned during the course of the exploration. Depth-first search is the obvious method to use, particularly if we require only one solution.

This technique has two advantages over the algorithm that systematically tries each permutation. First, the number of nodes in the tree is less than $8! = 40\,320$. Although it is not easy to calculate this number theoretically, using a computer it is straightforward to count the nodes: there are 2057. In fact, it suffices to explore 114 nodes to obtain a first solution. Second, to decide whether a vector is k-promising, knowing that it is an extension of a $(k - 1)$-promising vector, we only need check the last queen to be added. This can be speeded up if we associate with each promising node the set of columns, of positive diagonals (at 45 degrees), and of negative diagonals (at 135 degrees) controlled by the queens already placed.

In the following procedure $sol[1..8]$ is a global array. To print all the solutions to the eight queens problem, call $queens(0, \emptyset, \emptyset, \emptyset)$.

```
procedure queens(k, col, diag45, diag135)
    {sol[1..k] is k-promising,
     col = {sol[i] | 1 ≤ i ≤ k},
     diag45 = {sol[i] − i + 1 | 1 ≤ i ≤ k}, and
     diag135 = {sol[i] + i − 1 | 1 ≤ i ≤ k}}
    if k = 8 then {an 8-promising vector is a solution}
                  write sol
    else {explore (k + 1)-promising extensions of sol}
         for j ← 1 to 8 do
             if j ∉ col and j − k ∉ diag45 and j + k ∉ diag135
             then sol[k + 1] ← j
                  {sol[1..k + 1] is (k + 1)-promising}
                  queens(k + 1, col ∪ {j},
                         diag45 ∪ {j − k}, diag135 ∪ {j + k})
```

It is clear that the problem generalizes to an arbitrary number of queens: how can we place n queens on an $n \times n$ "chessboard" in such a way that none of them threatens any of the others? As we might expect, the advantage to be gained by using backtracking instead of an exhaustive approach becomes more pronounced as n increases. For example, for $n = 12$ there are 479 001 600 possible permutations to be considered. Using the permutation generator given previously, the first solution to be found corresponds to the 4 546 044th position examined. On the other hand, the tree explored by the backtracking algorithm contains only 856 189 nodes, and a solution is obtained when the 262nd node is visited. The problem can be further generalized to placing "queens" in three dimensions on an $n \times n \times n$ board; see Problem 9.49.

9.6.3 The general template

Backtracking algorithms can be used even when the solutions sought do not necessarily all have the same length. Here is the general scheme.

```
procedure backtrack(v[1..k])
    {v is a k-promising vector}
    if v is a solution then write v
    {else} for each (k + 1)-promising vector w
                such that w[1..k] = v[1..k]
                do backtrack(w[1..k + 1])
```

The **else** should be present if and only if it is impossible to have two different solutions such that one is a prefix of the other.

Both the knapsack problem and the n queens problem were solved using depth-first search in the corresponding tree. Some problems that can be formulated in terms of exploring an implicit graph have the property that they correspond to an infinite graph. In this case it may be necessary to use breadth-first search to avoid

the interminable exploration of some fruitless infinite branch. Breadth-first search is also appropriate if we have to find a solution starting from some initial position and taking as few steps as possible.

9.7 Branch-and-bound

Like backtracking, branch-and-bound is a technique for exploring an implicit directed graph. Again, this graph is usually acyclic or even a tree. This time, we are looking for the optimal solution to some problem. At each node we calculate a bound on the possible value of any solutions that might lie farther on in the graph. If the bound shows that any such solution must necessarily be worse than the best solution found so far, then we need not go on exploring this part of the graph.

In the simplest version, calculation of the bounds is combined with a breadth-first or a depth-first search, and serves only, as we have just explained, to prune certain branches of a tree or to close paths in a graph. More often, however, the calculated bound is also used to choose which open path looks the most promising, so it can be explored first.

In general terms we may say that a depth-first search finishes exploring nodes in inverse order of their creation, using a stack to hold nodes that have been generated but not yet explored fully. A breadth-first search finishes exploring nodes in the order of their creation, using a queue to hold those that have been generated but not yet explored. Branch-and-bound uses auxiliary computations to decide at each instant which node should be explored next, and a priority list to hold those nodes that have been generated but not yet explored. Remember that heaps are often ideal for holding priority lists; see Section 5.7. We illustrate the technique with two examples.

9.7.1 The assignment problem

In the *assignment problem*, n agents are to be assigned n tasks, each agent having exactly one task to perform. If agent i, $1 \leq i \leq n$, is assigned task j, $1 \leq j \leq n$, then the cost of performing this particular task will be c_{ij}. Given the complete matrix of costs, the problem is to assign agents to tasks so as to minimize the total cost of executing the n tasks.

For example, suppose three agents a, b and c are to be assigned tasks 1, 2 and 3, and the cost matrix is as follows:

	1	2	3
a	4	7	3
b	2	6	1
c	3	9	4

If we allot task 1 to agent a, task 2 to agent b, and task 3 to agent c, then our total cost will be $4 + 6 + 4 = 14$, while if we allot task 3 to agent a, task 2 to agent b, and task 1 to agent c, the cost is only $3 + 6 + 3 = 12$. In this particular example, the reader may verify that the optimal assignment is $a \rightarrow 2$, $b \rightarrow 3$, and $c \rightarrow 1$, whose cost is $7 + 1 + 3 = 11$.

