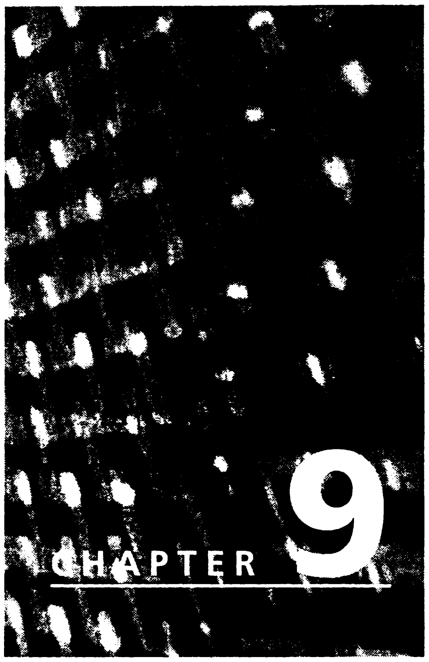


DYNAMIC PROGRAMMING

Dynamic programming is a design strategy that involves dynamically constructing a solution S to a given problem using solutions $S_1, S_2, \ldots, S_m$ to smaller (or simpler) instances of the problem. The solution S_i to a given smaller problem instance is itself built from the solutions to even smaller (or simpler) problem instances, and so forth. We start with the known solutions to the smallest (simplest) problem instances and build from there in a bottom-up fashion. To be able to reconstruct S from $S_1, S_2, \ldots, S_m$, we usually require some additional information. We let *Combine* denote the function that combines $S_1, S_2, \ldots, S_m$, using the additional information to obtain S, so that

$$S = \mathit{Combine}(S_1, S_2, \ldots, S_m).$$

Dynamic programming is similar to divide-and-conquer in the sense that it is based on a recursive division of a problem instance into smaller or simpler problem instances. However, whereas divide-and-conquer algorithms often use a top-down resolution method, dynamic programming algorithms invariably proceed by solving all the simplest problem instances before combining them into more complicated problem instances in a bottom-up fashion. Further, unlike many instances of divide-and-conquer, dynamic programming algorithms typically do not recalculate the solution to a given problem instance. Dynamic programming algorithms for optimization problems also can avoid generating suboptimal problem instances when the *Principle of Optimality* holds, thereby leading to increased efficiency.

9.1 Optimization Problems and the Principle of Optimality

The method of dynamic programming is most effective in solving optimization problems when the Principle of Optimality holds. Consider the set of all *feasible* solutions to an optimization problem; that is, all the solutions satisfying the constraints of the problem. An *optimal* solution S is a solution that optimizes (minimizes or maximizes) the objective function. If we wish to obtain an optimal solution S to the given problem instance, then we must optimize (minimize or maximize) over *all* solutions $S_1, S_2, \ldots, S_m$ such that $S = \mathit{Combine}(S_1, S_2, \ldots, S_m)$. For many problems, it is computationally infeasible to examine all feasible solutions because exponentially many possibilities exist. Fortunately, we can drastically reduce the number of problem instances that we need to consider if the Principle of Optimality holds.

DEFINITION 9.1.1 Given an optimization problem and an associated function *Combine*, the *Principle of Optimality* holds if the following is always true: If $S = \mathit{Combine}(S_1, S_2, \ldots, S_m)$ and S is an *optimal* solution to the problem instance, then $S_1, S_2, \ldots . S_m$, are *optimal* solutions to their associated problem instances.

Key Fact

The efficiency of dynamic programming solutions based on a recurrence relation expressing the principle of optimality results from (1) the bottom-up resolution of the recurrence, thereby eliminating redundant recalculations, and (2) eliminating suboptimal solutions to subproblems as we build up optimal solutions to larger problems; that is, we use only optimal solution "building blocks" in constructing our optimal solution.

We first illustrate the Principle of Optimality for the problem of finding a parenthesization of a matrix product of matrices $M_0, \ldots, M_{n-1}$ that minimizes the total number of (scalar) multiplications over all possible parenthesizations. If $(M_0 \cdots M_k)(M_{k+1} \cdots M_{n-1})$ is the "first-cut" set of parentheses (and the last product performed), then the matrix products $M_0 \cdots M_k$ and $M_{k+1} \cdots M_{n-1}$ must both be parenthesized in such a way as to minimize the number of multiplications required to carry out the respective products. As a second example, consider the problem of finding optimal binary search trees for a set of distinct keys. Recall that a binary search tree T for keys $K_0 < \cdots < K_{n-1}$ is a binary tree on n nodes, each containing a key such that the following property is satisfied: Given any node v in the tree, each key in the left subtree rooted at v is no larger than the key in v, and each key in the right subtree rooted at v is no smaller than the key in v (see Figure 4.17). If K_i is the key in the root, then the left subtree L of the root contains $K_0, \dots, K_{i-1}$, and the right subtree R of the root contains $K_{i+1}, \dots, K_{n-1}$. Given a binary search tree T for keys $K_0, \dots, K_{n-1}$, let K_i denote the key associated with the root of T, and let L and R denote the left and right subtrees (of the root) of T, respectively. Again, it follows that L (solution S_1) is a binary search tree for keys $K_0, \dots, K_{i-1}$, and R (solution S_2) is a binary search tree for keys $K_{i+1}, \dots, K_{n-1}$. Given L and R, the function *Combine*(L,R) merely reconstructs the tree T using K_i as the root. In the next section, we show that the Principle of Optimality holds for this problem by showing that if T is an optimal binary search tree, then so are L and R.

9.2 Optimal Parenthesization for Computing a Chained Matrix Product

Our first example of dynamic programming is an algorithm for the problem of parenthesizing a chained matrix product so as to minimize the number of multiplications performed when computing the product. When solving this problem, we will assume the straightforward method of matrix multiplication. If A and B are matrices of dimensions $p \times q$ and $q \times r$, then the matrix product AB involves pqr multiplications. Given a sequence (or chain) of matrices $M_0, M_1, \dots, M_{n-1}$, consider the product $M_0 M_1 \cdots M_{n-1}$, where the matrix M_i has dimension $d_i \times d_{i+1}$, $i = 0, \dots, n$, for a suitable sequence of positive integers $d_0, d_1, \dots, d_n$. Because a matrix product is an associative operation, we can evaluate the chained product in one of many ways, depending on how we choose to parenthesize the expression. It turns out that the manner in which the expression is parenthesized can make a major difference in the total number of multiplications performed when computing the chained product. In this section, we consider the problem of finding an *optimal parenthesization*—that is, a parenthesization that minimizes the total number of multiplications performed using ordinary matrix products.

We illustrate the problem with an example that commonly occurs in multivariate calculus. Suppose A and B are $n \times n$ matrices, X is an $n \times 1$ column vector, and we wish to evaluate ABX. The product ABX can be parenthesized in two ways, $(AB)X$ and $A(BX)$, resulting in $n^3 + n^2$ and $2n^2$ multiplications, respectively. Thus, the two ways of parenthesizing make a rather dramatic difference in the number of multiplications performed; that is, order $\Theta(n^3)$ versus order $\Theta(n^2)$.

The following is a formal, recursive definition of a fully parenthesized chained matrix product and its associated first cut.

DEFINITION 9.2.1 Given the sequence of matrices $M_0, M_1, \dots, M_{n-1}$, P is a *fully parenthesized* matrix product of $M_0, M_1, \dots, M_{n-1}$ (which, for convenience, we simply call a *parenthesization of* $M_0M_1 \cdots M_{n-1}$) if P satisfies

$$P = M_0, \quad n = 1,$$
$$P = (P_1P_2), \quad n > 1,$$

where for some k, P_1 and P_2 are parenthesizations of the matrix products $M_0M_1 \cdots M_k$ and $M_{k+1}M_{k+2} \cdots M_{n-1}$, respectively. We call P_1 and P_2 the *left* and *right* parenthesizations of P, respectively. We call the index k the *first-cut index of P*.

The table in Figure 9.1 shows all the parenthesizations of the matrices M_0, M_1, M_2, and M_3 having dimensions 20×10, 10×50, 50×5, and 5×30, respectively, with the optimal parenthesizations highlighted.

There is one-to-one correspondence between parenthesizations of $M_0M_1 \cdots M_{n-1}$ and 2-trees having n leaf nodes. Given a parenthesization P of $M_0M_1 \cdots M_{n-1}$, if $n = 1$, its associated 2-tree $T(P)$ consists of a single node corresponding to the matrix M_0; otherwise, $T(P)$ has left subtree $T(P_1)$ and right subtree $T(P_2)$, where P_1 and P_2 are the left and right parenthesizations of P. The 2-tree $T(P)$ is the *expression tree* for P (see Figure 9.2).

FIGURE 9.1 Number of multiplications performed for each full parenthesization shown for matrices M_0, M_1, M_2, and M_3 having dimensions 20×10, 10×50, 50×5, and 5×30, respectively. The optimal parenthesizations are shaded.

no. mult.	no. mult.	no. mult.	no. mult.
M_0	(M_0M_1) 10000	$(M_0(M_1M_2))$ 3500	$(M_0(M_1(M_2M_3)))$ 28500
		$((M_0M_1)M_2)$ 15000	$(M_0((M_1M_2)M_3))$ 10000
			$((M_0M_1)(M_2M_3))$ 47500
			$((M_0(M_1M_2))M_3)$ 6500
			$(((M_0M_1)M_2)M_3)$ 18000

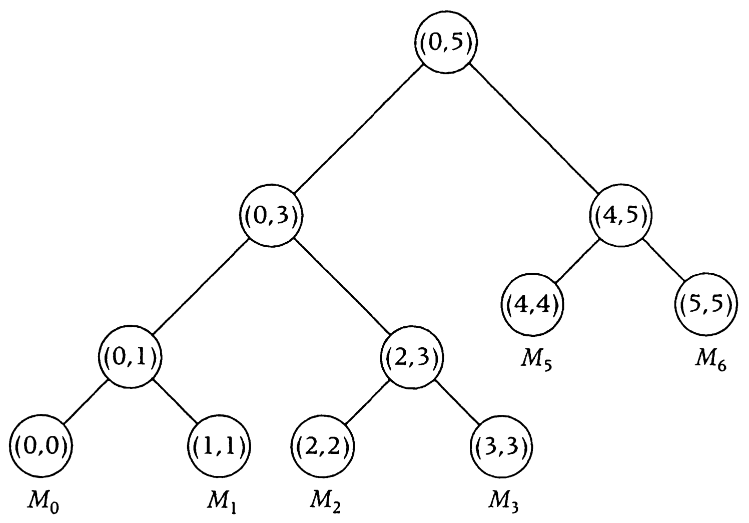


FIGURE 9.2

Associated expression 2-tree for parenthesization $(((M_0M_1)(M_2M_3))(M_4M_5))$. The label (i, j) inside each node indicates that the matrix product associated with the node involves matrices $M_i, M_{i+1}, \dots, M_j$.

Thus, the number of parenthesizations p_n equals the number t_n of 2-trees having n leaf nodes, so that by Exercise 4.14 we have

$$p_n = \frac{1}{n}\binom{2n-2}{n-1} \geq \frac{4^{n-1}}{2n^2 - n} \in \Omega\left(\frac{4^n}{n^2}\right). \qquad \textbf{(9.2.1)}$$

Hence, a brute-force algorithm that examines all possible parenthesizations is computationally infeasible.

We are led to consider a dynamic programming solution to our problem by noting that the Principle of Optimality holds for optimal parenthesizing. Indeed, if we consider any optimal parenthesization P for $M_0M_1 \cdots M_{n-1}$, clearly both the left and right parenthesizations P_1 and P_2 of P must be optimal for P to be optimal.

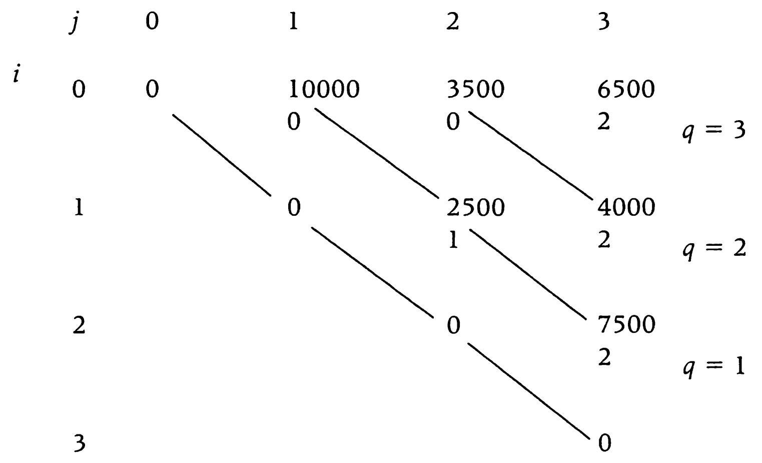
For $0 \leq i \leq j \leq n - 1$, let m_{ij} denote the number of multiplications performed using an optimal parenthesization of $M_iM_{i+1} \cdots M_j$. By the Principle of Optimality, we have the following recurrence for m_{ij} based on making an optimal choice for the first-cut index:

$$\begin{aligned} m_{ij} &= \min_k\{m_{ik} + m_{k+1,j} + d_id_{k+1}d_{j+1} : 0 \leq i \leq k < j \leq n-1\} \\ &\qquad\qquad \textbf{init. cond. } m_{ii} = 0, i = 0, \dots, n-1. \end{aligned} \qquad \textbf{(9.2.2)}$$

The value $m_{0,n-1}$ corresponds to the minimum number of multiplications performed when computing $M_0M_1 \cdots M_{n-1}$. We could base a divide-and-conquer algorithm *ParenthesizeRec* directly on a top-down implementation of the recurrence relation (9.2.2). Unfortunately, a great many recalculations are performed by *ParenthesizeRec*, and it ends up doing $\Omega(3^n)$ multiplications to compute the minimum number $m_{0,\,n-1}$ corresponding to an optimal parenthesization.

A straightforward dynamic programming algorithm proceeds by computing the values m_{ij}, $0 \le i \le j \le n - 1$, in a bottom-up fashion using (9.2.2), thereby avoiding recalculations. Note that the values m_{ij}, $0 \le i \le j \le n - 1$, occupy the upper-right triangular portion of an $n \times n$ table. Our bottom-up resolution proceeds throughout the upper-right triangular portion diagonal by diagonal, starting from the bottom diagonal consisting of the elements $m_{ii} = 0$, $i = 0, \dots, n - 1$. The qth diagonal consists of the elements $m_{i,\,i+q}$, $q = 0, \dots, n - 1$. Figure 9.3 illustrates the computation of m_{ij} for the example given in Figure 9.1. When computing m_{ij}, we also generate a table c_{ij} of indices k, where the minimum in (9.2.2) occurs; that is, c_{ij} is where the first cut in $M_iM_{i+1} \cdots M_j$ is made in an optimal parenthesization. The values c_{ij} can then be used to actually compute the matrix product according to the optimal parenthesization.

The following procedure, *OptimalParenthesization,* accepts as input the *dimension sequence* $d[0{:}n]$, where matrix M_i has dimension $d_i \times d_{i+1}$, $i = 0, \dots, n - 1$. Procedure *OptimalParenthesization* outputs the matrix $m[0{:}n-1, 0{:}n-1]$, where $m[i,j] = m_{ij}$, $0 \le i \le j \le n - 1$, is defined by recurrence (9.2.2). *OptimalParenthesization* also outputs the matrix *FirstCut*$[0{:}n-1, 0{:}n-1]$, where *FirstCut*$[i,j] = c_{ij}$, $0 \le i \le j \le n - 1$, which is the first-cut index in an optimal parenthesization for $M_i \cdots M_j$.



i \ j	0	1	2	3	
0	0	10000 0	3500 0	6500 2	$q = 3$
1		0	2500 1	4000 2	$q = 2$
2			0	7500 2	$q = 1$
3				0	$q = 0$

FIGURE 9.3 Table showing values m_{ij}, $0 \le i \le j \le 3$, computed diagonal by diagonal from $q = 0$ to $q = 3$ using the bottom-up resolution of (9.2.2) for matrices M_0, M_1, M_2, and M_3 having dimensions 20×10, 10×50, 50×5, and 5×30, respectively. The values of c_{ij} are shown underneath each m_{ij}, $0 \le i \le j \le 3$.

```
procedure OptimalParenthesization(d[0:n], m[0:n − 1, 0:n − 1], FirstCut[0:n − 1,
          0:n − 1])
Input:   d[0:n] (dimension sequence for matrices $M_0, M_1, \ldots, M_{n-1}$)
Output:  m[0:n − 1, 0:n − 1]   (m[i, j] = number of multiplications performed in an
                               optimal parenthesization for computing $M_i \cdots M_j$,
                               $0 \le i \le j \le n - 1$)
         FirstCut[0:n − 1, 0:n − 1]   (index of first cut in optimal parenthesization of
                                      $M_i \cdots M_j$, $0 \le i \le j \le n - 1$)
   for i ← 0 to n − 1 do        // initialize M[i, i] to zero
       m[i, i] ← 0
   endfor
   for diag ← 1 to n − 1 do
       for i ← 0 to n − 1 − diag do
           j ← i + diag           // compute $m_{ij}$ according to (9.2.2)
           Min ← m[i + 1, j] + d[i]*d[i + 1]*d[j + 1]
           TempCut ← i
           for k ← i + 1 to j − 1 do
               Temp ← m[i, k] + m[k + 1, j] + d[i]*d[k + 1]*d[j + 1]
               if Temp < Min then
                   Min ← Temp
                   TempCut ← k
               endif
           endfor
       m[i, j] ← Min
       FirstCut[i, j] ← TempCut
       endfor
   endfor
end OptimalParenthesization
```

A simple loop counting shows that the complexity of *OptimalParenthesization* is in $\Theta(n^3)$.

It is now straightforward to write pseudocode for a recursive function *ChainMatrixProd* for computing the chained matrix product $M_0 \cdots M_{n-1}$ using an optimal parenthesization. We assume that the matrices $M_0, \ldots, M_{n-1}$ and the matrix *FirstCut*[0:n − 1, 0:n − 1] are global variables to the procedure *ChainMatrixProd*. The chained matrix product $M_0 \cdots M_{n-1}$ is computed by initially invoking the function *ChainMatrixProd* with $i = 0$ and $j = n - 1$. *ChainMatrixProd* invokes a function *MatrixProd*, which computes the matrix product of two input matrices.

```
function ChainMatrixProd(i, j) recursive
Input:   i, j (indices delimiting matrix chain $M_i, \ldots, M_j$)
         $M_0, \ldots, M_{n-1}$ (global matrices)
         FirstCut[0:n − 1, 0:n − 1] (global matrix computed by
         OptimalParenthesization)
Output:  $M_i \cdots M_j$ (matrix chain)
    if j > i then
        X ← ChainMatrixProd(i, FirstCut[i, j])
        Y ← ChainMatrixProd(FirstCut[i, j] + 1, j)
        return(MatrixProd(X, Y))
    else
        return($M_i$)
    endif
end ChainMatrixProd
```

For the example given in Figure 9.1, invoking *ChainMatrixProd* with M_0, M_1, M_2, M_3 computes the chained matrix product $M_0M_1M_2M_3$ according to the parenthesization $((M_0(M_1M_2))M_3)$.

9.3 Optimal Binary Search Trees

We now use dynamic programming and the Principle of Optimality to generate an algorithm for the problem of finding optimal binary search trees. Given a search tree T and a search element X, the following recursive strategy finds any occurrence of a key X. First, X is compared to the key K associated with the root. If X is found there, we are done. If X is not found, and if X is less than K, then we search the left subtree; otherwise, we search the right subtree.

Consider, for example, the binary search tree given in Figure 9.4 involving the four keys "Ann," "Joe," "Pat," and "Ray". The internal nodes correspond to the successful searches X = "Ann," X = "Joe," X = "Pat," X= "Ray," and the leaf nodes correspond to the unsuccessful searches $X <$ "Ann," "Ann" $< X <$ "Joe," "Joe" $< X <$ "Pat," "Pat" $< X <$ "Ray," "Ray" $< X$. Suppose, for example that X = "Ann." Then *SearchBinSrchTree* makes three comparisons, first comparing X to "Pat," then comparing X to "Joe," and finally comparing X to "Ann." Now suppose that X = "Pete." Then *SearchBinSrchTree* makes two comparisons, first comparing X to "Pat" and then comparing X to "Ray." *SearchBinSrchTree* implicitly branches to the left child of the node containing the key "Ray"—that is, to the leaf (implicit node) corresponding to the interval "Pat" $< X <$ "Ray." Let p_0, p_1, p_2, p_3 be the probability that X = "Ann," X = "Joe," X = "Pat," X = "Ray," respectively, and let q_0, q_1, q_2, q_3, q_4, denote the probability that $X <$ "Ann," "Ann" $< X <$ "Joe," "Joe" $< X <$ "Pat," "Pat" $< X <$ "Ray," "Ray" $< X$, respectively.

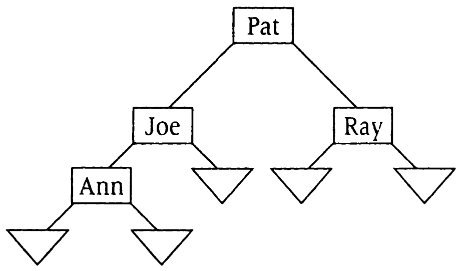


FIGURE 9.4

Search tree with leaf nodes drawn representing unsuccessful searches.

Then, the average number of comparisons made by *SearchBinSrchTree* for the tree T of Figure 9.4 is given by

$$3p_0 + 2p_1 + p_2 + 2p_3 + 3q_0 + 3q_1 + 2q_2 + 2q_3 + 2q_4.$$

Now consider a general binary search tree T whose internal nodes correspond to a fixed set of n keys $K_0, K_1, \dots, K_{n-1}$ with associated probabilities $\mathbf{p} = (p_0, p_1, \dots, p_{n-1})$, and whose $n + 1$ leaf (external) nodes correspond to the $n + 1$ intervals I_0: $X < K_0$, I_1: $K_0 < X < K_1$, ... , I_{n-1}: $K_{n-2} < X < K_{n-1}$, I_n: $X > K_{n-1}$ with associated probabilities $\mathbf{q} = (q_0, q_1, \dots, q_n)$. (When implementing T, the leaf nodes need not actually be included. However, when discussing the average behavior of *SearchBinSrchTree*, it is useful to include them.) We now derive a formula for the average number of comparisons $A(T, n, \mathbf{p}, \mathbf{q})$ made by *SearchBinSrchTree*. Let d_i denote the depth of the internal node corresponding to K_i, $i = 0, \dots, n - 1$. Similarly, let e_i denote the depth of the leaf node corresponding to the interval I_i, $i = 0, 1, \dots, n$. If $X = K_i$, then *SearchBinSrchTree* traverses the path from the root to the internal node corresponding to K_i. Thus, it terminates after performing $d_i + 1$ comparisons. On the other hand, if X lies in I_i, then *SearchBinSrchTree* traverses the path from the root to the leaf node corresponding to I_i and terminates after performing e_i comparisons. Thus, we have

$$A(T, n, \mathbf{p}, \mathbf{q}) = \sum_{i=0}^{n-1} p_i(d_i + 1) + \sum_{i=0}^{n} q_i e_i. \qquad \textbf{(9.3.1)}$$

We now consider the problem of determining an *optimal* binary search tree T, optimal in the sense that T minimizes $A(T, n, \mathbf{p}, \mathbf{q})$ over all binary search trees T. This problem is solved by a complete tree in the case where all the p_i's are equal and all the q_i's are equal. Here we use dynamic programming to solve the problem for general probabilities p_i and q_i. In fact, we solve the slightly more general problem, where we relax the condition that $p_0, \dots, p_{n-1}$ and $q_0, \dots, q_n$ are probabilities by allowing them to be arbitrary nonnegative real numbers. One could regard these numbers as frequencies, as we did when discussing Huffman codes in Chapter 7. That is, we solve the problem

$$\underset{T}{\text{minimize}}\, A(T, n, \mathbf{p}, \mathbf{q}) \qquad \textbf{(9.3.2)}$$

over all binary search trees T of size n, where $p_0, \ldots, p_{n-1}$ and $q_0, \ldots, q_n$ are given fixed nonnegative real numbers. For convenience, we sometimes refer to $A(T, n, \mathbf{p}, \mathbf{q})$ as the *cost* of T. We define $\sigma(\mathbf{p}, \mathbf{q})$ by

$$\sigma(\mathbf{p}, \mathbf{q}) = \sum_{i=0}^{n-1} p_i + \sum_{i=0}^{n} q_i. \qquad \textbf{(9.3.3)}$$

Note that we have removed the probability constraint that $\sigma(\mathbf{p}, \mathbf{q}) = 1$.

As with chained matrix products, we could obtain an optimal search tree by enumerating all binary search trees on the given identifiers and choosing the one with minimum $A(T, n, \mathbf{p}, \mathbf{q})$. However, the number of different binary search trees on n identifiers is the same as the number of binary trees on n nodes, which is given by the nth Catalan number

$$b_n + \frac{1}{n+1}\binom{2n}{n} \in \Omega\left(\frac{4^n}{n^2}\right).$$

Thus, a brute-force algorithm for determining an optimal binary search tree using simple enumeration is computationally infeasible. Fortunately, the Principle of Optimality holds for the optimal binary search tree problem, so we look for a solution using dynamic programming.

Let K_i denote the key associated with the root of T, and let L and R denote the left and right subtrees (of the root) of T, respectively. As we remarked earlier, L is a binary search tree for the keys $K_0, \ldots, K_{i-1}$, and R is a binary search tree for the keys $K_{i+1}, \ldots, K_{n-1}$. For convenience, let $A(T) = A(T, n, \mathbf{p}, \mathbf{q})$, $A(L) = A(L, i, p_0, \ldots, p_{i-1}, q_0, \ldots, q_i)$, and $A(R) = A(R, n-i-1, p_{i+1}, \ldots, p_{n-1}, q_{i+1}, \ldots, q_n)$. Clearly, each node of T that is different from the root corresponds to exactly one node in either L or R. Further, if N is a node in T corresponding to a node N' in L, then the depth of N in T is exactly one greater then the depth of N' in L. A similar result holds if N corresponds to a node in R. Thus, it follows immediately from Formula (9.3.1) that

$$A(T) = A(L) + A(R) + \sigma(\mathbf{p}, \mathbf{q}). \qquad \textbf{(9.3.4)}$$

We now employ recurrence relation (9.3.4) to show that the Principle of Optimality holds for the problem of finding an optimal search tree. Suppose that T is an optimal search tree; that is, T minimizes $A(T)$. We must show that L and R are also optimal search trees. Suppose there exists a binary search tree L' with $i - 1$ nodes involving the keys $K_0, \dots, K_{i-1}$ such that $A(L') < A(L)$. Clearly, the tree T' obtained from T by replacing L with L' is a binary search tree. Further, it follows from Formula (9.3.4) that $A(T') < A(T)$, contradicting the assumption that T is an optimal binary search tree. Hence, L is an optimal binary search tree. By symmetry, R is also an optimal binary search tree, which establishes that the Principle of Optimality holds for the optimal binary search tree problem.

Because the Principle of Optimality holds, when constructing an optimal search tree T, we need only consider binary search trees L and R, both of which are optimal. This observation, together with recurrence relation (9.3.4), is the basis of the following dynamic programming algorithm for constructing an optimal binary search tree. For $i,j \in \{0, \dots, n-1\}$, we let T_{ij} denote an *optimal* search tree involving the consecutive keys $K_i, K_{i+1}, \dots, K_j$, where T_{ij} is the null tree if $i > j$. Thus, if K_k is the root key, then the left subtree L is $T_{i,k-1}$, and the right subtree R is $T_{k+1,j}$ (see Figure 9.5). Moreover, $T = T_{0,n-1}$ is an optimal search tree involving all n keys. For convenience, we define $\sigma(i,j) = \sum_{k=i}^{j} p_k + \sum_{k=i}^{j+1} q_k$.

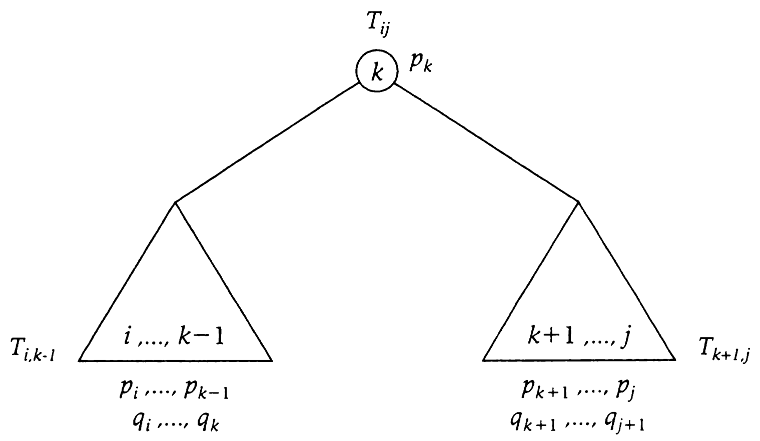


FIGURE 9.5

Principle of Optimality: If T_{ij} is optimal, then L and R must be optimal; that is, $L = T_{i,k-1}$ and $R = T_{k+1,j}$.

We define $A(T_{ij})$ by

$$A(T_{ij}) = A(T_{ij}, j - i + 1, p_i, p_{i+1}, \dots, p_j, q_i, q_{i+1}, \dots, q_{j+1}). \quad \textbf{(9.3.5)}$$

Because the keys are sorted in nondecreasing order, it follows from the Principle of Optimality and (9.3.4) that

$$A(T_{ij}) = \min_k\{A(T_{i,k-1}) + A(T_{k+1,j})\} + \sigma(i,j), \qquad \textbf{(9.3.6)}$$

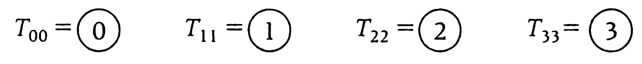
where the minimum is taken over all $k \in \{i, i+1, \dots, j\}$.

Recurrence relation (9.3.6) yields an algorithm for computing an optimal search tree T. The algorithm begins by generating all single-node binary search trees, which are trivially optimal. Namely, $T_{00}, T_{11}, \dots, T_{n-1,n-1}$. Using (9.3.6), the algorithm can then generate optimal search trees $T_{01}, T_{12}, \dots, T_{n-2,n-1}$. In general, at the kth stage in the algorithm, recurrence relation (9.3.6) is applied to construct the optimal search trees $T_{0,k-1}, T_{1,k}, \dots, T_{n-k,n-1}$, using the previously generated optimal search trees as building blocks. Figure 9.6 illustrates the algorithm for a sample instance involving $n = 4$ keys. Note that there are two possible choices for T_{02} in Figure 9.6, each having a minimum cost of 1.1. The tree with the smaller root key was selected.

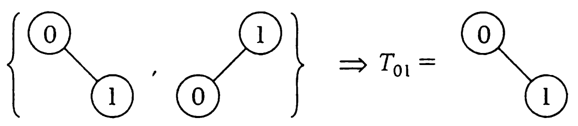
FIGURE 9.6

Action of algorithm to find an optimal binary search tree using dynamic programing.

i	0	1	2	3	4
p_i	.15	.1	.2	.3	
q_i	.05	.05	0	.05	.1

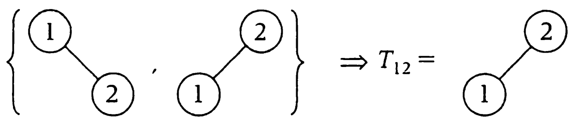


$A(T_{00}) = .25$ $A(T_{11}) = .15$ $A(T_{22}) = .25$ $A(T_{33}) = .45$



$A(T_{01}) = .5$

min {0 + .15 = .15, .25 + 0 = .25} + .35 = .5

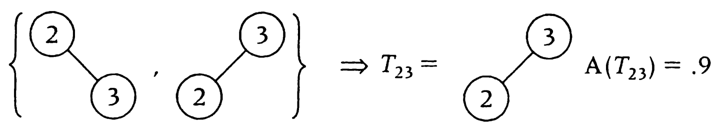


$A(T_{12}) = .55$

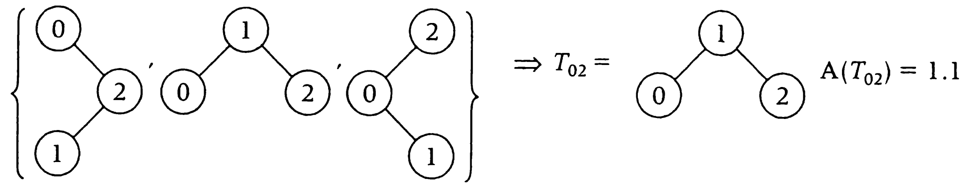
min {0 + .25 = .25, .15 + 0 = .15} + .4 = .55

continued

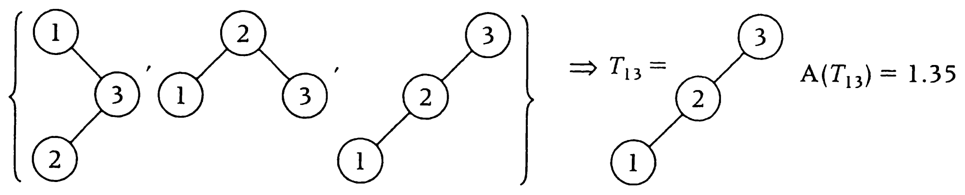
FIGURE 9.6
Continued



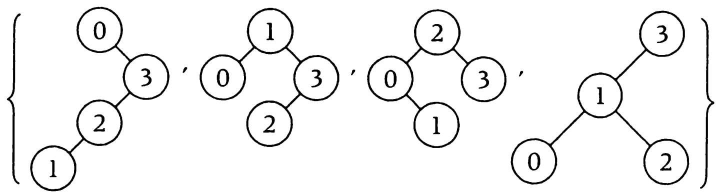
min {0 + .45 = .45, .25 + 0 = .25} + .65 = .9



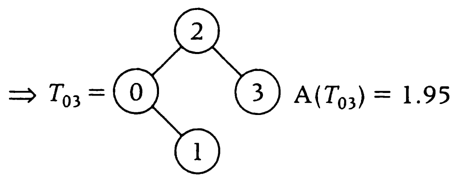
min {0 + .55 = .55, .25 + .25 = .5, .5 + 0 = .5} + .6 = 1.1



min {0 + .9 = .9, .15 + .45 = .6, .55 + 0 = .55} + .8 = 1.35



min {0 + 1.35 = 1.35, .25 + .9 = 1.15, .5 + .45 = .95, 1.1 + 0 = 1.1} + 1 = 1.95



The following pseudocode for the algorithm *OptimalSearchTree* implements the preceding strategy. *OptimalSearchTree* computes the root, *Root*[i, j], of each tree T_{ij}, and the cost, $A[i, j]$, of T_{ij}. An optimal binary search tree T for all the keys (namely, $T_{0,n-1}$) can be easily constructed using recursion from the two-dimensional array *Root*[0:n − 1, 0:n − 1].

```
procedure OptimalSearchTree(P[0:n − 1], Q[0:n], Root[0:n − 1, 0:n − 1], A[0:n − 1,
          0:n − 1])
Input:    P[0:n − 1] (an array of probabilities associated with successful searches)
          Q[0:n] (an array of probabilities associated with unsuccessful searches)
Output:   Root[0:n − 1, 0:n − 1] (Root[i, j] is the key of the root node of $T_{ij}$)
          A[0:n − 1, 0:n − 1] (A[i, j] is the cost $A(T_{ij})$ of $T_{ij}$)
  for i ← 0 to n − 1 do
     Root[i, i] ← i
     Sigma[i, i] ← p[i] + q[i] + q[i + 1]
     A[i, i] ← Sigma[i, i]
  endfor
  for Pass ← 1 to n − 1 do // Pass is one less than the size of the optimal
                                   trees $T_{ij}$ being constructed in the given pass.
     for i ← 0 to n − 1 − Pass do
        j ← i + Pass
                               //Compute $\sigma(p_i, \ldots, p_j, q_i, \ldots, q_{j+1})$
        Sigma[i, j] ← Sigma[i, j − 1] + p[j] + q[j+1]
        Root[i, j] ← i
        Min ← A[i + 1, j]
        for k ← i + 1 to j do
           Sum ← A[i, k − 1] + A[k + 1, j]
           if Sum < Min then
              Min ← Sum
              Root[i, j] ← k
           endif
        endfor
        A[i, j] ← Min + Sigma[i, j]
     endfor
  endfor
end OptimalSearchTree
```

In Figure 9.7, we illustrate the action of *OptimalSearchTree* for the optimal binary search tree just described. For convenience, we change all the probabilities to frequencies, which, in the case we are considering, result by multiplying each probability by 100 to change all the numbers to integers. As we have remarked, working with frequencies instead of probabilities can always be done. In fact, when we are constructing optimal subtrees, it is actually frequencies that we are dealing with instead of probabilities. The optimal subtrees T_{ij} are built starting from the base case T_{ii}, $i = 0, 1, 2, 3$. The figure shows how the tables are built during each pass for $A(T_{ij}) = A[i, j]$,

$Root(T_{ij}) = Root[i, j]$, and $Sigma[i, j] = p_i + \cdots + p_j + q_i + \cdots + q_{j+1} = Sigma[i, j-1] + p_j + q_{j+1}$.