The assignment problem has numerous applications. For instance, instead of talking about agents and tasks, we might formulate the problem in terms of

buildings and sites, where c_{ij} is the cost of erecting building i on site j, and we want to minimize the total cost of the buildings. Other examples are easy to invent. In general, with n agents and n tasks, there are $n!$ possible assignments to consider, too many for an exhaustive search even for moderate values of n. We therefore resort to branch-and-bound.

Suppose we have to solve the instance whose cost matrix is shown in Figure 9.13. To obtain an upper bound on the answer, note that $a \rightarrow 1$, $b \rightarrow 2$, $c \rightarrow 3$, $d \rightarrow 4$ is one possible solution whose cost is $11 + 15 + 19 + 28 = 73$. The optimal solution to the problem cannot cost more than this. Another possible solution is $a \rightarrow 4$, $b \rightarrow 3$, $c \rightarrow 2$, $d \rightarrow 1$ whose cost is obtained by adding the elements in the other diagonal of the cost matrix, giving $40 + 13 + 17 + 17 = 87$. In this case the second solution is no improvement over the first. To obtain a lower bound on the solution, we can argue that whoever executes task 1, the cost will be at least 11; whoever executes task 2, the cost will be at least 12, and so on. Thus adding the smallest elements in each column gives us a lower bound on the answer. In the example, this is $11 + 12 + 13 + 22 = 58$. A second lower bound is obtained by adding the smallest elements in each row, on the grounds that each agent must do something. In this case we find $11 + 13 + 11 + 14 = 49$, not as useful as the previous lower bound. Pulling these facts together, we know that the answer to our instance lies somewhere in $[58..73]$.

	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

Figure 9.13. The cost matrix for an assignment problem

To solve the problem by branch-and-bound, we explore a tree whose nodes correspond to partial assignments. At the root of the tree, no assignments have been made. Subsequently, at each level we fix the assignment of one more agent. At each node we calculate a bound on the solutions that can be obtained by completing the corresponding partial assignment, and we use this bound both to close off paths and to guide the search. Suppose for example that, starting from the root, we decide first to fix the assignment of agent a. Since there are four ways of doing this, there are four branches from the root. Figure 9.14 illustrates the situation.

Here the figure next to each node is a lower bound on the solutions that can be obtained by completing the corresponding partial assignment. We have already seen how the bound of 58 at the root can be obtained. To calculate the bound for the node $a \rightarrow 1$, for example, note first that with this partial assignment task 1 will cost 11. Task 2 will be executed by b, c or d, so the lowest possible cost is 14. Similarly tasks 3 and 4 will also be executed by b, c or d, and their lowest possible costs will be 13 and 22 respectively. Thus a lower bound on any solution obtained by completing the partial assignment $a \rightarrow 1$ is $11 + 14 + 13 + 22 = 60$. Similarly for the node $a \rightarrow 2$, task 2 will be executed by agent a at a cost of 12, while tasks 1, 3

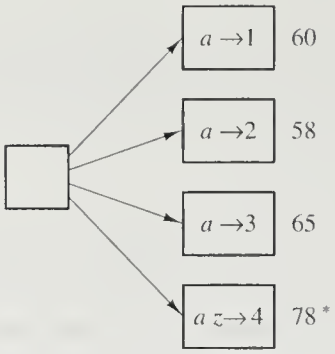

Figure 9.14. After assigning agent a

and 4 will be executed by agents b, c and d at a minimum cost of 11, 13 and 22 respectively. Thus any solution that includes the assignment $a \rightarrow 2$ will cost at least $12 + 11 + 13 + 22 = 58$. The other two lower bounds are obtained similarly. Since we know the optimal solution cannot exceed 73, it is already clear that there is no point in exploring the node $a \rightarrow 4$ any further: any solution obtained by completing this partial assignment will cost at least 78, so it cannot be optimal. The asterisk on this node indicates that it is "dead". However the other three nodes are still alive. Node $a \rightarrow 2$ has the smallest lower bound. Arguing that it therefore looks more promising than the others, this is the one to explore next. We do this by fixing one more element in the partial assignment, say b. In this way we arrive at the situation shown in Figure 9.15.

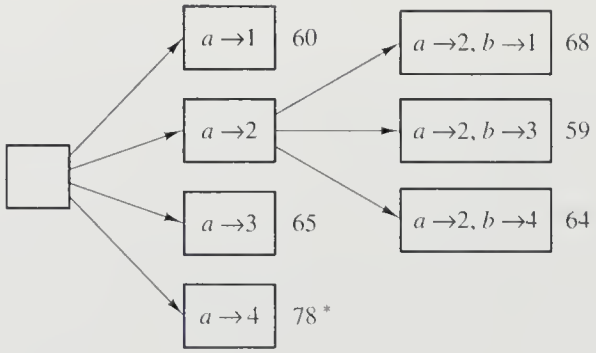

Figure 9.15. After assigning agent b

Again, the figure next to each node gives a lower bound on the cost of solutions that can be obtained by completing the corresponding partial assignment. For example, at node $a \rightarrow 2$, $b \rightarrow 1$, task 1 will cost 14 and task 2 will cost 12. The remaining tasks 3 and 4 must be executed by c or d. The smallest possible cost for task 3 is thus 19, while that for task 4 is 23. Hence a lower bound on the possible solutions is $14 + 12 + 19 + 23 = 68$. The other two new bounds are calculated similarly.

The most promising node in the tree is now $a \rightarrow 2$, $b \rightarrow 3$ with a lower bound of 59. To continue exploring the tree starting at this node, we fix one more element

in the partial assignment, say c. When the assignments of a, b and c are fixed, however, we no longer have any choice about how we assign d, so the solution is complete. The right-hand nodes in Figure 9.16, which shows the next stage of our exploration, therefore correspond to complete solutions.