FIGURE 9.7 Building arrays $A[i, j]$, $Root[i, j]$, $Sigma[i, j]$ from inputs $P[0:3]$ = (15,10,20,30) and $Q[0:4]$ = (5,5,0,5,10) to *OptimalSearchTree*.

i	0	1	2	3	4
p_i	15	10	20	30	
q_i	5	5	0	5	10

A[*i*, *j*]

j	0	1	2	3
i				
0	25	*	*	*
1		15	*	*
2			25	*
3				45

Root[*i*, *j*]

j	0	1	2	3
i				
0	0	*	*	*
1		1	*	*
2			2	*
3				3

Sigma[*i*, *j*]

j	0	1	2	3
i				
0	25	*	*	*
1		15	*	*
2			25	*
3				45

j	0	1	2	3
i				
0	25	50	*	*
1		15	55	*
2			25	90
3				45

j	0	1	2	3
i				
0	0	0	*	*
1		1	2	*
2			2	3
3				3

j	0	1	2	3
i				
0	25	35	*	*
1		15	40	*
2			25	65
3				45

j	0	1	2	3
i				
0	25	50	110	*
1		15	55	135
2			25	90
3				45

j	0	1	2	3
i				
0	0	0	1	*
1		1	2	3
2			2	3
3				3

j	0	1	2	3
i				
0	25	35	60	*
1		15	40	80
2			25	65
3				45

j	0	1	2	3
i				
0	25	50	110	195
1		15	55	135
2			25	90
3				45

j	0	1	2	3
i				
0	0	0	1	2
1		1	2	3
2			2	3
3				3

j	0	1	2	3
i				
0	25	35	60	100
1		15	40	80
2			25	65
3				45

Because *OptimalSearchTree* does the same amount of work for any input $P[0:n-1]$ and $Q[0:n]$, the best-case, worst-case, and average complexities are equal. Clearly, the number of additions made in computing *Sum* has the same order as the total number of additions made by *OptimalSearchTree*. Therefore, we choose the addition made in computing *Sum* as the basic operation. Since *Pass* varies from 1 to $n - 1$, i varies from 0 to $n - 1 - Pass$, and k varies from $i + 1$ to

i + *Pass*, it follows that the total number of additions made in computing *Sum* is given by

$$\begin{aligned}
&\sum_{t=1}^{n-1}\sum_{i=0}^{n-1-t}\sum_{k=i+1}^{i+t} 1 \\
&= \sum_{t=1}^{n-1}(n-t)t \\
&= n\sum_{t=1}^{n-1}t - \sum_{t=1}^{n-1}t^2 \\
&= n\left[(n-1)\frac{n}{2}\right] - \left[(n-1)n\frac{(2n-1)}{6}\right] \in \Theta(n^3).
\end{aligned}$$

9.4 Longest Common Subsequence

In this section, we consider the problem of determining how close two character strings are to one another. For example, a spell checker might compare a text string created on a word processor with pattern strings from a stored dictionary. If there is no exact match between the text string and any pattern string, then the spell checker offers several alternative pattern strings that are fairly close, in some sense, to the text string. As another example, a forensic scientist might compare two DNA strings to measure how close they match.

We can measure the closeness of two strings in several ways. In Chapter 20, we will consider one important measure called the *edit distance*, which is commonly used by search engines to find approximate matchings for a user-entered text string for which an exact match cannot be found. The edit distance is also used by spell checkers. Roughly speaking, the edit distance between two strings is the minimum number of changes that need to be made (adding, deleting, or changing characters) to transform one string to the other.

In this section, we consider another closeness measure, the longest common subsequence (LCS) contained in a text string and a particular pattern string. Computing either the LCS or the edit distance is an optimization problem that satisfies the Principle of Optimality and can be solved using dynamic programming. However, the solution to the LCS problem is easier to understand because it has a simpler recurrence relation, which we now describe.

Suppose $T = T_0T_1 \cdots T_{n-1}$ is a text string that we want to compare to a pattern string $P = P_0 P_1 \cdots P_{m-1}$, where we assume that the characters in each string are drawn from some fixed alphabet A. A *subsequence* of T is a string of the form $T_{i_1}T_{i_2} \cdots T_{i_k}$, where $0 \leq i_1 < i_2 < \cdots < i_k \leq n - 1$. Note that a *substring* of T is a special case of a subsequence of T in which the subscripts making up the subsequence increase by one. For example, consider the pattern string "Cincinnati" and the text string "Cincinatti" (a common misspelling). You can easily check that the longest common subsequence of the pattern string and the text string has length 9 (just one less than the common length of both strings), whereas it takes two changes to transform the text string to the pattern string (so that the edit distance between the two strings is two).

We now describe a dynamic programming algorithm to determine the length of the longest common subsequence of T and P. For simplicity of notation, we assume that the strings are stored in arrays $T[0{:}n - 1]$ and $P[0{:}m - 1]$, respectively. For integers i and j, we define $LCS[i,j]$ to be the length of the longest common subsequence of the substrings $T[0{:}i - 1]$ and $P[0{:}j - 1]$ (so that $LCS[n, m]$ is the length of the longest common subsequence of T and P). For convenience, we set $LCS[i,j] = 0$ if $i = 0$ or $j = 0$ (corresponding to empty strings). Note that $LCS[1, 1] = 1$ if $T[0] = P[0]$; otherwise, $LCS[1, 1] = 0$. This initial condition is actually a special case of the following recurrence relation for $LCS[i,j]$:

$$\begin{aligned}&LCS[i,j] = LCS[i-1,j-1] + 1 \quad \text{if } T[i-1] = P[j-1];\\ &\text{otherwise, } LCS[i,j] = \max\{LCS[i,j-1], LCS[i-1,j]\}.\end{aligned} \qquad \textbf{(9.4.1)}$$

To verify recurrence relation (9.4.1), note first that if $T[i - 1] \neq P[j - 1]$, then a longest common subsequence of $T[0{:}i - 1]$ and $P[0{:}j - 1]$ might end in $T[i - 1]$ or $P[j - 1]$, but certainly not both. In other words, if $T[i - 1] \neq P[j - 1]$, then a longest common subsequence of $T[0{:}i - 1]$ and $P[0{:}j - 1]$ must be drawn from either the pair $T[0{:}i - 2]$ and $P[0{:}j - 1]$ or from the pair $T[0{:}i - 1]$ and $P[0{:}j - 2]$. Moreover, such a longest common subsequence must be a longest common subsequence of the pair of substrings from which it is drawn (that is, the principle of optimality holds). This verifies that

$$LCS[i,j] = \max\{LCS[i,j-1], LCS[i-1,j]\} \text{ if } T[i-1] \neq P[j-1]. \quad \textbf{(9.4.2)}$$

On the other hand, if $T[i - 1] = P[j - 1] = C$, then a longest common subsequence must end either at $T[i - 1]$ in $T[0{:}i - 1]$ or at $P[j - 1]$ in $P[0, j - 1]$, or both,; otherwise, by adding the common value C to a given subsequence, we would increase the length of the subsequence by 1. Also, if the last term of a longest common subsequence ends at an index $k < i - 1$ in $T[0{:}i - 1]$ (so that $T[k] = C$), then clearly we achieve an equivalent longest common subsequence by swapping $T[i - 1]$ for $T[k]$ in the subsequence. By a similar argument involving $P[0{:}j - 1]$, when $T[i - 1] = P[j - 1]$, we can assume without loss of generality that a longest common subsequence in $T[0{:}i - 1]$ and $P[0{:}j - 1]$ ends at $T[i - 1]$ and $P[j - 1]$. However, then removing these end points from the subsequence clearly must result in a longest common subsequence in $T[0{:}i-2]$ and $P[0{:}j-2]$, respectively (that is, the principle of optimality again holds). It follows that

$$LCS[i,j] = LCS[i-1, j-1] + 1 \text{ if } T[i-1] = P[j-1], \qquad \mathbf{(9.4.3)}$$

which, together with Formula (9.4.2), completes the verification of Formula (9.4.1).

The following algorithm is the straightforward row-by-row computation of the array $LCS[0{:}n, 0{:}m]$ based on the recurrence (9.4.1).

```
procedure LongestCommonSubseq(T[0:n - 1], P[0:m - 1], LCS[0:n, 0:m])
Input:    T[0:n - 1], P[0:m - 1] (strings)
Output:   LCS[0:n, 0:m] (array such that LCS[i, j] is length of the longest common
          subsequence of T[0:i - 1] and P[0:j - 1] )
   for i ← 0 to n do       // initialize for boundary conditions
      LCS[i, 0] ← 0
   endfor
   for j ← 0 to m do       // initialize for boundary conditions
      LCS[0, j] ← 0
   endfor
   for i ← 1 to n do       // compute the row index by i of LCS[0:n, 0:m]
      for j ← 1 to m do // compute LCS[i, j] using (9.4.1)
         if T[i - 1] = P[j - 1] then
            LCS[i, j] ← LCS[i - 1, j - 1] + 1
         else
            LCS[i, j] ← max(LCS[i, j - 1], LCS[i - 1, j])
         endif
      endfor
   endfor
end LongestCommonSubseq
```

Using the comparison of text characters as our basic operation, we see that *LongestCommonSubseq* has complexity in $O(nm)$, which is a rather dramatic improvement over the exponential complexity $\Theta(2^n m)$ brute-force algorithm that would examine each of the 2^n subsquences of $T[0{:}n-1]$ and determine the longest subsequence that also occurs in $P[0{:}m-1]$.

Figure 9.8 shows the array $LCS[0{:}8, 0{:}11]$ output by *LongestCommonSubseq* for $T[0{:}7]$ = "usbeeune" and $P[0{:}10]$ = "subsequence".

Note that *LongestCommonSubseq* determines the length of the longest common subsequence of $T[0{:}n-1]$ and $P[0{:}m-1]$ but does not output the actual subsequence itself. In the previous problem of finding the optimal paranthesization of a chained matrix product, in addition to knowing the minimum number of multiplications required, it was also important to determine the actual paranthesization that did the job. Thus, we needed to compute the array *FirstCut*$[0{:}n-1, 0{:}n-1]$ to be able to construct the optimal paranthesization. Similarly, in the optimal binary search tree problem, in addition to knowing the average search complexity of the optimal search tree, it was important to determine the optimal search tree itself. Thus, we kept track of the key *Root*$[i, j]$ in the root of the optimal binary search tree containing the keys $K_i < \cdots < K_j$. However, in the LCS problem, knowing the actual common subsequence is not as important as knowing its length. For example, in a process like spell checking, the subsequence is not as important as its length. Typically, to correct a misspelled word in the text, the spell checker displays a list of pattern strings that share subsequences exceeding a threshold length (depending on the length of the strings), as opposed to exhibiting common subsequences. Nevertheless, it is interesting that a longest common subsequence can be determined just from the array $LCS[0{:}n-1, 0{:}m-1]$ (and $T[0{:}n-1]$ and $P[0{:}m-1]$) without the need to maintain any additional information.

One way we can generate a longest common subsequence is to start at the bottom-right corner (n, m) of the array *LCS* and work our way backward through the array to build the subsequence in reverse order. The moves are dictated by looking at how we get the value assigned to a given position when we used (9.4.1) to build the array *LCS*. More precisely, if we are currently at position (i, j) in *LCS*, and $T[i-1] = P[j-1]$, then this common value is appended to the beginning of the string already generated (starting with the null string), and we move to position $(i-1, j-1)$ in *LCS*. On the other hand, if $T[i-1] \neq P[j-1]$, then we move to position $(i-1, j)$ or $(i, j-1)$, depending on whether $LCS[i-1, j]$ is greater than $LCS[i, j-1]$. When $LCS[i-1, j]$ is equal to $LCS[i, j-1]$, either move can be made. In the latter case, the two different choices might not only generate different longest common subsequences but also yield different longest common strings corresponding to these subsequences. For example, Figure 9.8a

FIGURE 9.8(a)

The matrix *LCS*[0:8, 0:11] for the strings *T*[0:7] = "usbeeune" and *P*[0:10] = "subsequence", with the path in *LCS* generating the longest common string "sbeune" using the move-left rule.

			s	u	b	s	e	q	u	e	n	c	e
	LCS	0	1	2	3	4	5	6	7	8	9	10	11
	0	0	0	0	0	0	0	0	0	0	0	0	0
u	1	0	0	1	1	1	1	1	1	1	1	1	1
s	2	0	1	1	1	2	2	2	2	2	2	2	2
b	3	0	1	1	2	2	2	2	2	2	2	2	2
e	4	0	1	1	2	2	3	3	3	3	3	3	3
e	5	0	1	1	2	2	3	3	3	4	4	4	4
u	6	0	1	2	2	2	3	3	4	4	4	4	4
n	7	0	1	2	2	2	3	3	4	4	5	5	5
e	8	0	1	2	2	2	3	3	4	5	5	5	6

FIGURE 9.8(b)

The path in the matrix *LCS* generating the longest common string "useene" using the move-up rule.

			s	u	b	s	e	q	u	e	n	c	e
	LCS	0	1	2	3	4	5	6	7	8	9	10	11
	0	0	0	0	0	0	0	0	0	0	0	0	0
u	1	0	0	1	1	1	1	1	1	1	1	1	1
s	2	0	1	1	1	2	2	2	2	2	2	2	2
b	3	0	1	1	2	2	2	2	2	2	2	2	2
e	4	0	1	1	2	2	3	3	3	3	3	3	3
e	5	0	1	1	2	2	3	3	3	4	4	4	4
u	6	0	1	2	2	2	3	3	4	4	4	4	4
n	7	0	1	2	2	2	3	3	4	4	5	5	5
e	8	0	1	2	2	2	3	3	4	5	5	5	6

shows the path generated using the move-left rule, which requires us to move to position $(i-1, j)$ when $T[i-1] \neq P[j-1]$, whereas Figure 9.8b shows the path resulting from always moving up to position $(i, j-1)$. The darker shaded positions (i, j) in these paths correspond to where $T[i-1] = P[j-1]$. These two paths yield the longest common strings "sbeune" and "useune", respectively. When generating a path in *LCS*, we obtain a longest common subsequence when we reach a position where $LCS[i, j] = 0$.

9.5 Closing Remarks

In subsequent chapters, we will use dynamic programming to solve a number of important problems. For example, in Chapter 12, we will discuss Floyd's dynamic programming solution to the all-pairs shortest-path problem in weighted directed graphs. In Chapter 20 we will use dynamic programming to solve the edit distance version of the approximate string matching problem. Dynamic programming, because it is based on a bottom-up resolution of recurrence relations, is usually amenable to straightforward level-by-level parallelization. However, this straightforward parallelization usually does not result in optimal speedup, and more clever parallel algorithms, sometimes based on finding recurrences better suited to parallelization, must be sought. We will see such an example in Chapter 16 for computing shortest paths.

References and Suggestions for Further Reading

Bellman, R. E. *Dynamic Programming.* Princeton, NJ: Princeton University Press, 1957. The first systematic study of dynamic programming.

Two other classic references on dynamic programming are:

Bellman, R. E., and S. E. Dreyfus. *Applied Dynamic Programming.* Princeton, NJ: Princeton University Press, 1962.

Nemhauser, G. *Introduction to Dynamic Programming.* New York: Wiley, 1966.

EXERCISES

Section 9.1 Optimization Problems and the Principle of Optimality

9.1 Suppose the matrix $C[0{:}n-1, 0{:}n-1]$ contains the cost of $C[i, j]$ of flying directly from airport i to airport j. Consider the problem of finding the cheapest flight from i to j where we may fly to as many intermediate airports as desired. Verify that the Principle of Optimality holds for the minimum-cost flight. Derive a recurrence relation based on the Principle of Optimality.

9.2 Does the Principle of Optimality hold for the costliest trips (no revisiting of airports, please)? Discuss.

9.3 Does the Principle of Optimality hold for coin changing? Discuss with various interpretations of the *Combine* function.

Section 9.2 Optimal Parenthesization for Computing a Chained Matrix Product

9.4 Given the matrix product $M_0M_1 \cdots M_{n-1}$ and a 2-tree T with n leaves, show that there is a unique parenthesization P such that $T = T(P)$.

9.5 Give pseudocode for *ParenthesizeRec* and analyze its complexity.

9.6 Show that the complexity of *OptimalParenthesization* is in $\Theta(n^3)$.

9.7 Using *OptimalParenthesization*, find an optimal parenthesization for the chained product of five matrices with dimensions 6×7, 7×8, 8×3, 3×10, and 10×6.

9.8 Write a program implementing *OptimalParenthesization* and run it for some sample inputs.

Section 9.3 Optimal Binary Search Trees

9.9 Use dynamic programming to find an optimal binary search tree for the following probabilities, where we assume that the search key is in the search tree; that is, $q_i = 0$, $i = 0, \ldots, n$:

keys	i	0	1	2	3
probabilities	p_i	.4	.3	.2	.1

9.10 Use dynamic programming to find an optimal search tree for the following probabilities:

i	0	1	2	3	4	5
p_i	.2	.1	.2	.05	.05	
q_i	.05	0	.25	0	.1	0

9.11 a. Design and analyze a recursive algorithm that computes an optimal binary search tree T for all the keys from the two-dimensional array *Root*$[0{:}n-1, 0{:}n-1]$ generated by *OptimalSearchTree*.

b. Show the action of your algorithm from part (a) for the instance given in Exercise 9.10.

9.12 a. Give a set of probabilities $p_0, \ldots, p_{n-1}$ (assume a successful search so that $q_0 = q_1 = \cdots = q_n = 0$), such that a completely right-skewed search tree T (the left child of every node is **null**) is an optimal search tree with respect to these probabilities.

b. More generally, prove the following induction on n: If T is *any* given binary search tree with n nodes, then there exists a set of probabilities $p_0, \ldots, p_{n-1}$ such that T is the unique optimal binary search tree with respect to these probabilities.

Section 9.4 Longest Common Subsequence

9.13 Show the array $LCS[0{:}9, 0{:}10]$ that is built by *LongestCommonSubseq* for $T[0{:}8]$ = "alligator" and $P[0{:}9]$ = "algorithms".

9.14 By following various paths in the array $LCS[0{:}8, 0{:}11]$ given in Figure 9.8, find all the longest common subsequences of the strings "usbeeune" and "subsequence".

9.15 Design and analyze an algorithm that generates all longest common subsequences given the input array $LCS[0{:}n-1, 0{:}m-1]$.

9.16 Write a program that implements *LongestCommonSubseq*, and run it for some sample inputs.

Additional Problems

9.17 Consider a sequence of n distinct integers. Design and analyze a dynamic programming algorithm to find the length of the longest increasing subsequence. For example, consider the sequence:

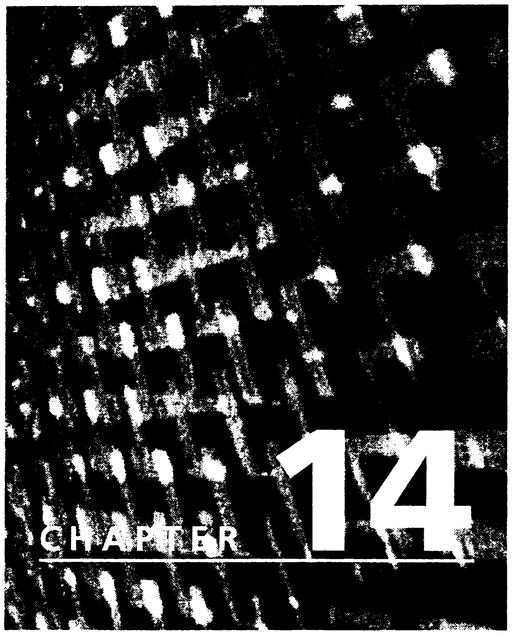
45 23 9 3 99 108 76 12 77 16 18 4

A longest increasing subsequence is 3 12 16 18, having length 4.

9.18 The 0/1 knapsack problem is NP-hard when the input is measured in binary. However, when the input is measured in unary (see the discussion at the end of Section 7.3 of Chapter 7), dynamic programming can be used to find a polynomial-complexity solution. Design and analyze a dynamic programming solution to the 0/1 knapsack problem, with positive integer capacity and weights, which is quadratic in $C + n$, where C is the capacity and n is the number of objects. *Hint:* Let $V[i, j]$ denote the maximum value that can be placed in a knapsack of capacity j using objects drawn from $\{b_0, \dots, b_{i-1}\}$. Use the principle of optimality to find a recurrence relation for $V[i, j]$.

9.19 Design and analyze a dynamic programming solution to the coin-changing problem under similar assumptions to that in the previous exercise.

9.20 Given n integers, the partition problem is to find a bipartition of the integers into two subsets having the same sum or determine that no such bipartition exists. Design and analyze a dynamic programming algorithm for solving the partition problem.



CHAPTER 14

MATCHING AND NETWORK FLOW ALGORITHMS

Finding matchings and maximum flows in graphs and networks are two fundamental problems with myriad practical applications. We begin this chapter with an algorithm for finding a perfect matching in a bipartite graph and a maximum-weighted perfect matching in a weighted complete bipartite graph. Some of the techniques we use to solving the matching problems, such as finding augmenting paths when constructing a perfect matching in a bipartite graph, can be generalized to apply to the maximum flow problem.

There are many natural interpretations of flows in networks, such as fluid flow through a network of pipelines, data flow through a computer network, traffic flow through a network of highways, current flow through an electrical network, and so forth. Usually, each edge in a network has a certain flow capacity (a *capacitated* network), and the problem arises of finding the maximum flow subject to the capacity constraints of the edges. One of the most celebrated theorems about capacitated networks is the max-flow min-cut theorem of Ford and Fulkerson. In this chapter, we present this theorem and an associated algorithm for finding a maximum flow in a capacitated network. Finding a maximum flow is useful in solving the well-known marriage problem, which is equivalent to the problem of finding a perfect matching in a bipartite graph.

14.1 Perfect Matchings in Bipartite Graphs

An independent set of edges, or *matching,* in a graph G is a set of edges that are pairwise vertex disjoint. A matching that spans the vertices is called a *perfect matching*. In this section, we discuss an algorithm known as the Hungarian algorithm for finding a perfect matching or determining it does not exist. We also discuss the Kuhn-Munkres algorithm, which employs the Hungarian algorithm to find a maximum-weight perfect matching in an edge-weighted complete bipartite graph. We begin by discussing the marriage problem, which is a colorful interpretation of the problem of finding a perfect matching in a bipartite graph. In Chapter 24, we will give a probabilistic algorithm for determining whether or not a bipartite graph has a perfect matching.

14.1.1 The Marriage Problem

Suppose we have a set of n boys $b_1, b_2, \ldots, b_n$ and a set of n girls $g_1, g_2, \ldots g_n$, where each boy knows some of the girls. The classical *marriage problem* is to determine a necessary and sufficient condition so that each boy can marry a girl that he knows (no two boys can marry the same girl). More formally, the marriage problem is finding the conditions under which the permutation σ: $\{1, 2, \ldots, n\} \rightarrow \{1, 2, \ldots, n\}$ exists such that boy b_i knows girl $g_{\sigma(i)}$, $i = 1, 2, \ldots, n$. If such a permutation σ exists, then the corresponding set of matches $P_\sigma = \{\{b_i, g_{\sigma(i)}\}| \ i = 1, 2, \ldots, n\}$ is called a perfect matching.

Let K_i denote the set of girls that boy b_i knows, $i = 1, \ldots, n$. For example, suppose that $n = 5$ and the sets K_i, $i = 1, \ldots, 5$, are given by

$$K_1 = \{g_1, g_2, g_4, g_5\}, K_2 = \{g_1, g_3\}, K_3 = \{g_2, g_3\}, K_4 = (g_3, g_4, g_5\}, K_5 = \{g_3\}.$$

Then, the following set of pairs determines a perfect matching:

$$\{b_1, g_4\}, \{b_2, g_1\}, \{b_3, g_2\}, \{b_4, g_5\}, \{b_5, g_3\}.$$

The marriage problem is naturally modeled using a bipartite graph G, where one set X of the vertex bipartition consists of the set of n boys and the other set Y consists of the set of n girls. A vertex b_i in X is joined to a vertex g_j in Y whenever the boy b_i knows the girl g_j (see Figure 14.1). The marriage problem then becomes determining a necessary and sufficient condition for a bipartite graph to contain a perfect matching—that is, a set of n edges no two of which have a vertex in common.

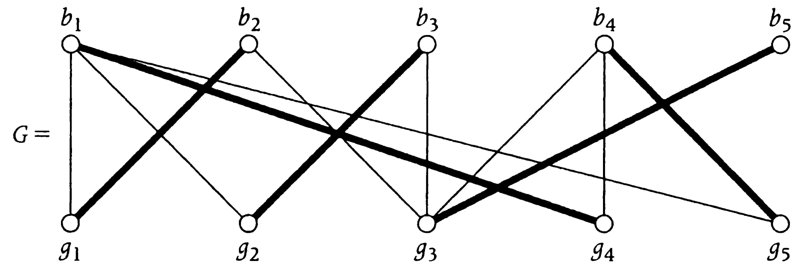


FIGURE 14.1 The bipartite graph G corresponding to sets $K_1 = \{g_1, g_2, g_4, g_5\}$, $K_2 = \{g_1, g_3\}$, $K_3 = \{g_2, g_3\}$, $K_4 = \{g_3, g_4, g_5\}$, $K_5 = \{g_3\}$. G contains the perfect matching $\{b_1, g_4\}$, $\{b_2, g_1\}$, $\{b_3, g_2\}$, $\{b_4, g_5\}$, $\{b_5, g_3\}$.

Suppose in the previous example that K_1 is replaced with the set

$$K_1' = \{g_1, g_2, g_3\}.$$

Because the four boys b_1, b_2, b_3, b_5 collectively know only the three girls g_1, g_2, g_3, a perfect matching cannot exist. More generally, if there exists a set of k boys who collectively know strictly less than k girls, then a perfect matching does not exit. Surprisingly, the converse is also true, by a theorem of Hall, which states that a solution to the marriage problem exists if and only if every set of k boys collectively knows at least k girls, $k \in \{1, 2, \dots, n\}$.

We can restate Hall's theorem for bipartite graphs as follows: For S, a set of vertices of G, let $\Gamma(S)$ denote the set of all vertices that are adjacent to those in S; that is,

$$\Gamma(S) = \{v \in V \mid \text{there exists a vertex } u \text{ in } S \text{ such that } uv \in E\}. \qquad \textbf{(14.1.1)}$$

Theorem 14.1.1 **Hall's Theorem**

A bipartite graph with vertex bipartition $V = X \cup Y$ contains a perfect matching if and only if, for every subset S of X,

$$|S| \le |\Gamma(S)|. \qquad \textbf{(14.1.2)}$$

PROOF

First, suppose that there exists a perfect matching M in G. For $u \in V$, let $M(u)$ denote the *mate* of u in M—that is, the unique vertex v such that $uv \in M$. For $S \subseteq X$, let

$$M(S) = \{M(x) \mid x \in S\}. \qquad \textbf{(14.1.3)}$$

Clearly, $M(S) \subseteq \Gamma(S)$. Hence,

$$|\Gamma(S)| \geq |M(S)| = |S|. \qquad ■$$

We complete the proof of Theorem 14.1.1 in the next subsection by presenting a "matchmaker" algorithm, called Hungarian, which accepts an initial matching M (possibly empty) and repeatedly augments M until either a perfect matching is generated or a set $S \subseteq X$ is found such that $|\Gamma(S)| < |S|$.

14.1.2 The Hungarian Algorithm

The Hungarian algorithm uses the notion of an M-alternating path. Given a matching M, we say that a vertex v of G is *M-matched* if it is incident with an edge of M; otherwise, we say that v is *M-unmatched.* An *M-alternating path* is one where the edges alternately belong to $E(G) \setminus M$ and M. An *M-augmenting path* is an alternating path in which the initial and terminal vertices are both M-unmatched.

Key Fact

Given an M-augmenting path P, the size of the matching M can be increased by 1 by removing the edges of M belonging to P, and adding to M the remaining edges of P.

The larger matching described in this key fact can be expressed as the *symmetric difference* $M \oplus E(P)$ of M and the edges $E(P)$ of P; that is

$$M \oplus E(P) = (M \cup E(P)) \setminus (M \cap E(P)). \qquad \textbf{(14.1.4)}$$

The Hungarian algorithm searches for augmenting paths. To find an M-augmenting path in G, the algorithm grows a tree T, called an *M-alternating tree*, having the following properties:

1. The root r of T is an M-unmatched vertex belonging to X.
2. For each odd integer i less than the depth of T, each edge of T joining a vertex at level i to a vertex at level $i + 1$ belongs to M.
3. The leaf nodes of T all belong to X.

An M-alternating tree is illustrated in Figure 14.2. Clearly, all the paths in an M-alternating tree T are M-alternating paths, and X_T and Y_T denote the subset of vertices of X and Y, respectively, belonging to T.

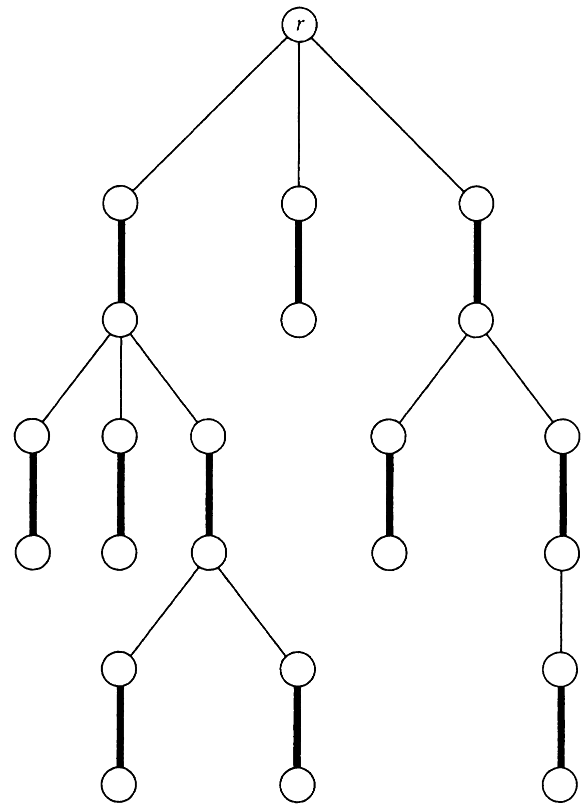


FIGURE 14.2
An M-alternating tree T rooted at vertex r.

The proofs of the next three lemmas are straightforward and left as exercises.

Lemma 14.1.2 If M is a matching and P is an M-augmenting path, then $M \oplus E(P)$ is a matching of size one greater than M. □

Lemma 14.1.3 If T is an M-alternating tree, then

$$|X_T| = |Y_T| + 1. \qquad \textbf{(14.1.5)} \quad \square$$

Lemma 14.1.4 Given an M-alternating tree T, if there exists a vertex $x \in X_T$ that is adjacent to an M-unmatched vertex y (not in T), then the path from the root r of T to x, together with the edge xy, determine an M-augmenting path. □

If all the vertices in X are M-matched, then M is a perfect matching. In this case, the Hungarian algorithm returns the perfect matching M and terminates. In the case where the vertices in X are not all matched, the Hungarian algorithm looks for an augmenting path by growing an M-alternating tree T. The tree T may be initialized to consist of the single vertex r, where r is chosen arbitrarily from the set of unmatched vertices in X. At each stage of growing the tree T, we encounter one of the following three cases:

1. $\Gamma(X_T) - Y_T$ is empty. (Action: algorithm terminates.)
   Then by Lemma 14.1.3, $|\Gamma(X_T)| = |Y_T| = |X_T| - 1$. The Hungarian algorithm then returns $S = X_T$ and terminates, because by Theorem 14.1.1, no perfect matching exists.
   In the next two cases, $\Gamma(X_T) - Y_T$ is nonempty, and we choose y to be any vertex in $\Gamma(X_T) - Y_T$ and x to be any vertex in X_T adjacent to y.
2. Vertex y is matched. (Action: T is augmented.)
   Then we augment T to obtain a new M-alternating tree by adding the edges xy and yz, where z is the mate of y in M.
3. Vertex y is unmatched. (Action: M is augmented.)
   Then an augmenting path P in T from r to x together with the edge xy has been found. We then replace M by the augmented matching $M \oplus E(P)$ containing one more edge.

In case 3, we either have found a perfect matching or we look for another augmenting path by growing another M-alternating tree rooted at a new unmatched vertex in X. Clearly, M can be augmented at most n times. Because at each stage the alternating tree T can be grown in time $O(n^2)$, the worst-case complexity of the following procedure *Hungarian* is $O(n^3)$. Pseudocode for *Hungarian* follows.