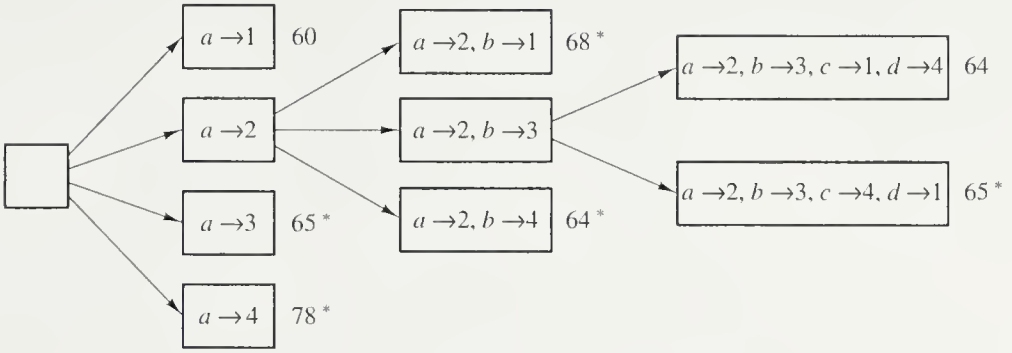

Figure 9.16. After assigning agent c

The solution $a \to 2$, $b \to 3$, $c \to 1$, $d \to 4$, with a cost of 64, is better than either of the solutions we found at the outset, and provides us with a new upper bound on the optimum. Thanks to this new upper bound, we can remove nodes $a \to 3$ and $a \to 2$, $b \to 1$ from further consideration, as indicated by the asterisks. No solution that completes these partial assignments can be as good as the one we have just found. If we only want one solution to the instance, we can eliminate node $a \to 2, b \to 4$ as well.

The only node still worth exploring is $a \to 1$. Proceeding as before, after two steps we obtain the final situation shown in Figure 9.17. The best solution found is $a \to 1$, $b \to 3$, $c \to 4$, $d \to 2$ with a cost of 61. At the remaining unexplored nodes the lower bound is greater than 61, so there is no point in exploring them further. The solution above is therefore the optimal solution to our instance.

The example illustrates that, although at an early stage node $a \to 2$ was the most promising, the optimal solution did not in fact come from there. To obtain our answer, we constructed just 15 of the 41 nodes (1 root, 4 at depth 1, 12 at depth 2, and 24 at depth 3) that are present in a complete tree of the type illustrated. Of the 24 possible solutions, only 6 (including the two used to determine the initial upper bound) were examined.

9.7.2 The knapsack problem (4)

As a second example, consider the knapsack problem; see Sections 8.4 and 9.6.1. Here we require to maximize $\sum_{i=1}^{n} x_i v_i$ subject to $\sum_{i=1}^{n} x_i w_i \leq W$, where the v_i and w_i are all strictly positive, and the x_i are nonnegative integers. This problem too can be solved by branch-and-bound.

Suppose without loss of generality that the variables are numbered so that $v_i/w_i \geq v_{i+1}/w_{i+1}$. Then if the values of $x_1, \ldots, x_k$, $0 \leq k < n$, are fixed, with $\sum_{i=1}^{k} x_i w_i \leq W$, it is easy to see that the value obtainable by adding further items

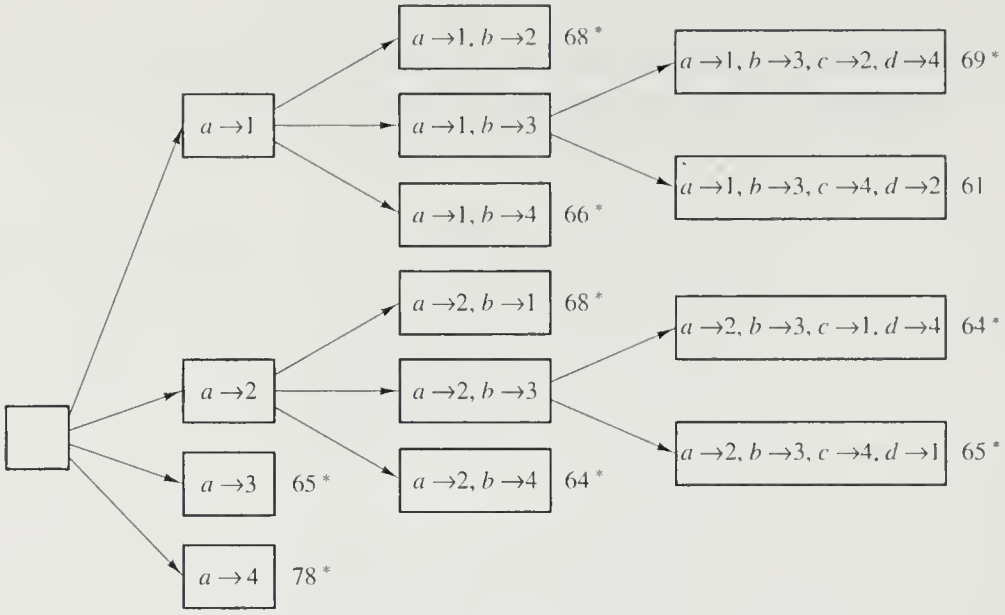

Figure 9.17. The tree completely explored

of types $k+1, \ldots, n$ to the knapsack cannot exceed

$$\sum_{i=1}^{k} x_i v_i + \left(W - \sum_{i=1}^{k} x_i w_i \right) v_{k+1} / w_{k+1}.$$

Here the first term gives the value of the items already in the knapsack, while the second is a bound on the value that can be added.