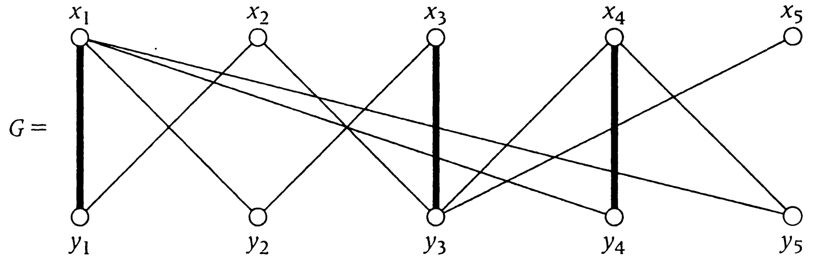
```
procedure Hungarian(G, M, S)
Input:   G (a bipartite graph with vertex bipartition (X, Y))
         M (an initial matching, possibly empty)
Output:  M (a perfect matching if one exists)
         S (a set of vertices with the property that $|\Gamma(S)| < |S|$ if no perfect matching
         exists)
   AugmentingM ← .true.
   while AugmentingM do
      if all the vertices in X are M-matched then      //M is a perfect matching
         AugmentingM ← .false.
      else                                              //Grow M-alternating tree T
         r ← any M-unmatched vertex in X
         T ← tree consisting of the single vertex r
         GrowingTree ← .true.
```

```
            while GrowingTree do
                if $\Gamma(X_T) = Y_T$ then
                    $S \leftarrow X_T$                    //$|\Gamma(S)| < |S|$
                    GrowingTree ← .false.
                    AugmentingM ← .false.
                else
                    $y \leftarrow$ any vertex in $\Gamma(X_T) - Y_T$
                    $x \leftarrow$ any vertex in $X_T$ adjacent to adjacent to $y$
                    if $y$ is $M$-matched then        //augment $T$
                        $z \leftarrow M(y)$               //$z$ is the mate of $y$ in $M$
                        $T \leftarrow T \cup xy \cup yz$
                    else                           //an $M$-augmenting path has been
                                                     found
                        $P \leftarrow$ path in $T$ from $r$ to $x$ together with the edge $xy$
                        $M = M \oplus E(P)$           //augment $M$
                        GrowingTree ← .false.
                    endif
                endif
            endwhile
        endif
    endwhile
end Hungarian
```

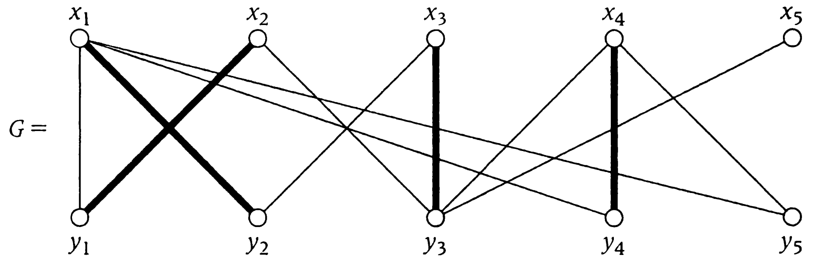
Figure 14.3 illustrates the action of procedure *Hungarian* for a sample bipartite graph. For definiteness, in Figure 14.3 we show the perfect matching generated by always choosing vertices in order of their vertex number. More precisely, the phrase "any ... vertex" occurring in three statements in the pseudocode for *Hungarian* is replaced by the phrase "the ... vertex of smallest index" when generating the perfect matching in Figure 14.3. We leave the intermediate steps in finding the augmenting paths as an exercise.

FIGURE 14.3

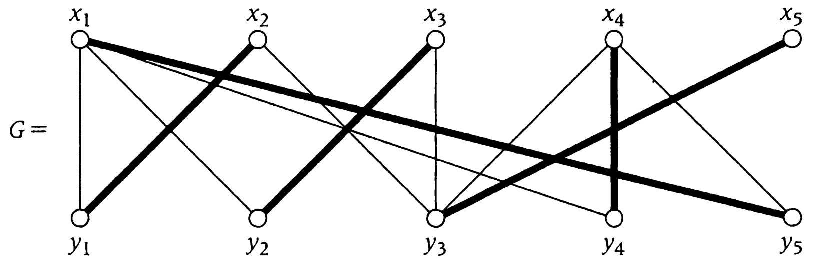
Action of procedure *Hungarian* for a sample bipartite graph with initial matching $\{x_1, y_1\}$, $\{x_3, y_3\}$, $\{x_4, y_4\}$. Vertices are considered in the order of their indices.



Sample bipartie graph G and initial matching $\{x_1,y_1\}$, $\{x_3,y_3\}$, $\{x_4,y_4\}$



A first M-augmenting path P: $x_2y_1x_1y_2$, found by *Hungarian* with starting M-unmatched vertex x_2 yielding augmented matching $M \oplus E(P) = \{x_1y_2, x_2y_1, x_3y_3, x_4y_4\}$



A second M-augmenting path P: $x_5y_3x_3y_2x_1y_5$, found by *Hungarian* with starting M-unmatched vertex x_5 yielding augmented perfect matching $M \oplus E(P) = \{x_1y_5, x_2y_1, x_3y_2, x_4y_4, x_5y_3\}$

14.1.3 Maximum Perfect Matching in a Weighted Bipartite Graph

Suppose n workers $x_1, x_2, \ldots, x_n$ are to be assigned to n jobs $y_1, y_2, \ldots, y_n$, where each worker is qualified to perform any of the jobs. Associated with each worker-job pair (x_i, y_j) is a weight ω_{ij} measuring how effectively worker x_i can perform job y_j. The natural problem arises of finding assignments of workers to jobs so that the total effectiveness of each workers is optimized. This problem can be modeled using a weighted complete bipartite graph $G = (V, E)$, with vertex bipartition $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_n\}$. Each edge x_iy_j of G is assigned the weight ω_{ij}, $i, j \in \{1, \ldots, n\}$. Clearly, a perfect matching in G corresponds to an assignment of each worker to a job so that no two workers are assigned the same job. We define the *weight* of a perfect matching M, denoted by $\omega(M)$, to be the sum of the weights of its edges.

$$\omega(M) = \sum_{e \in M} \omega(e). \tag{14.1.6}$$

A *maximum perfect matching* is a perfect matching of maximum weight over all perfect matchings of G. Because G is complete, any permutation π of $\{1, 2, \ldots, n\}$ determines a perfect matching $M_\pi = \{x_i y_{\pi(i)} \mid i \in \{1, \ldots, n\}\}$ and conversely. Thus, a brute-force algorithm that enumerates all $n!$ perfect matchings and chooses one of maximum weight is hopelessly inefficient.

We now describe an $O(n^3)$ algorithm due to Kuhn and Munkres for finding a maximum perfect matching in a weighted complete bipartite graph. The Kuhn-Munkres algorithm uses the Hungarian algorithm, together with the notion of a feasible vertex weighting.

DEFINITION 14.1.1 A *feasible vertex weighting* is a mapping ϕ from V to the real numbers such that for each edge $xy \in E$ (each $x \in X$ and $y \in Y$),

$$\phi(x) + \phi(y) \geq \omega(xy). \tag{14.1.7}$$

Any sufficiently large ϕ is a feasible vertex weighting. For example, the following vertex weighting is feasible:

$$\phi(v) = \begin{cases} \max \quad \{\omega(vy) \mid vy \in E(G)\} & \text{if } v \in X, \\ 0 & \text{otherwise} \end{cases} \tag{14.1.8}$$

The following proposition is easily verified.

Proposition 14.1.5 Let ϕ be any feasible vertex weighting and M any perfect matching of G. Then,

$$\omega(M) \leq \sum_{v \in V} \phi(v). \tag{14.1.9}$$ □

Given a vertex weighting ϕ, the *equality subgraph* $G_\phi = (V_\phi, E_\phi)$ is a subgraph of G such that $V_\phi = V(G)$ and E_ϕ consists of all the edges xy such that $\omega(xy) = \phi(x) + \phi(y)$. Note that it is possible for some of the vertices of G to be isolated vertices in G_ϕ.

proposition 14.1.6 Let ϕ be any feasible vertex weighting. If the equality subgraph G_ϕ contains a perfect matching M, then M is a maximum perfect matching in G. □

Observe that if M is a perfect matching in G_ϕ, then $\omega(M) = \Sigma_{v \in V}\phi(v)$. Hence, Proposition 14.1.6 follows from Proposition 14.1.5.

Starting with an initial feasible vertex weighting, such as the one given by Formula (14.1.8), the Kuhn-Munkres algorithm applies the Hungarian algorithm to the subgraph G_ϕ. If a perfect matching M_ϕ is found, then by Proposition 14.1.5, M_ϕ is a maximum perfect matching in G. On the other hand, suppose the Hungarian algorithm terminates by finding a matching M and an M-alternating tree T such that $\Gamma(X_T) - Y_T$ is empty. Then we have found a set $S = X_T$ such that $|\Gamma_\phi(S)| = |S| - 1 < |S|$. Instead of terminating at this point, we replace ϕ with a new feasible weighting ϕ' and continue the Hungarian algorithm in the equality graph $G_{\phi'}$ of ϕ'. The new feasible weighting ϕ' is constructed as follows. Set

$$\varepsilon = \min\{\phi(x) + \phi(y) - \omega(xy) \mid x \in S, y \in Y - \Gamma_\phi(S)\}. \quad \textbf{(14.1.10)}$$

For each $v \in V(G)$, the weighting ϕ' is defined by

$$\phi'(v) = \begin{cases} \phi(v) - \varepsilon & v \in S, \\ \phi(v) + \varepsilon & v \in \Gamma_\phi(S), \\ \phi(v) & \text{otherwise.} \end{cases} \quad \textbf{(14.1.11)}$$

It is easily verified that the vertex weighting ϕ' given by (14.1.11) is a feasible vertex weighting. Moreover, the following key fact allows us to continue growing the M-alternating tree T.

Key Fact

The equality subgraph $G_{\phi'}$ contains the M-alternating tree T. Further, $G_{\phi'}$ contains at least one edge $\{x, y\}$, where $x \in S$ and $y \in Y - \Gamma_{\phi'}(S)$. Since y is unmatched by M, the path in $G_{\phi'}$ consisting of the path in T from its root to x together with edge xy is an M-augmenting path.

Thus, by continuing the Hungarian algorithm in $G_{\phi'}$, the matching M will be augmented by at least one edge. After at most $n/2$ steps in which ϕ is replaced with ϕ', we obtain a perfect matching M and a feasible weighting ϕ^* such that M is contained in the equality subgraph G_{ϕ^*} of ϕ^*. By Proposition 14.1.6, such a perfect matching will necessarily be a maximum perfect matching in G.

In the following pseudocode for the Kuhn-Munkres algorithm, we call the procedure *Hungarian2*, which is identical to *Hungarian* except that the alternating tree T is added as an input/output parameter.

```
procedure KuhnMunkres($G$, $\phi$, $M$)
Input:   $G$ (a weighted complete bipartite graph with vertex bipartition
                $(X, Y)$, $|X| = |Y| = n$)
         $\phi$ (a positive edge weighting of $G$)
Output:  $M$ (a maximum perfect matching)
   $\phi \leftarrow$ any feasible vertex weighting        //in particular, the one given by
                                               Formula 14.1.8
   $G_\phi \leftarrow$ equality subgraph for $\phi$
   $M \leftarrow$ any matching in $G_\phi$                //in particular, M can be chosen to be
                                               empty
   PerfectMatchingFound $\leftarrow$ .false.
   while .not. PerfectMatchingFound do
      Hungarian2($G_\phi$, $M$, $S$, $T$)
      if $M$ is a perfect matching then
         PerfectMatchingFound $\leftarrow$ .true.
      else
         $\varepsilon \leftarrow \min\{\phi(x) + \phi(y) - \omega(xy) \mid x \in S, y \in Y - \Gamma_\phi(S)\}$
         for all $x \in S$ do
            $\phi(x) = \phi(x) - \varepsilon$
         endfor
         for all $y \in \Gamma_\phi(S)$ do
            $\phi(y) = \phi(y) + \varepsilon$
         endfor
      endif
   endwhile
end KuhnMunkres
```

We leave it as an exercise to show that the worst-case complexity of procedure *KuhnMunkres* is $O(n^3)$

14.2 Maximum Flows in Capacitated Networks

The problem of finding a maximum flow in a network from a source s to a sink t, where each link in the network has a given capacity, and the dual problem of finding a minimum cut in such a network, is a classical optimization problem with many applications. Our study of flows in networks begins by formally defining the notion of a flow in a digraph (an *uncapacitated* network) and establishing some elementary results about flows.

REMARK

The theory of flows in networks modeled on digraphs includes as an important special case flows modeled on graphs by associating with the graph its combinatorially equivalent (symmetric) digraph.

14.2.1 Flows in Digraphs

Let $D = (E, V)$ be a digraph with vertex set V and directed edge set E. A *real weighting* ω of the edges of D is a mapping from E to the set $\mathbb{R}$ of real numbers. We refer to $\omega(e)$ as the *ω-weight* of edge e. For $v \in V$, we let $\sigma_{\text{in}}(\omega, v)$ and $\sigma_{\text{out}}(\omega, v)$ denote the sum of the ω-weights over all the edges having head v and tail v, respectively, so that

$$\sigma_{\text{in}}(\omega, v) = \sum_{uv \in E} \omega(uv),$$

$$\sigma_{\text{out}}(\omega, v) = \sum_{vw \in E} \omega(vw).$$

By convention, $\sigma_{\text{in}}(\omega, v) = 0$ if there are no edges having head v. Similarly, $\sigma_{\text{out}}(\omega, v) = 0$ if there are no edges having tail v.

The following proposition is easily verified.

Proposition 14.2.1 Given any real weighting ω of the edges in a digraph D,

$$\sum_{v \in V} (\sigma_{\text{in}}(\omega, v) - \sigma_{\text{out}}(\omega, v)) = 0. \qquad \square$$

Now suppose we are given two vertices s and t such that there are no edges having head s or tail t. A *flow* f from s to t is a weighting of the edges such that

$$\sigma_{\text{in}}(f, v) = \sigma_{\text{out}}(f, v), \quad \forall v \in V \setminus \{s, t\}. \qquad \textbf{(14.2.1)}$$

Given any vertex v in D, we refer to $\sigma_{\text{in}}(f, v)$ and $\sigma_{\text{out}}(f, v)$ as the *flow into* v and the *flow out of* v, respectively. Formula (14.2.1) is called a *flow conservation equation*. The *value of flow* f, denoted by $val(f)$, is defined to be the flow out of s. It follows easily from Proposition 14.2.1 that the flow out of s equals the flow into t. Hence,

$$val(f) = \sigma_{\text{out}}(f, s) = \sigma_{\text{in}}(f, t). \qquad \textbf{(14.2.2)}$$

A *unit flow* is a flow f such that $val(f) = 1$. Figure 14.4 illustrates a flow f of value 55 on a sample digraph D.

The following proposition is easily verified.

Proposition 14.2.2 The set of all flows is closed under linear combinations; that is, for any flows f_1 and f_2 and real numbers λ_1 and λ_2, $\lambda_1 f_1 + \lambda_2 f_2$ is a flow. Moreover,

$$val(\lambda_1 f_1 + \lambda_2 f_2) = \lambda_1 val(f_1) + \lambda_2 val(f_2). \qquad \square$$

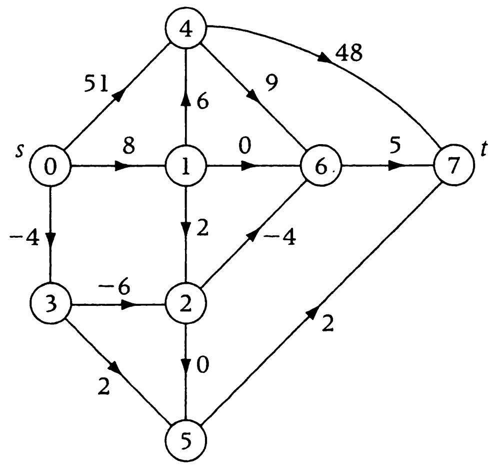


FIGURE 14.4
A flow of value 55.

Now consider a directed path $P = se_1u_1e_2u_2 \dots u_{p-1}e_pt$ from s to t (not necessarily a simple path). Associated with P is the flow χ_P from s to t given by

$$\chi_P(e) = \begin{cases} 1 & e \in E(P) \\ 0 & e \notin E(P), \quad \forall e \in E. \end{cases}$$

We call χ_P the *characteristic flow* of P. Note that the characteristic flow is a *unit* flow.

In the next section, we compute maximum flows in a capacitated network by using the notion of a semipath, which is a path where the orientation of the edges is ignored. More precisely, a *semipath* S from s to t is an alternating sequence of vertices and edges $se_1u_1e_2u_2 \dots u_{p-1}e_pt$ such that either e_i has tail u_{i-1} and head u_i (e_i is a *forward edge of* S) or e_i has tail u_i and head u_{i-1} (e_i is *backward edge of* S), $i = 1, \dots, p$, where $u_0 = s$ and $u_p = t$. Associated with semipath S is the flow χ_S, called the *characteristic flow* of S, given by

$$\chi_S(e) = \begin{cases} 1 & e \text{ is a forward edge of } S, \\ -1 & e \text{ is a backward edge of } S, \\ 0 & \text{otherwise,} \qquad \forall e \in E. \end{cases} \qquad \textbf{(14.2.3)}$$

It is easily verified that χ_S is a unit flow.

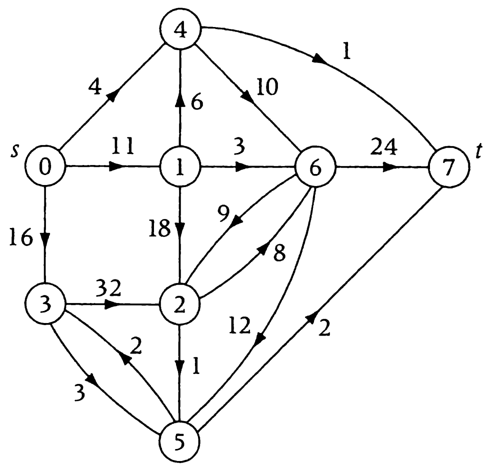
14.2.2 Flows in Capacitated Networks

A *capacitated network* N (sometimes called a *transportation* or *flow* network) consists of the 4-tuple (D, s, t, c), where $D = (V, E)$ is a digraph; s and t are two distinguished vertices of D called the *source* and *sink*, respectively; and c is a positive real weighting of the edges, called the *capacity weighting* of D. We assume that all the edges incident with s are directed out of s, and all the edges incident with t are directed into t. We refer to $c(e)$ as the *capacity* of edge e.

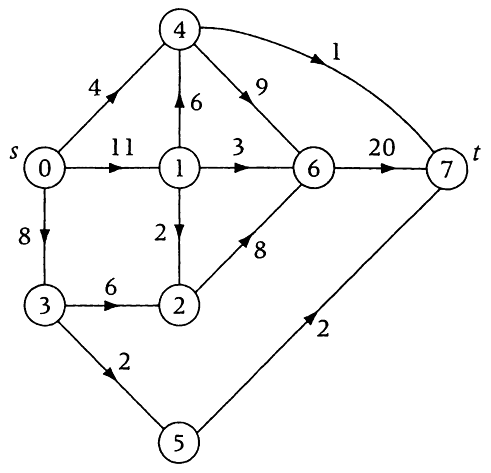
A *flow f in a capacitated network N* is a flow in D from s to t such that for each $e \in E$, $f(e)$ is nonnegative and does not exceed the capacity of e; that is

$$0 \le f(e) \le c(e), \quad \forall e \in E.$$

Figure 14.5 shows a flow of value 23 in a sample capacitated network N on eight vertices. In Figure 14.5 and all subsequent figures, when illustrating a flow f, all edges e such that $f(e) = 0$ are omitted.