To solve the problem by branch-and-bound, we explore a tree where at the root none of the values x_i is fixed, and then at each successive level the value of one more variable is determined, in numerical order of the variables. At each node we explore, we only generate those successors that satisfy the weight constraint, so each node has a finite number of successors. Whenever a node is generated we calculate an upper bound on the value of the solution that can be obtained by completing the partially specified load, and use these upper bounds to cut useless branches and to guide the exploration of the tree. We leave the details to the reader.

9.7.3 General considerations

The need to keep a list of nodes that have been generated but not yet completely explored, situated in several levels of the tree and preferably sorted in order of the corresponding bounds, makes branch-and-bound hard to program. The heap is an ideal data structure for holding this list. Unlike depth-first search and its related techniques, no elegant recursive formulation of branch-and-bound is available to the programmer. Nevertheless the technique is sufficiently powerful that it is often used in practical applications.

It is next to impossible to give any precise idea of how well the technique will perform on a given problem using a given bound. There is always a compromise to

be made concerning the quality of the bound to be calculated. With a better bound we look at less nodes, and if we are lucky we may be guided to an optimum solution more quickly. On the other hand, we most likely spend more time at each node calculating the corresponding bound. In the worst case it may turn out that even an excellent bound does not let us cut any branches off the tree, so all the extra work at each node is wasted. In practice, however, for problems of the size encountered in applications, it almost always pays to invest the necessary time in calculating the best possible bound (within reason). For instance, one finds applications such as integer linear programming handled by branch-and-bound, the bound at each node being obtained by solving a related problem in linear programming with continuous variables.

9.8 The minimax principle

Whichever search technique we use, the awkward fact remains that for a game such as chess a complete search of the associated graph is out of the question. In this situation we have to be content with a partial search around the current position. This is the principle underlying an important heuristic called *minimax*. Although this heuristic does not allow us to be certain of winning whenever this is possible, it finds a move that may reasonably be expected to be among the best moves available, while exploring only part of the graph starting from some given position. Exploration of the graph is normally stopped before the terminal positions are reached, using one of several possible criteria, and the positions where exploration stopped are evaluated heuristically. In a sense, this is merely a systematic version of the method used by some human players that consists of looking ahead a small number of moves. Here we only outline the technique.

Suppose then that we want to play a good game of chess. The first step is to define a static evaluation function *eval* that attributes a value to each possible position. Ideally, the value of $eval(u)$ should increase as the position u becomes more favourable to White. It is customary to give values not too far from zero to positions where neither side has a marked advantage, and large negative values to positions that favour Black. This evaluation function must take account of many factors: the number and the type of pieces remaining on both sides, control of the centre, freedom of movement, and so on. A compromise must be made between the accuracy of the function and the time needed to calculate it. When applied to a terminal position, the evaluation function should return $+\infty$ if Black has been mated, $-\infty$ if White has been mated, and 0 if the game is a draw. For example, an evaluation function that takes good account of the static aspects of the position, but that is too simplistic to be of real use, might be the following: for nonterminal positions, count 1 point for each white pawn, $3^1/_4$ points for each white bishop or knight, 5 points for each white rook, and 10 points for each white queen; subtract a similar number of points for each black piece.

If the static evaluation function were perfect, it would be easy to determine the best move to make. Suppose it is White's turn to move in position u. The best move would be to the position v that maximizes $eval(v)$ among all the successors of u.

```
val ← −∞
for each position w that is a successor of u do
    if eval(w) ≥ val then val ← eval(w)
                          v ← w
```

This simplistic approach would not be very successful using the evaluation function suggested earlier, since it would not hesitate to sacrifice a queen to take a pawn!

If the evaluation function is not perfect, a better strategy for White is to assume that Black will reply with the move that minimizes the function *eval*, since the smaller the value of this function, the better the position is supposed to be for him. Ideally, Black would like a large negative value. We are now looking one move ahead. (Remember we agreed to call each action a move, avoiding the term "half-move".)

```
val ← −∞
for each position w that is a successor of u do
    if w has no successor
        then valw ← eval(w)
        else valw ← min{eval(x) | x is a successor of w }
    if valw ≥ val then val ← valw
                       v ← w
```

There is now no question of giving away a queen to take a pawn: which of course may be exactly the wrong rule to apply if it prevents White from finding the winning move. Maybe if he looked further ahead the gambit would turn out to be profitable. On the other hand, we are sure to avoid moves that would allow Black to mate immediately (provided we *can* avoid this).

To add more dynamic aspects to the static evaluation provided by *eval*, it is preferable to look several moves ahead. To look n moves ahead from position u, White should move to the position v given by

```
val ← −∞
for each position w that is a successor of u do
    B ← Black(w, n)
    if B ≥ val then val ← B
                    v ← w
```

where the functions *Black* and *White* are the following.

```
function Black(w, n)
    if n = 0 or w has no successor
        then return eval(w)
        else return min{White(x, n − 1) | x is a successor of w}

function White(w, n)
    if n = 0 or w has no successor
        then return eval(w)
        else return max{Black(x, n − 1) | x is a successor of w}
```

We see why the technique is called minimax. Black tries to minimize the advantage he allows to White, and White, on the other hand, tries to maximize the advantage he obtains from each move.

More generally, suppose Figure 9.18 shows part of the graph corresponding to some game. If the values attached to the nodes on the lowest level are obtained by applying the function *eval* to the corresponding positions, the values for the other nodes can be calculated using the minimax rule. In the example we suppose that player A is trying to maximize the evaluation function and that player B is trying to minimize it. If A plays to maximize his advantage, he will choose the second of the three possible moves. This assures him a value of at least 10; but see Problem 9.55.

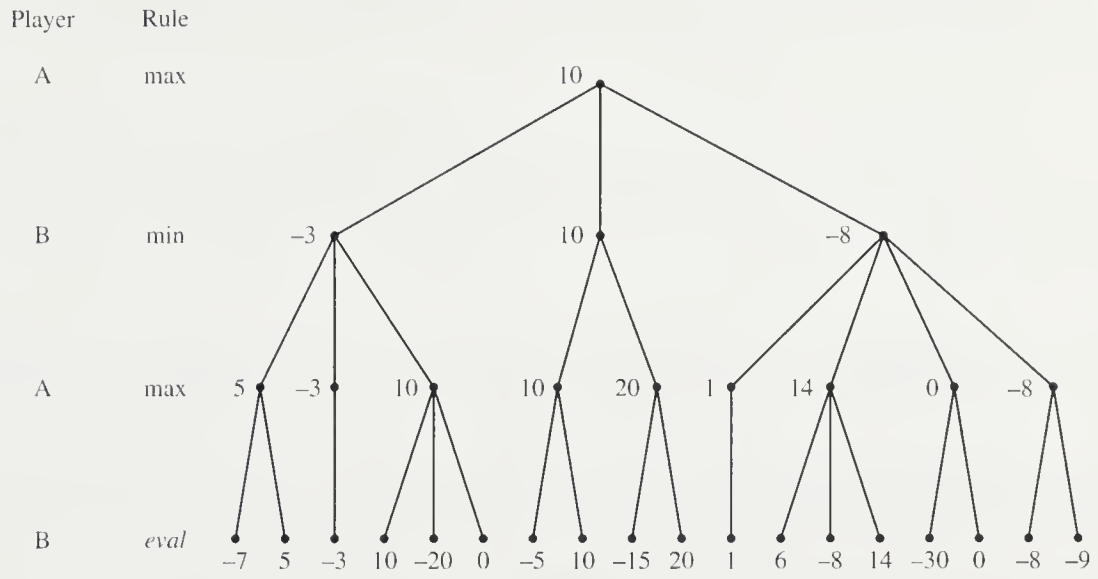

Figure 9.18. The minimax principle

The basic minimax technique can be improved in a number of ways. For example, it may be worthwhile to explore the most promising moves in greater depth. Similarly, the exploration of certain branches can be abandoned early if the information we have about them is already sufficient to show that they cannot possibly influence the value of nodes further up the tree. This second type of improvement, which we shall not describe in this book, is generally known as *alpha-beta pruning*.

9.9 Problems

Problem 9.1. Add nodes $\langle 8,7\rangle$, $\langle 7,6\rangle$, $\langle 6,5\rangle$ and their descendants to the graph of Figure 9.1.

Problem 9.2. Can a winning position in the game described in Section 9.1 have more than one losing position among its successors? In other words, are there positions in which several different winning moves are available? Can this happen in the case of a winning initial position $\langle n, n-1\rangle$?

Problem 9.3. Suppose we change the rules of the game of Section 9.1 so that the player who is forced to take the last match *loses*. This is the *misère* version of the game. Suppose also that the first player must take at least one match and that he must leave at least two. Among the initial positions with three to eight matches, which are now winning positions for the first player?

Problem 9.4. Modify the algorithm *recwin* of Section 9.1 so it returns an integer k, where $k = 0$ if the position is a losing position, and $1 \leq k \leq j$ if it is a winning move to take k matches.

Problem 9.5. Prove that in the game described in Section 9.1 the first player has a winning strategy if and only if the initial number of matches does not appear in the Fibonacci sequence.

Problem 9.6. Consider a game that cannot continue for an infinite number of moves, and where no position offers an infinite number of legal moves to the player whose turn it is. Let G be the directed graph corresponding to this game. Show that the method described in Section 9.1 allows *all* the nodes of G to be labelled as *win*, *lose* or *draw*.

Problem 9.7. Consider the following game. Initially a heap of n matches is placed on the table between two players. Each player in turn may either (a) split any heap on the table into two unequal heaps, or (b) remove one or two matches from any heap on the table. He may not do both. He may only split one heap, and if he chooses to remove two matches, they must both come from the same heap. The player who removes the last match wins.
For example, suppose that during play we arrive at the position $\{5, 4\}$; that is, there are two heaps on the table, one of 5 matches, the other of 4. The player whose turn it is may move to $\{4, 3, 2\}$ or $\{4, 4, 1\}$ by splitting the heap of 5, to $\{5, 3, 1\}$ by splitting the heap of 4 (but not to $\{5, 2, 2\}$, since the new heaps must be unequal), or to $\{4, 4\}$, $\{4, 3\}$, $\{5, 3\}$ or $\{5, 2\}$ by taking one or two matches from either of the heaps.
Sketch the graph of the game for $n = 5$. If both play correctly, does the first or the second player win?

Problem 9.8. Repeat the previous problem for the *misère* version of the game, where the player who takes the last match loses.

Problem 9.9. Consider a game of the type described in Section 9.1. When we use a graph of winning and losing positions to describe such a game, we implicitly assume that both players will move intelligently so as to maximize their chances of winning. Can a player in a winning position lose if his opponent moves stupidly and makes an unexpected "error"?

Problem 9.10. For any of the tree traversal techniques mentioned in Section 9.2, prove that a recursive implementation takes storage space in $\Omega(n)$ in the worst case.