A capacitated network N



A flow f in N

FIGURE 14.5
A sample capacitated network N and a flow f having value $val(f) = 23$.

A *maximum (value) flow* is a flow f in N whose value is maximum over all flows in N. A naive attempt to finding a maximum flow might proceed as follows. First we find a path P_1 from s to t in N using an algorithm such as a breadth-first search. Let λ_1 denote the minimum capacity among all the edges of P_1. Then $f = \lambda_1\chi_{P_1}$ is a flow in N having value λ_1. Next, we adjust the capacities of the edges of N by subtracting λ_1 from each edge belonging to P_1 and deleting all the edges of N whose capacity becomes zero. We then find a path P_2 (if one exists) in the new network. We now augment the flow f by $\lambda_2\chi_{P_2}$, where λ_2 is the minimum capacity (in the new network) among all the edges of P_2. The current flow $f = \lambda_1\chi_{P_1} + \lambda_2\chi_{P_2}$ has value $\lambda_1 + \lambda_2$. Again, we subtract λ_2 from every edge belonging to P_2 and delete all the edges whose capacity becomes zero. Continuing in this way, we find paths $P_3, \ldots, P_k$ and associated real numbers $\lambda_3, \ldots, \lambda_k$, respectively, until no paths are left from s to t in the final network.

We illustrate the action of our naive attempt in Figure 14.6 for a sample capacitated network. Unfortunately, the final flow f attained has value 14, whereas the maximum flow shown in Figure 14.7 has value 24. However, any flow generated by our naive attempt does have a maximality property—namely, if g is any flow different from f, then $g(e) < f(e)$ for some edge e.

FIGURE 14.6

Action of naive attempt to find maximum flow for a sample capacitated network N, yielding flow $f = 10\chi_{P_1} + \chi_{P_2} + 3\chi_{P_3}$ having value 14. This flow is suboptimal because a maximum flow f^* shown in Figure 14.7 has value 24.

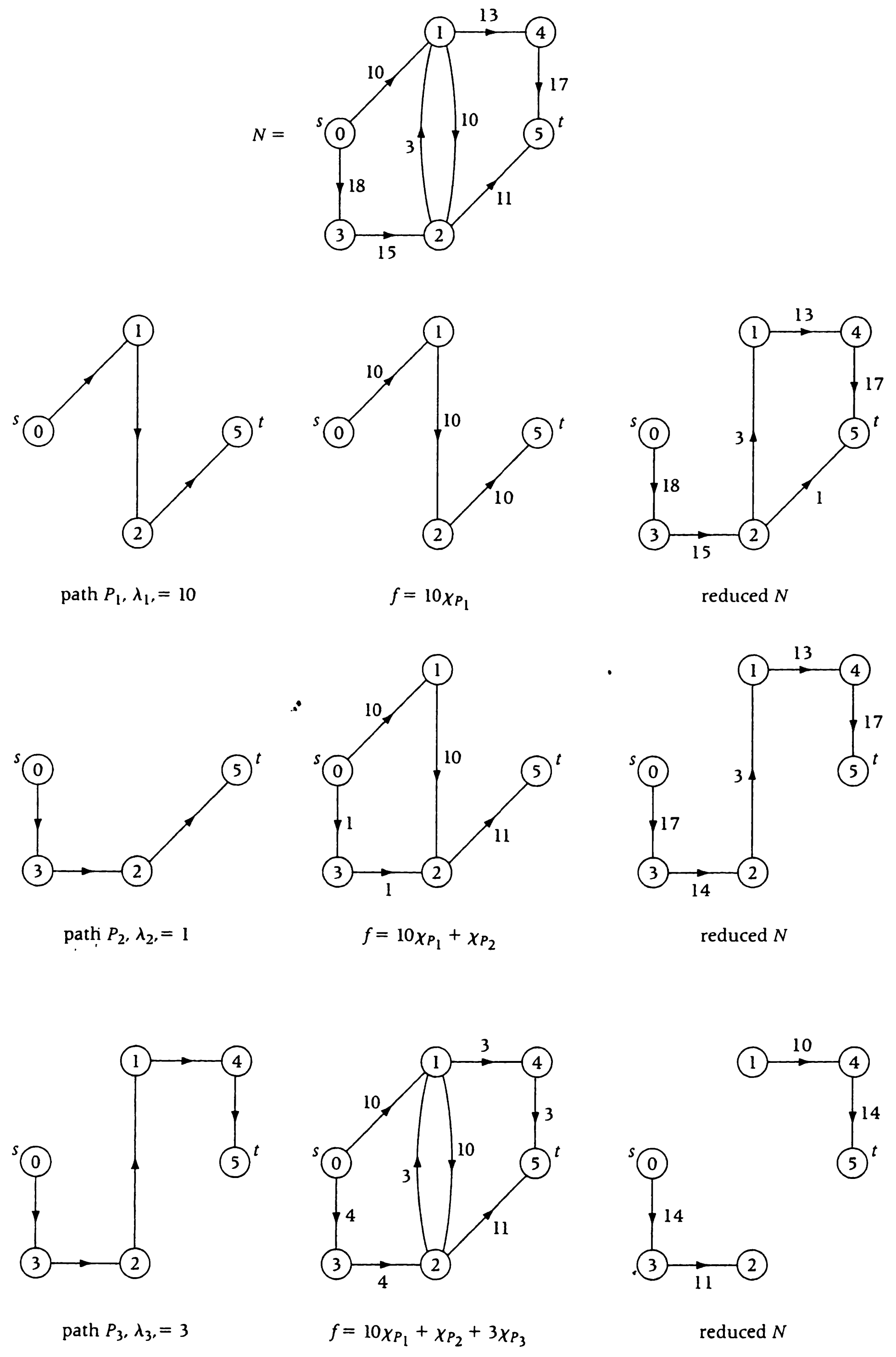
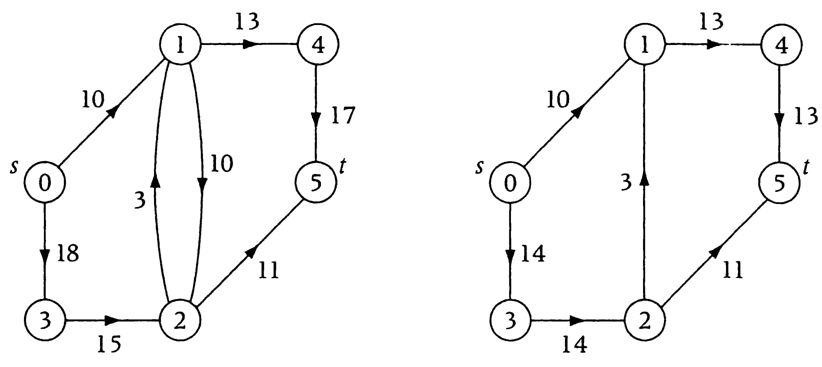

N =
path P_1, λ_1 = 10
$f = 10\chi_{P_1}$
reduced N
path P_2, λ_2 = 1
$f = 10\chi_{P_1} + \chi_{P_2}$
reduced N
path P_3, λ_3 = 3
$f = 10\chi_{P_1} + \chi_{P_2} + 3\chi_{P_3}$
reduced N

FIGURE 14.7

A maximum flow f^* having value 24 in the capacitated network N of Figure 14.5.



A capacitated network N A maximum flow f^* on N

The flow f of Figure 14.6 generated by our naive algorithm is a dead end in our search for a maximum flow in the sense that we can no longer augment f by simply finding directed paths from s to t. The following key fact using the notion of a semipath allows us to continue to augment the flow f.

Key Fact

Given a semipath with no forward edge used to capacity and nonzero flow in each backward edge, we can continue to augment f by adding flow to forward edges while removing flow from backward edges.

14.2.3 Finding an Augmenting Semipath

The *residual capacity* with respect to a flow f is $c(e) - f(e)$. An edge e is *f-saturated* if the residual capacity is zero; otherwise, edge e is *f-unsaturated.* A semipath S is an *f-augmenting semipath* if every forward edge of S is f-unsaturated and $f(e) > 0$ for every backward edge e of S. For $e \in E(S)$, we let

$$c_f(S, e) = \begin{cases} c(e) - f(e) & e \text{ is a forward edge of } S, \\ f(e) & e \text{ is a backward edge of } S. \end{cases}$$

Let $c_f(S)$ denote the minimum value of $c_f(S, e)$ among all the edges of S; that is,

$$c_f(S) = \min\{c_f(S, e) \mid e \in E(S)\}.$$

Now consider the edge weighting $\hat{f}$ given by

$$\hat{f} = f + c_f(S)\chi_S, \qquad \textbf{(14.2.4)}$$

where χ_S is the characteristic flow of S given by Formula (14.2.3). Thus, for each $e \in E(G)$,

$$\hat{f}(e) = \begin{cases} f(e) + c_f(S) & e \text{ is a forward edge of } S, \\ f(e) - c_f(S) & e \text{ is a backward edge of } S, \\ f(e) & \text{otherwise.} \end{cases}$$

Proposition 14.2.2 implies that the edge weighting $\hat{f}$ given by Formula (14.2.4) is a flow in the digraph D. Further, it is immediate from the definition of $c_f(S)$ that for all $e \in E$,

$$0 \leq \hat{f}(e) \leq c(e).$$

Hence, $\hat{f}$ is a flow in the capacitated network N. Further, since χ_S is a unit flow, it follows from Proposition 14.2.2 that

$$val(\hat{f}) = val(f) + c_f(S),$$

so that $\hat{f}$ has a strictly greater value than f.

A semipath S and the value $c_f(S)$ can be computed by finding a path from s to t in the *f-derived network* N_f constructed from the network N and the flow f. The network N_f is obtained by starting with vertex set V and adding edges as follows. For each edge uv of N that is f-unsaturated, we add an edge uv to N_f having weight $c(uv) - f(uv)$. For each edge uv of N such that $f(uv) > 0$, we add an edge vu to N_f having weight $f(uv)$. When uv and vu are both edges of N, it is possible for N_f to have two edges joining vertex u to vertex v. In our search for a maximum flow, allowing such pairs of edges does not present a problem. In fact, we can even eliminate the possibility of such pairs in N_f by replacing f with a new flow f obtained by subtracting min $\{f(uv), f(vu)\}$ from both $f(uv)$ and $f(vu)$ so that one of them becomes equal to zero. The latter operation does not affect the value of the flow f. Note that if $val(f) = 0$, then $N_f = N$. The f-derived network N_f for a sample network N and flow f is illustrated in Figure 14.8.

For P, a path from s to t in N_f, we define the weight $\mu(P)$ to be the minimum weight over all the edges of P; that is,

$$\mu(P) = \min\{w(e) \mid e \in E(P)\}.$$

The following proposition follows immediately from the definitions of N_f and the f-augmenting semipath S.

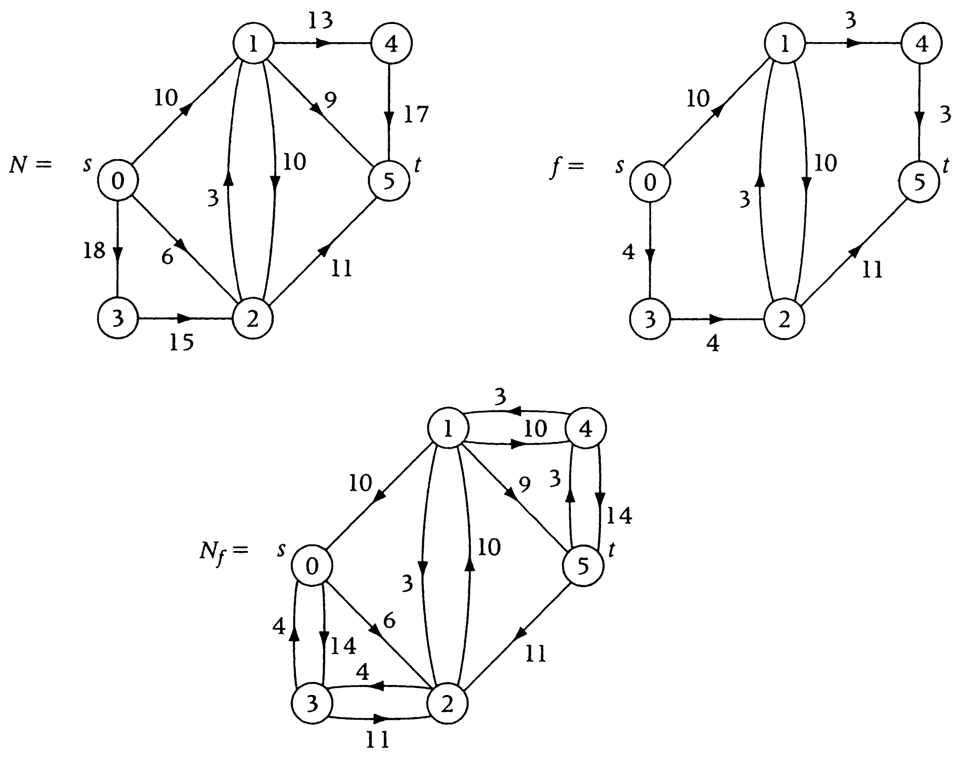


FIGURE 14.8 The f-derived network N_f for a sample network N and flow f.

Proposition 14.2.3 If P is a path from s to t in N_f, then the corresponding semipath S in N is an f-augmenting semipath. Further,

$$c_f(S) = \mu(P). \qquad \square$$

It follows from Formula (14.2.4) that if there exists an f-augmenting semipath S, then we can find a flow whose value is strictly greater than f. The Ford-Fulkerson algorithm is based on the fact that the converse is also true; that is, *if there is no f-augmenting semipath, then f is a maximum flow.* To prove this important result, we introduce the concept of a cut.

14.2.4 Bounding Flow Values by Capacities of Cuts

Consider a bipartition of the vertex set V into two disjoint sets X and Y. We denote this bipartition by (X, Y). The cut associated with the bipartition (X, Y), denoted $cut(X, Y)$, is defined by

$$cut(X, Y) = \{xy \in E \mid x \in X, y \in Y\}.$$

We say a set of edges Γ is a *cut* if $\Gamma = cut(X, Y)$ for some bipartition (X, Y) of V. For $u, v \in V$, if $u \in X$ and $v \in Y$, then we say that Γ *separates u and v*. Unless otherwise stated, we assume that Γ separates the source s from the sink t. The *capacity of* Γ, denoted by $cap(\Gamma)$, is the sum of the capacities of all the edges in Γ; that is,

$$cap(\Gamma) = \sum_{e \in \Gamma} c(e).$$

A *minimum capacity cut* (or simply *minimum cut*) is a cut Γ whose capacity is minimum over all cuts separating s and t.

Because deleting all the edges of a cut disconnects the source s from the sink t, intuitively we would expect that the value of any flow f is not greater than the capacity of any cut Γ. The following proposition affirms this intuition.

Proposition 14.2.4 Let f be any flow from s to t in N, and let $\Gamma = (X, Y)$ be any cut separating s and t. Then, the value of f is bounded above by the capacity of Γ; that is,

$$val(f) \le cap(\Gamma).$$

PROOF

Given a nonnegative weighting ω of E, we extend ω to a mapping of all $V \times V$ as follows:

$$\omega(u, v) = \begin{cases} \omega(uv) & uv \in E, \\ 0 & \text{otherwise.} \end{cases}$$

For $A, B \subseteq V$, let

$$\omega(A, B) = \sum_{a \in A} \sum_{b \in B} \omega(a, b). \qquad \textbf{(14.2.5)}$$

In the notation of Formula (14.2.5), the flow conservation equation (14.2.1) can be rewritten as follows:

$$f(u, V) - f(V, u) = 0, \quad u \in V - \{s,t\}. \qquad \textbf{(14.2.6)}$$

Formula (14.2.2) then becomes

$$f(s, V) = f(V, t) = val(f). \qquad \textbf{(14.2.7)}$$

It follows immediately from Formulas (14.2.6) and (14.2.7) that

$$f(X, V) - f(V, X) = val(f). \tag{14.2.8}$$

Clearly, for U, which is any subset of V, we have

$$\begin{aligned} f(U, V) &= f(U, X) + f(U, Y), \\ f(V, U) &= f(X, U) + f(Y, U). \end{aligned} \tag{14.2.9}$$

Substituting $U = X$ in both parts of Formula (14.2.9), subtracting the second part from the first, and employing Formula (14.2.8) yields

$$f(X, Y) - f(Y, X) = f(X, V) - f(V, X) = val(f). \tag{14.2.10}$$

Using Formula (14.2.10), we have

$$\begin{aligned} val(f) &= f(X, Y) - f(Y, X) \\ &\le f(X, Y) \\ &= \sum_{xy \in \Gamma} f(xy) \\ &\le \sum_{xy \in \Gamma} c(xy) \quad (\text{since } f(e) \le c(e) \text{ for all } e \in E) \\ &= cap(\Gamma). \end{aligned}$$

■

Corollary 14.2.5 If f is a flow from s to t and Γ is a cut separating s and t such that $val(f) = cap(\Gamma)$, then f is a maximum flow and Γ is a minimum cut.

PROOF
Let f' be any flow from s to t and let Γ' be any cut separating s and t. Then by Proposition 14.2.4, we have

$$\begin{aligned} val(f') &\le cap(\Gamma) = val(f), \\ cap(\Gamma') &\ge val(f) = cap(\Gamma). \end{aligned}$$

■

Corollary 14.2.5 states a condition guaranteeing that f is a maximum flow and Γ is a minimum cut, but it does not tell us whether such a flow f and cut Γ actually exist. In fact, a seminal result in flow theory is that such a flow f and cut Γ always exist.