Problem 9.11. Show how any of the tree traversal techniques mentioned in Section 9.2 can be implemented so as to take only a time in $\Theta(n)$ and storage space in $\Theta(1)$, even when the nodes do not contain a pointer to their parents (in which case the problem is trivial).

Problem 9.12. Generalize the concepts of preorder and postorder to arbitrary (nonbinary) trees. Assume the trees are represented as in Figure 5.7. Prove that both these techniques still run in a time in the order of the number of nodes in the tree to be traversed.

Problem 9.13. In Section 9.2 we gave one way of preconditioning a tree so as to be able thereafter to verify rapidly whether one node is an ancestor of another. There exist several similar ways of arriving at the same result. Show, for example, that it can be done using a traversal in preorder followed by a traversal in inverted preorder, which visits first a node and then its subtrees from right to left.
If the trees are represented as in Figure 5.7, is this method more or less efficient than the one given in Section 9.2, or are they comparable?

Problem 9.14. Show how a depth-first search progresses through the graph of Figure 9.3 if the starting point is node 6 and the neighbours of a given node are examined (a) in numerical order, and (b) in decreasing numerical order.
Exhibit the spanning tree and the numbering of the nodes of the graph generated by each of these searches.

Problem 9.15. Analyse the running time of algorithm *dfs* if the graph to be explored is represented by an adjacency matrix (type *adjgraph* of Section 5.4) rather than by lists of adjacent nodes.

Problem 9.16. Show how depth-first search can be used to find the connected components of an undirected graph.

Problem 9.17. Let G be an undirected graph, and let T be the spanning tree generated by a depth-first search of G. Prove that an edge of G that has no corresponding edge in T cannot join nodes in different branches of the tree, but must necessarily join some node v to one of its ancestors in T.

Problem 9.18. Prove or give a counterexample:

(a) if a graph is biconnected, then it is bicoherent;

(b) if a graph is bicoherent, then it is biconnected.

Problem 9.19. Prove that a node v in a connected graph is an articulation point if and only if there exist two nodes a and b different from v such that every path joining a and b passes through v.

Problem 9.20. Prove that for every pair of distinct nodes v and w in a biconnected graph, there exist at least two paths joining v and w that have no nodes in common except the starting and ending nodes.

Problem 9.21. In the algorithm for finding the articulation points of an undirected graph given in Section 9.3.1, show how to calculate the values of both *prenum* and *highest* during the depth-first search of the graph, and implement the corresponding algorithm.

Problem 9.22. The example in Section 9.3.1 finds the articulation points for the graph of Figure 9.3 using a depth-first search starting at node 1. Verify that the same articulation points are found if the search starts at node 6.

Problem 9.23. Illustrate how the algorithm for finding the articulation points of an undirected graph given in Section 9.3.1 works on the graph of Figure 9.19, starting the search (a) at node 1, and (b) at node 3.

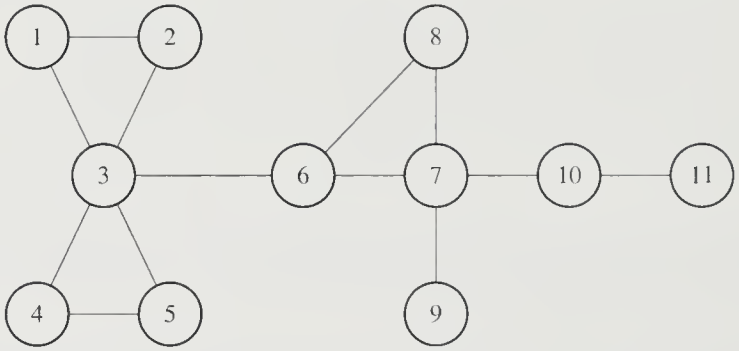

Figure 9.19. A graph with articulation points

Problem 9.24. A graph is *planar* if it can be drawn on a sheet of paper so that none of the edges cross. Use depth-first search to design an algorithm capable of deciding in linear time if a given graph is planar.

Problem 9.25. Illustrate the progress of the depth-first search algorithm on the graph of Figure 9.5 if the starting point is node 1 and the neighbours of a given node are examined in decreasing numerical order.

Problem 9.26. Let F be the forest generated by a depth-first search on a directed graph $G = \langle N, A \rangle$. Prove that G is acyclic if and only if the set of edges of G with no corresponding edge in F includes no edge leading from a node to one of its ancestors in the forest.

Problem 9.27. For the graph of Figure 9.8, what is the topological order obtained if we use the procedure suggested in Section 9.4.1, starting the depth-first search at node 1, and visiting the neighbours of a node in numerical order?

Problem 9.28. What is wrong with the following argument? When we visit node v of a graph G using depth-first search, we immediately explore all the other nodes that can be reached from v by following edges of G. In the topological ordering,

these other nodes must come later than v. Thus to obtain a topological ordering of the nodes, it suffices to add an extra line

write v

at the beginning of procedure *dfs*.

Problem 9.29. A directed graph is *strongly connected* if there exist paths from u to v and from v to u for every pair of nodes u and v. If a directed graph is not strongly connected, the largest sets of nodes such that the induced subgraphs are strongly connected are called the *strongly connected components* of the graph. For example, the strongly connected components of the graph in Figure 9.5 are $\{1,2,3\}$, $\{5,6\}$ and $\{4,7,8\}$. Design an efficient algorithm based on depth-first search to find the strongly connected components of a graph.

Problem 9.30. After a breadth-first search in an undirected graph G, let F be the forest of trees that is generated. If $\{u, v\}$ is an edge of G that has no corresponding edge in F (such an edge is represented by a broken line in Figure 9.10), show that the nodes u and v lie in the same tree in F, but neither u nor v is an ancestor of the other.

Problem 9.31. Show how a breadth-first search progresses through the graph of Figure 9.5, assuming that the neighbours of a node are visited in numerical order, and that the necessary starting points are also chosen in numerical order.

Problem 9.32. Sketch the forest generated by the search of Problem 9.31, showing the remaining edges of the graph as broken lines. How many kinds of "broken" edges are possible?

Problem 9.33. Justify the claim that a depth-first search of the graph of Figure 9.11, starting at node 1 and visiting neighbours in the order "first division by 3, then multiplication by 2", works down to node 12 and then visits successively nodes 24, 48, 96, 32 and 64.

Problem 9.34. List the first 15 nodes visited by a breadth-first search of the graph of Figure 9.11 starting at node 1 and visiting neighbours in the order "first division by 3, then multiplication by 2".

Problem 9.35. Section 9.5 gives one way to produce the value 13, starting from 1 and using the operations multiplication by 2 and division by 3. Find another way to produce the value 13 using the same starting point and the same operations.

Problem 9.36. A node p of a directed graph $G = \langle N, A \rangle$ is called a *sink* if for every node $v \in N$, $v \neq p$, the edge (v, p) exists, whereas the edge (p, v) does not exist. Write an algorithm that can detect the presence of a sink in G in a time in $O(n)$, where n is the number of nodes in the graph. Your algorithm should accept the graph represented by its adjacency matrix (type *adjgraph* of Section 5.4). Notice that a running time in $O(n)$ for this problem is remarkable given that the instance takes a space in $\Omega(n^2)$ merely to write down.

Problem 9.37. An *Euler path* in an undirected graph is a path where every edge appears exactly once. Design an algorithm that determines whether or not a given graph has an Euler path, and prints the path if so. How much time does your algorithm take?

Problem 9.38. Repeat Problem 9.37 for a directed graph.

Problem 9.39. In either a directed or an undirected graph, a path is said to be Hamiltonian if it passes exactly once through each node of the graph, without coming back to the starting node. In a directed graph, however, the direction of the edges in the path must be taken into account. Prove that if a directed graph is complete (that is, if each pair of nodes is joined in at least one direction) then it has a Hamiltonian path. Give an algorithm for finding such a path in this case.

Problem 9.40. Sketch the search tree explored by a backtracking algorithm solving the same instance of the knapsack problem as that in Section 9.6.1, but assuming this time that items are loaded in order of decreasing weight.

Problem 9.41. Solve the same instance of the knapsack problem as that in Section 9.6.1 by dynamic programming. You will need to work Problem 8.15 first.

Problem 9.42. Adapt the function *backpack* of Section 9.6.1 to give the composition of the best load as well as its value.

Problem 9.43. Let the vector $q[1..8]$ represent the positions of eight queens on a chessboard, with one queen in each row: if $q[i] = j$, $1 \leq i \leq 8$, $1 \leq j \leq 8$, the queen in row i is in column j. Write a function *solution*(q) that returns *false* if at least one pair of queens threaten each another, and returns *true* otherwise.

Problem 9.44. Given that the algorithm *queens*1 finds a solution and stops after trying 1 299 852 positions, solve the eight queens problem without using a computer.

Problem 9.45. Suppose the procedure *use*(T) called from the procedure *perm*(i) of Section 9.6 consists simply of printing the array T on a new line. Show the result of calling *perm*(1) when $n = 4$.

Problem 9.46. Suppose the procedure *use*(T) called from the procedure *perm*(i) of Section 9.6 takes constant time. How much time is needed, as a function of n, to execute the call *perm*(1)?
Rework the problem assuming now that *use*(T) takes a time in $\Theta(n)$.

Problem 9.47. For which values of n does the n queens problem have no solutions? Prove your answer.

Problem 9.48. How many solutions are there to the eight queens problem? How many *distinct* solutions are there if we do not distinguish solutions that can be transformed into one another by rotations and reflections?

Problem 9.49. Investigate the problem of placing k "queens" on an $n \times n \times n$ three-dimensional board. Assume that a "queen" in three dimensions threatens positions in the same rows, columns or diagonals as herself in the obvious way. Clearly k cannot exceed n^2. Not counting the trivial case $n = 1$, what is the smallest value of n for which a solution with n^2 "queens" is possible?

Problem 9.50. A Boolean array $M[1..n, 1..n]$ represents a square maze. In general, starting from a given point, you may move to adjacent points in the same row or the same column. If $M[i, j]$ is *true*, you may pass through point (i, j); if $M[i, j]$ is *false*, you may not pass through point (i, j). Figure 9.20 gives an example.
Give a backtracking algorithm that finds a path, if one exists, from $(1, 1)$ to (n, n). Without being completely formal—for instance, you may use statements such as "**for** each point v that is a neighbour of x **do** ..."—your algorithm must be clear and precise.

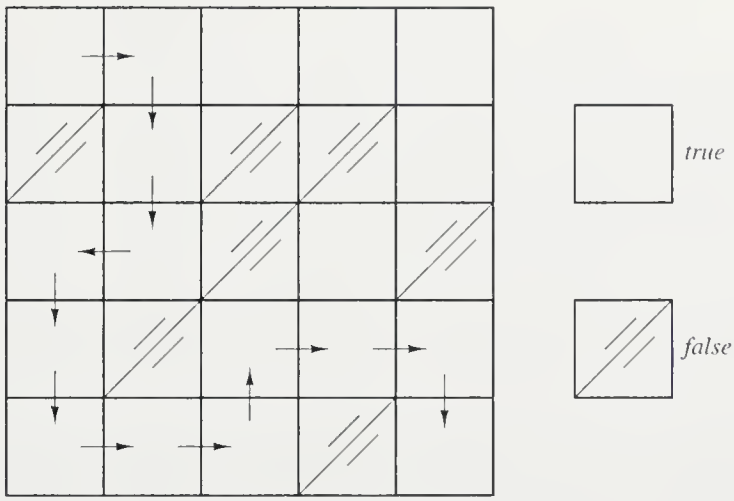

Figure 9.20. A maze

Problem 9.51. The backtracking method suggested in Problem 9.50 illustrates the principle, but is very inefficient in practice. Find a much better way of solving the same problem.