Theorem 14.2.6 **Max-Flow Min-Cut Theorem**
Let $N = (D, c, s, t)$ be a capacitated network. The maximum value of a flow from s to t equals the minimum capacity of a cut separating s and t. □

To prove Theorem 14.2.6, we use Corollary 14.2.5 to establish it is sufficient to exhibit a flow f and cut Γ such that $val(f) = cap(\Gamma)$. In the next subsection, we give an algorithm for computing such a flow f and cut Γ.

14.2.5 The Ford-Fulkerson Algorithm

The following procedure computes a maximum flow and minimum cut by repeatedly augmenting the current flow f using an f-augmenting semipath.

```
procedure FordFulkerson($N, f, \Gamma$)
  Input:  $N = (D, s, t, c)$ (a capacitated network)
Output:  $f$ (maximum flow)
         $\Gamma$(minimum cut)
    $f \leftarrow 0$
    $N_f \leftarrow N$
    while there is a directed path from $s$ to $t$ in the $f$-derived network $N_f$ do
        $P \leftarrow$ a path from $s$ to $t$ in $N_f$
        $S \leftarrow$ the $f$-augmenting semipath in $N$ corresponding to path $P$ in $N_f$
        $c_f(S) \leftarrow \mu(P)$          //$\mu(P)$ is the minimum weight over all the edges of $P$
        $f \leftarrow f + c_f(S)\chi_S$     //augment $f$
        update $N_f$
    endwhile
    $X \leftarrow$ set of vertices that are accessible from $s$ in $N_f$         //$s \in X$
    $Y \leftarrow$ set of vertices that are not accessible from $s$ in $N_f$     //$t \in Y$
    $\Gamma \leftarrow cut(X, Y)$
end FordFulkerson
```

The correctness of procedure *FordFulkerson* is established with the aid of the following two lemmas, whose proofs are left as exercises.

Lemma 14.2.7 Let f and (X, Y) be the flow and vertex bipartition, respectively, generated by procedure *FordFulkerson*. Then every edge xy of the cut $\Gamma = cut(X, Y)$ is saturated; that is,

$$f(xy) = c(xy).$$ □

Lemma 14.2.8 Let f and (X, Y) be the flow and vertex bipartition, respectively, generated by procedure *FordFulkerson*. Then for each edge $yx \in E(D)$, where $y \in Y$ and $x \in X$, we have

$$f(yx) = 0. \qquad \square$$

Lemmas 14.2.7 and 14.2.8 imply that

$$f(X, Y) = cap(\Gamma),$$
$$f(Y, X) = 0.$$

Thus, by Formula (14.2.10) we have

$$val(f) = f(X, Y) - f(Y, X) = cap(\Gamma).$$

By Corollary 14.2.5, f is a maximum flow and Γ is a minimum cut, which completes the correctness proof of procedure *FordFulkerson*.

The Ford-Fulkerson augmenting semipath algorithm is a general method for computing a maximum flow and minimum cut. In procedure *FordFulkerson*, we did not specify how the path P is generated in the derived network N_f at each stage. In general, there may be many such paths, and the efficiency of *Ford-Fulkerson* is dependent on which augmenting semipath S is chosen at each stage.

For a poor choice of augmenting semipaths S, procedure *FordFulkerson* may never terminate. However, in the case when the capacities on the edges are all integers, procedure *FordFulkerson* terminates after having performed at most $val(f)$ iterations of the **while** loop. Because each iteration can be performed in time $O(m)$, the worst-case complexity $W(n, m)$ of procedure *FordFulkerson* belongs to $O(m*val(f))$. Since $val(f)$ depends on the capacities of the edges, it can be arbitrarily large. It is not hard to find examples showing that for a poor choice of augmenting semipaths S, $W(n, m)$ can also be arbitrarily large.

Edmonds and Karp showed that a good choice for the augmenting semipath S at each stage of procedure *FordFulkerson* is the shortest one (with a minimum number of edges) over all such semipaths. At each stage, a shortest augmenting semipath S can be found by performing a breadth-first search of the f-derived network N_f to find a shortest path P from s to t. The Edmonds-Karp algorithm has worst-case complexity $W(n, m) \in O(nm^2)$. (The proof of this complexity result is beyond the scope of this book.) The Edmonds-Karp algorithm is illustrated in Figure 14.9 for a sample flow network N having eight vertices and 17 edges. The shortest path generated at each step is indicated with a dotted (as opposed to solid) line.

FIGURE 14.9

Action of the Edmonds-Karp algorithm for a sample capacitated network N.

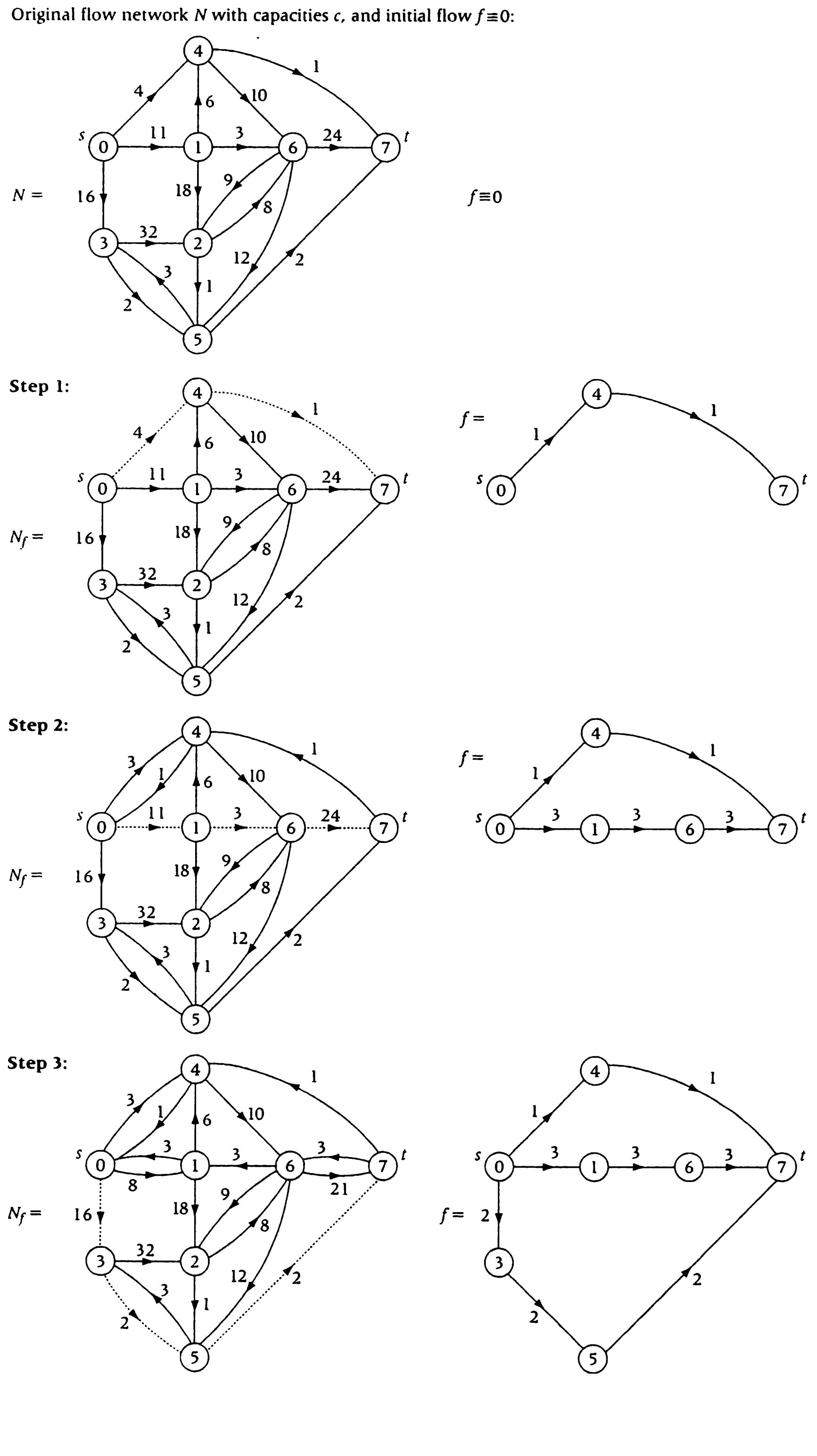
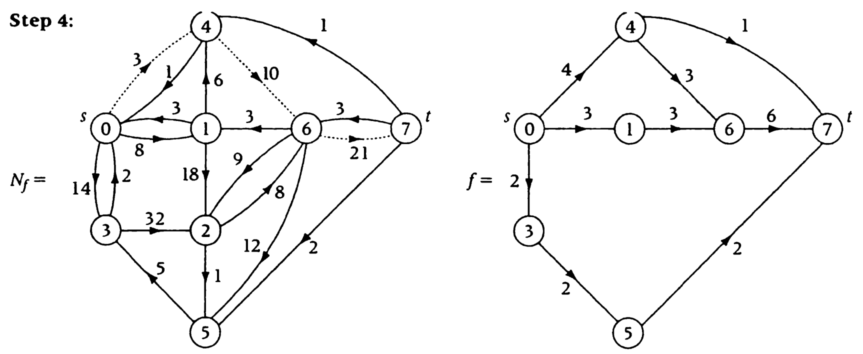
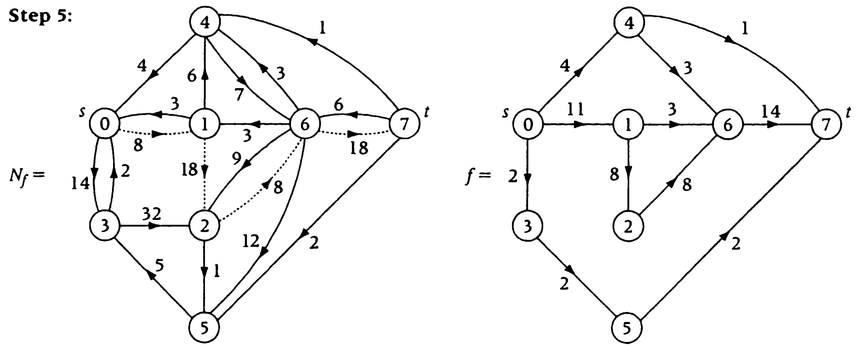

Original flow network N with capacities c, and initial flow f≡0:
N =
f≡0
Step 1:
N_f =
f =
Step 2:
N_f =
f =
Step 3:
N_f =
f =
s
t

FIGURE 14.9
Continued

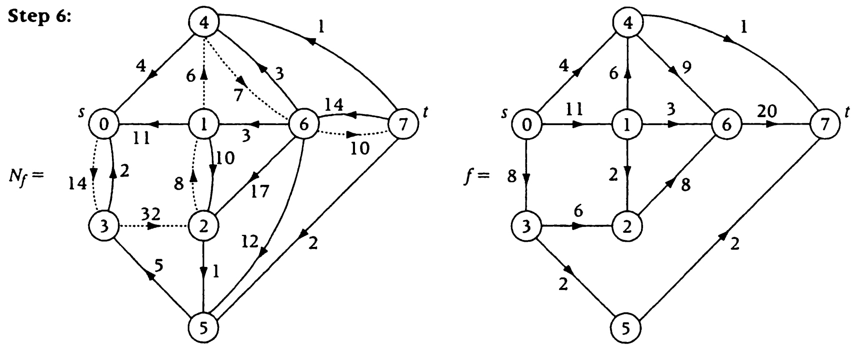
Step 4:



Step 5:

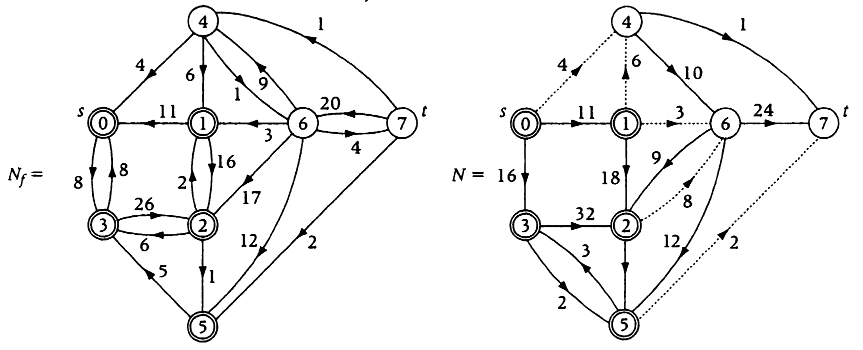


Step 6:



There are no more augmenting semipaths. The final flow f has value 23.

Step 7: Compute the f-derived network N_f and minimum cut $cut(X,Y)$.



The set $X = \{0,1,2,3,5\}$ of vertices that are accessible in N_f from the source s (marked with ◎) and the set $Y = \{4,6,7\}$ of vertices that are not accessible from s determine a cut $\Gamma = cut(X,Y)$ of capacity $c(X,Y) = 4 + 6 + 3 + 8 + 2 = 23$. Hence, we have $val(f) = 23 = cap(\Gamma)$, so that f is a maximum flow Γ and is a minimum cut.

14.2.6 Maximum Flows in 0/1 Networks: Menger's Theorem

An *integer flow* in a digraph $D = (V, E)$ is a flow f such that $f(e)$ is an integer for every edge $e \in E$.

?roposition 14.2.9 Let $D = (V, E)$ be a digraph, and let f be any integer flow from s to t, where $s, t \in V$. Then there exists a set $\mathscr{P}$ of simple paths from s to t and positive integers $\{\lambda_P \mid P \in \mathscr{P}\}$ such that the number of paths containing edge e is at most $c(e)$ and

$$val(f) = \sum_{P \in \mathscr{P}} \lambda_P. \qquad \textbf{(14.2.11)} \ \square$$

We leave the proof of Proposition 14.2.9 as an exercise. A set of paths $\mathscr{P}$ and associated positive integers $\{\lambda_P \mid P \in \mathscr{P}\}$ satisfying Formula (14.2.11) can be computed in time $O(m{*}val(f))$.

A *0/1 flow* is an integer flow such that, for each edge e, $f(e)$ is either 0 or 1. The following result is an immediate corollary of Proposition 14.2.9.

Corollary 14.2.10 Let f be any 0/1 flow from u to v. Then there exists a set $\mathscr{P}$ of pairwise edge-disjoint paths from u to v such that

$$val(f) = |\mathscr{P}| \qquad \square$$

When the capacities are all integers, $c_f(S)$ is an integer at each stage of the procedure *FordFulkerson*. Thus, for integer capacities, *FordFulkerson* generates a maximum integer flow f. In particular, if each edge has unit capacity, then a maximum 0/1 flow f is generated. By Corollary 14.2.10, if f is a 0/1 flow, then there exists a set $\mathscr{P}$ of pairwise edge-disjoint paths from u to v such that $val(f) = |\mathscr{P}|$. If $\mathscr{P}$ is a set of pairwise edge-disjoint paths from u to v, then $\Sigma_{P\in\mathscr{P}}\chi_P$ is a 0/1 flow. Thus, a set $\mathscr{P}$ of pairwise edge-disjoint paths from s to t has maximum cardinality over all such sets if and only if $\Sigma_{P\in\mathscr{P}}\chi_P$ is a maximum flow.

It follows that we can compute a maximum size set $\mathscr{P}$ of pairwise edge-disjoint paths from s to t in a given digraph D by first applying procedure *FordFulkerson* to the capacitated network $N = (D, c, s, t)$, where each edge e has capacity $c(e) = 1$ to obtain a maximum flow f, and then computing a set $\mathcal{P}$ of pairwise edge-disjoint paths from s to t such that $val(f) = |\mathcal{P}|$. Now consider the cut Γ generated by *FordFulkerson*. Because every edge has unit capacity, the capacity of Γ equals the size of (number of edges in) Γ. Since the capacity of Γ equals the value of f, it follows that the size of Γ equals the size of $\mathcal{P}$, which yields the classical theorem of Menger.

Theorem 14.2.11 **Menger's Theorem for Digraphs**
Let $D = (V, E)$ be a digraph. Then for $s, t \in V$, the maximum size of a set $\mathcal{P}$ of pairwise edge-disjoint paths from s to t equals the minimum size of a cut Γ separating s and t. □

The following corollary is the analog of Theorem 14.2.11 for graphs.

Corollary 14.2.12 **Menger's Theorem for Undirected Graphs**
Let $G = (V, E)$ be an undirected graph. Then for $s, t \in V$, the maximum size of a set $\mathcal{P}$ of pairwise edge-disjoint paths from s to t equals the minimum size of a cut Γ separating s and t. □

14.2.7 Maximum Size Matching

Procedure *FordFulkerson* can be applied to obtain a maximum size matching in a bipartite graph G. Let (X, Y) denote the associated bipartition of the vertex set V of G. Construct a digraph D as follows. The vertex set of D consists of the vertex set of G together with two new vertices s and t; that is,

$$V(D) = V(G) \cup \{s,t\}.$$

The edge set $E(D)$ of D consists of pairs xy such that $xy \in E(G)$, $x \in X$, and $y \in Y$, together with all pairs sx, $x \in X$, and all pairs yt, $y \in Y$; that is,

$$E(D) = \{xy \mid xy \in E(G), x \in X, y \in Y\} \cup \{sx \mid x \in X\} \cup \{yt \mid y \in Y\}.$$

Now consider the capacitated network $N = (D, c, s, t)$, where every edge e has capacity $c(e) = 1$. For f, a 0/1 flow from s to t, let $\mu(f)$ denote the set of all edges xy of G such that xy is an edge of D and $f(xy) = 1$. Clearly, $\mu(f)$ is a matching M in G, and $val(f)$ is equal to the size of M. Conversely, given any matching M of G there is a flow f in N such that $M = \mu(f)$. Let f^* be the flow generated by procedure *FordFulkerson*. Since f^* is a maximum-value flow in N, $\mu(f^*)$ is a maximum-size matching in G. The algorithm just described for finding a maximum-size matching is illustrated in Figure 14.10.