Problem 9.52. In Section 9.7 we calculated lower bounds for the nodes in the search tree by assuming that each unassigned task would be executed by the unassigned agent who could do it at least cost. This is like crossing out the rows and columns of the cost matrix corresponding to agents and tasks already assigned, and then adding the minimum elements from each remaining column. An alternative method of calculating bounds is to assume that each unassigned agent will perform the task he can do at least cost. This is like crossing out the rows and columns of the cost matrix corresponding to agents and tasks already assigned, and then adding the minimum elements from each remaining row. Show how a branch-and-bound algorithm for the instance in Section 9.7 proceeds using this second method of calculating lower bounds.

Problem 9.53. Use branch-and-bound to solve the assignment problems with the following cost matrices:

(a)

	1	2	3	4
a	94	1	54	68
b	74	10	88	82
c	62	88	8	76
d	11	74	81	21

(b)

	1	2	3	4	5
a	11	17	8	16	20
b	9	7	12	6	15
c	13	16	15	12	16
d	21	24	17	28	26
e	14	10	12	11	15

Problem 9.54. Four types of object are available, whose weights are respectively 7, 4, 3 and 2 units, and whose values are 9, 5, 3 and 1. We can carry a maximum of 10 units of weight. Objects may not be broken into smaller pieces. Determine the most valuable load that can be carried, using (a) dynamic programming, and (b) branch-and-bound. For part (a), you will need to work Problem 8.15 first.

Problem 9.55. Looking at the tree of Figure 9.18, we said that the player about to move in this position is assured a value of at least 10. Is this strictly true?

Problem 9.56. Let u correspond to the initial position of the pieces in the game of chess. What can you say about $White(u, 12345)$, besides the fact that it would take far too long to calculate in practice, even with a special-purpose computer? Justify your answer.

Problem 9.57. Three-dimensional tictactoe is played on a $4 \times 4 \times 4$ "board". As in ordinary tictactoe, players X and O mark squares alternately. The winner is the first to align four of his own marks in any direction, parallel to one side of the board or along a diagonal. There are 76 ways of winning. Devise an evaluation function for positions in this game, and use it to write a program to play the game against a human opponent.

9.10 References and further reading

There exist a number of books concerning graph algorithms or combinatorial problems that are often posed in terms of graphs. We mention in chronological order Christofides (1975), Lawler (1976), Reingold, Nievergelt and Deo (1977), Gondran and Minoux (1979), Even (1980), Papadimitriou and Steiglitz (1982), Tarjan (1983) and Melhorn (1984b). The mathematical notion of a graph is treated at length in Berge (1958, 1970).

Berlekamp, Conway and Guy (1982) give more versions of the game of Nim than most people will care to read. The game in Problem 9.7 is a variant of Grundy's game, also discussed by Berlekamp, Conway and Guy (1982). The book by Nilsson (1971) is a goldmine of ideas concerning graphs and games, the minimax technique, and alpha-beta pruning. Some algorithms for playing chess appear in Good (1968). A lively account of the first time a computer program beat the world backgammon champion is given in Deyong (1977). For a more technical description of this feat, consult Berliner (1980). In 1994, Garri Kasparov, the world chess champion, was beaten by a Pentium microcomputer. At the time of writing, humans are still unbeaten at the game of go.

A solution to Problem 9.11 is given in Robson (1973). To learn more about preconditioning, read Brassard and Bratley (1988). Many algorithms based on depth-first search can be found in Tarjan (1972), Hopcroft and Tarjan (1973) and Aho, Hopcroft and Ullman (1974, 1983). Problem 9.24 is solved in Hopcroft and Tarjan (1974). See also Rosenthal and Goldner (1977) for an efficient algorithm that, given an undirected graph that is connected but not biconnected, finds a smallest set of edges that can be added to make the graph biconnected.

The problem involving multiplying by 3 and dividing by 2 is reminiscent of Collatz's problem: see Problem E16 in Guy (1981). Backtracking is described in Golomb and Baumert (1965) and techniques for analysing its efficiency are given in Knuth (1975). The eight queens problem was invented by Bezzel in 1848; see the account of Ball (1967). Irving (1984) gives a particularly efficient backtracking algorithm to find all the solutions for the n queens problem. The problem can be solved in linear time with a divide-and-conquer algorithm due to Abramson and Yung (1989) provided we are happy with a single solution. We shall come back to this problem in Chapter 10. The three-dimensional n^2 queens problem mentioned in Problem 9.49 was posed by McCarty (1978) and solved by Allison, Yee and McGaughey (1989): there are no solutions for $n < 11$ but 264 solutions exist for $n = 11$.

Branch-and-bound is explained in Lawler and Wood (1966). The assignment problem is well known in operations research: see Hillier and Lieberman (1967). For an example of solving the knapsack problem by backtracking, see Hu (1981). Branch-and-bound is used to solve the travelling salesperson problem in Bellmore and Nemhauser (1968).