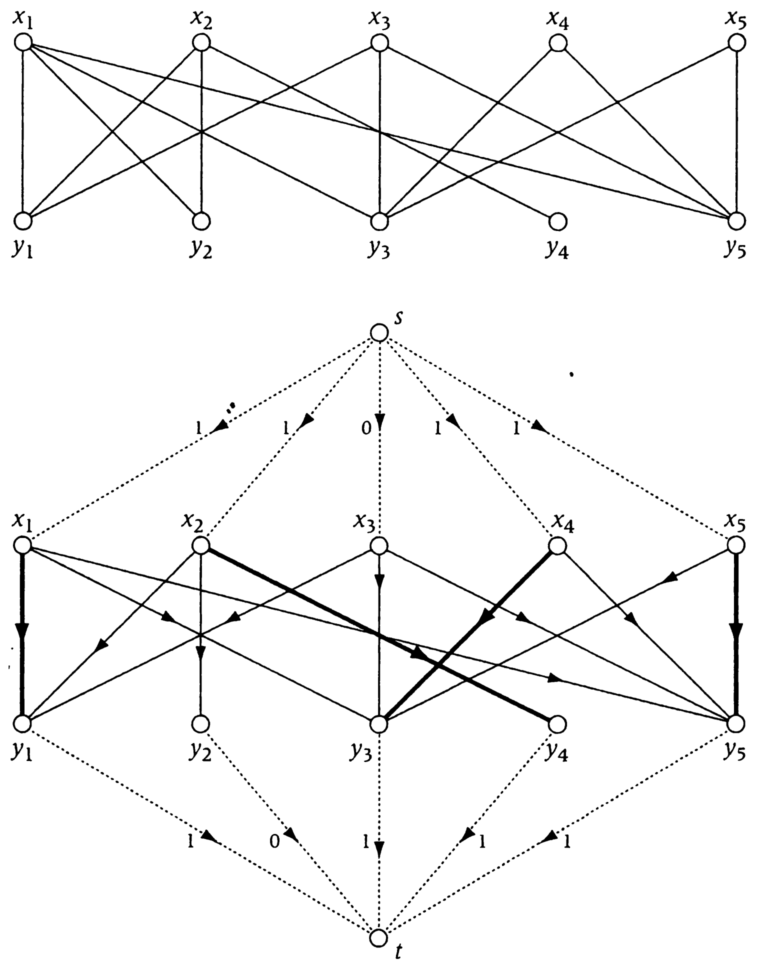


FIGURE 14.10 A maximum flow in an associated network where all edges have capacity 1 yielding a maximum matching in a bipartite graph G.

14.3 Closing Remarks

In addition to the Edmonds-Karp algorithm for finding a maximum flow, which uses the Ford-Fulkerson augmenting-path method, other efficient algorithms for finding maximum flows have been designed. These algorithms include the Dinac maximum flow algorithm and, more recently, the preflow push algorithm designed by Goldberg and Tarjan. The problem of finding a maximum flow from a single source *s* to a single sink *t* generalizes to the problem of finding a multicommodity flow from a set of sources to a set of sinks. The theory of multicommodity flows is an active research area, with applications to such areas as routing in communication networks and VLSI layout.

In this chapter, we discussed the Hungarian algorithm for finding a perfect matching in bipartite graph and the Kuhn-Munkres algorithm for finding a maximum-weight perfect matching in a weighted bipartite graph. We also showed how a maximum flow can be used to find a maximum-size matching in a bipartite graph. Algorithms for finding maximum flows and perfect matchings in general graphs can be found in the references.

In Chapter 24, we will present a parallel probabilistic algorithm for determining whether or not a bipartite graph contains a perfect matching.

References and Suggestions for Further Reading

Good references for network optimization algorithms, including flow and matching algorithms:

Ahuja, R. K. *Network Flows: Theory, Algorithms, and Applications.* Englewood Cliffs, NJ: Prentice-Hall, 1993.

Gordon, M., and M. Minoux. *Graphs and Algorithms* (trans. by S. Vajda). New York: Wiley, 1984.

Lawler, E. L. *Combinatorial Optimization: Networks and Matroids.* New York: Holt, Rinehart and Winston, 1976.

Papadimitriou, C. H., and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Englewood Cliffs, NJ: Prentice-Hall, 1982.

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

Lovász, L., and M. D. Plummer. *Matching Theory.* Amsterdam: North Holland, 1986. A nice reference to results and algorithms on matchings.

EXERCISES

Section 14.1 Perfect Matchings in Bipartite Graphs

14.1 Prove Lemma 14.1.2.

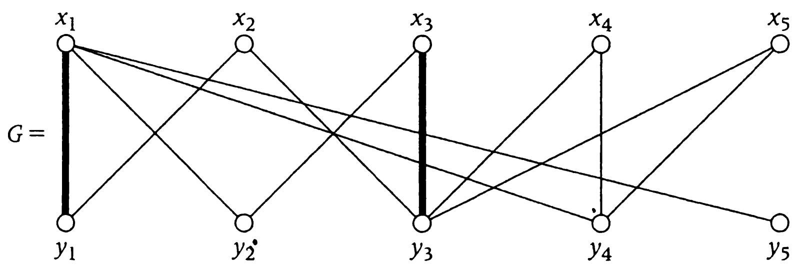
14.2 Prove Lemma 14.1.3.

14.3 Prove Lemma 14.1.4.

14.4 Show that at each stage of the procedure *Hungarian,* the alternating tree T can be grown in time $O(n^2)$, so the worst-case complexity of procedure *Hungarian* is $O(n^3)$.

14.5 Refine the pseudocode of the procedure *Hungarian* to include details for finding the augmenting paths.

14.6 Consider the following bipartite graph G:



Starting with the given matching $M = \{x_1y_1, x_3y_3\}$, determine the perfect matching output by *Hungarian.* Show each augmenting path that is generated.

14.7 Prove Proposition 14.1.5.

14.8 Verify that the vertex weighting given by Formula (14.1.8) is a feasible vertex weighting.

14.9 a. Verify that the vertex weighting ϕ' given by Formula (14.1.11) is a feasible vertex weighting whose equality subgraph $G_{\phi'}$ contains the M-alternating tree T.

b. Show that $G_{\phi'}$ contains at least one edge $\{x, y\}$, where $x \in S$ and $y \in Y - \Gamma_{\phi'}(S)$.

14.10 Show that the worst-case complexity of procedure *KuhnMunkres* is $O(n^3)$.

Section 14.2 Maximum Flows in Capacitated Networks

14.11 Let f be any flow from vertex s to vertex t in a digraph. Using Proposition 14.2.1, show that the flow out of s equals the flow into t.

14.12 Prove Proposition 14.2.2.

14.13 Verify that χ_S given by Formula (14.2.3) is a unit flow.

14.14 Verify that the final flow $f = \lambda_1\chi_{P_1} + \cdots + \lambda_k\chi_{P_k}$ generated by the naive algorithm illustrated in Figure 14.5 is maximal in the sense that if g is any other flow, then $g(e) < f(e)$ for some edge e.

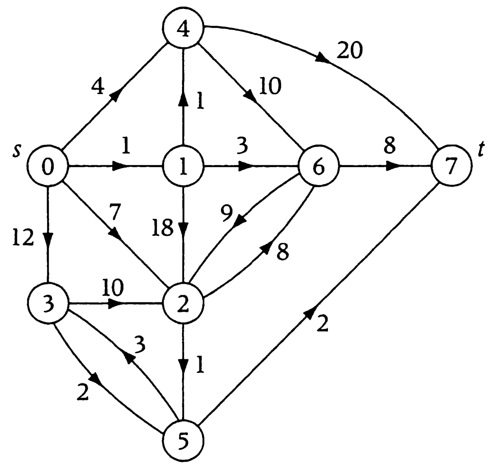
14.15 Prove Lemma 14.2.7.

14.16 Prove Lemma 14.2.8.

14.17 Show that for a poor choice of augmenting semipaths S, procedure *FordFulkerson* may never terminate.

14.18 Assuming that the capacities of the network are positive integers, show that for a poor choice of augmenting paths S, the worst-case complexity $W(n, m)$ of procedure *FordFulkerson* can also be arbitrarily large.

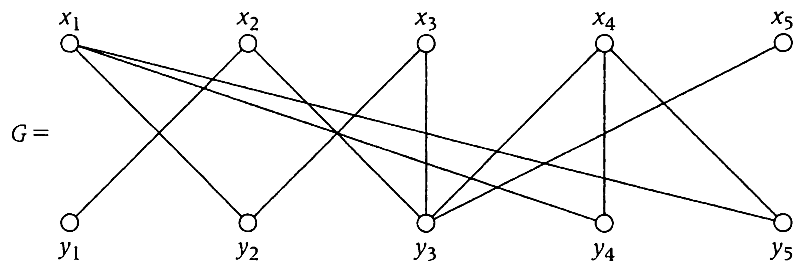
14.19 Using the Edmonds-Karp algorithm, find a maximum flow f and a minimum cut Γ in the following capacitated network N:

$N=$ 

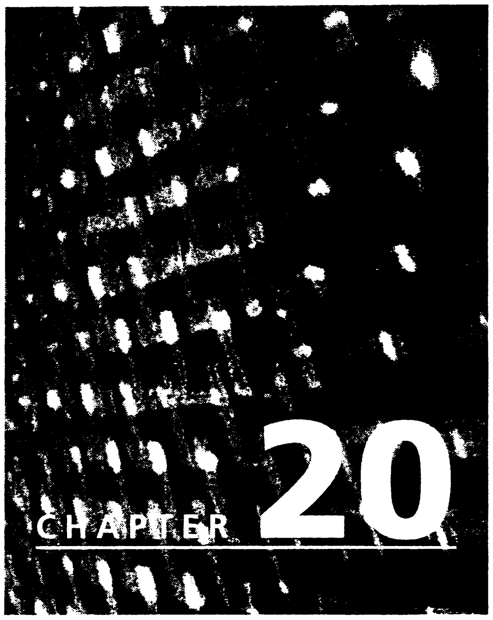
14.20 Design and analyze an algorithm for computing a set of paths P and associated positive integers $\{\lambda_P : P \in \mathrm{P}\}$ satisfying (14.2.11).

14.21 Derive Menger's theorem for undirected graphs (Corollary 14.2.12) from Menger's theorem for digraphs (Theorem 14.2.11).

14.22 Using the Edmonds-Karp algorithm, find a maximum matching in the following bipartite graph G by computing a maximum flow in the associated capacitated network.



14.23 Derive Konig's theorem, which states that the size of a maximum matching in a bipartite graph G equals the size of a minimum cover (a *cover* is a set of vertices such that every edge of G is incident with at least one vertex in the cover) from Menger's theorem for digraphs (Theorem 14.2.11).



CHAPTER 20
STRING MATCHING AND DOCUMENT PROCESSING

Finding some or all occurrences of a given pattern string in a given text is an important and commonly encountered problem. For example, most word processing software packages have built-in search-and-replace functions and spell checkers, both of which depend on finding the occurrences of words in texts. On the Internet, string matching is used for locating Web pages containing a given query string. String matching and approximate string matching is also a key technique in bioinformatics, which entails searching gene sequences for patterns of interest.

In this chapter, we present three standard string-matching algorithms, due to Knuth, Morris, and Pratt (the KMP algorithm); Boyer and Moore (the BM algorithm); and Karp and Rabin (the KR algorithm). The KMP and BM algorithms preprocess the pattern string so that information gained during a search for a match of the pattern can be used to shift the pattern more than the one position shifted by a naive algorithm when a mismatch occurs. The KR algorithm shifts the pattern by only one position at a time, but it performs an efficient (constant-time) check at each new position.

Often it is useful to find an approximate match in a text to a given pattern string. An important measure of approximation is known as the edit distance between two strings, which is, roughly speaking, the minimum number of single-character alterations that will transform one string into another. We present a dynamic programming solution to computing the edit distance between strings.

We finish the chapter with a discussion of tries and suffix trees. When the text is fixed, preprocessing the text as opposed to the pattern string leads to efficient string-matching algorithms. This preprocessing is based on constructing a trie and a related suffix tree corresponding to the text. Tries can be used to create inverted indexes to strings in a large collection of data files, such as Web pages on the Internet.

20.1 The Naive Algorithm

The string-matching problem can be formally described as follows. An *alphabet* is a set of characters or symbols $A = \{a_1, a_2, \ldots, a_k\}$. A *string* $S = S[0{:}n - 1]$ of length n on A is a sequence of n characters (repetitions allowed) from A. Such a string $S = s_0s_1, \ldots, s_{n-1}$ can be viewed as an array of characters $S[0{:}n - 1]$ from A, so that the $(i + 1)^{st}$ character s_i in the string is denoted by $S[i]$, $i = 0, \ldots, n - 1$. More generally, we denote the substring consisting of symbols in consecutive positions i through j of S by $S[i{:}j]$, $0 \leq i \leq j \leq n - 1$. The *null string*, denoted by ε, is the string that contains no symbols. We let A^* denote the set of all finite strings (including the null string ε) on A. The length of a string S, denoted $|S|$, is the number of characters in S. For $a \in A$, we let a^i denote the string of length i consisting of the single symbol a repeated i times.

Given a *pattern string* $P = P[0{:}m - 1]$ of length m and *text string* $T = T[0{:}n - 1]$ of length n, where $m \leq n$, the string-matching problem is to determine whether P occurs in T. In our string-matching algorithms, we assume that we are looking for the first occurrence (if any) of the pattern string in the text string. The algorithms can be readily modified to return all occurrences of the pattern string.

A naive algorithm for finding the first occurrence of P in T is to position P at the start of T and simply shift the pattern P along T, one position at a time, until either a match is found or the string T is exhausted (that is, position $n - m + 1$ in T is reached without finding a match).

```
function NaiveStringMatcher(P[0:m - 1], T[0:n - 1])
Input:  P[0:m - 1] (a pattern string of length m)
        T[0:n - 1] (a text string of length n)
Output: returns the position in T of the first occurrence of P, or -1 if P does not occur
        in T
        for s ← 0 to n − m do
            if T[s : s + m − 1] = P then
                return(s)
            endif
        endfor
        return(−1)
    end NaiveStringMatcher
```

When measuring the complexity of procedure *NaiveStringMatcher*, it is natural to choose comparison of symbols as our basic operation. We can test whether $T[s: s + m - 1] = P$ by using a simple linear scan. Clearly, this scan requires a single comparison in the best case ($P[0] \neq T[s]$) and m comparisons in the worst case ($P[i] = T[s + i], i = 0, \dots, m - 2$). Because the **for** loop of procedure *NaiveStringMatcher* is iterated $n - m + 1$ times, *NaiveStringMatcher* never performs more than $m(n - m + 1)$ comparisons. Moreover, $m(n - m + 1)$ comparisons are performed, for example, when $P[0:m - 1]$ and $T[0:n - 1]$ are the strings $0^{m-1}1$ and 0^n, respectively, over the alphabet $A = \{0, 1\}$. Thus, *NaiveStringMatcher* has worst-case complexity

$$W(m, n) = m(n - m + 1) \in O(nm).$$

The action of *NaiveStringMatcher* is illustrated for a sample pattern and text string in Figure 20.1.

P	0	0	1	0	0	2	0	1														
T	0	0	1	0	0	1	0	0	2	0	0	0	1	0	0	2	0	1	2	2	0	0
$s = 0$	0	0	1	0	0	2																
$s = 1$		0	0																			
$s = 2$			0																			
$s = 3$				0	0	1	0	0	2	0	1											
$s = 4$					0	0																
$s = 5$						0																
$s = 6$							0	0	1													
$s = 7$								0	1													
$s = 8$									1													
$s = 9$										0	0	1										
$s = 10$											0	0	1	0	0	2	0	1				

FIGURE 20.1 Action of procedure *NaiveStringMatcher* for a sample pattern string $P[0:7]$ and a text string $T[0:22]$ on alphabet $A = \{0, 1, 2\}$. For each shift s, we show that portion of P that matches T, with the first mismatch shaded. A match is found beginning at the 11th character of T.

20.2 The Knuth-Morris-Pratt Algorithm

There is an obvious inefficiency in *NaiveStringMatcher*: At a given point we might have matched a good part of the pattern P with the text T until we found a mismatch, but we don't exploit this in any way. The KMP string-matching algorithm is based on a strategy of using information from partial matchings of P to not only skip over portions of the text that cannot contain a match but also to avoid checking characters in T that we already know match a prefix of P. The KMP algorithm achieves $O(n)$ worst-case complexity by preprocessing the string P to obtain information that can exploit partial matchings.

To illustrate, consider the pattern string P = "00100201" and text string T = "0010010020001002012200". Placing P at the beginning of T, note that a mismatch occurs at index position 5. The naive algorithm would then shift by one to $T[1]$ and simply start all over again checking at the beginning of P. This completely ignores that we have already determined that $P[0{:}4] = 00100 = T[0{:}4]$. Indeed, by simply looking at $P[0{:}4]$, we see that the first position in T where a match could possibly occur is at $i = 3$, because any shift of P by less than three will cause mismatches between the relevant prefix of $P[0{:}3] = T[0{:}3]$ and suffix of $P[1{:}4] = T[1{:}4]$ determined by the shift. Indeed, if we shift by one, we would be comparing the prefix 0010 of $P[0{:}3]$ with the suffix 0100 of $P[1{:}4]$. Similarly, if we shift by two, we would be comparing the prefix 001 of $P[0{:}3]$ with the suffix 100 of $P[1{:}4]$. Thus, we need to shift by three before we match the prefix 00 of $P[0{:}3]$ with the suffix 00 of $P[1{:}4]$. Hence, from the mismatch that occurs at position $i = 5$, the next starting position where a match can occur is at position $5 - 2 = 3$. Moreover, we do not need to check the first two characters in P because they already match the first two characters of T at this new position for P. Note that next position where a match can occur was obtained by subtracting the length of the largest prefix of $P[0{:}3]$ that was also a suffix of $P[1{:}4]$ from the position where the mismatch occurred.

More generally, suppose we have detected a mismatch at position i in T, where $T[i] \neq P[j]$, but we know that the previous j characters of T match with $P[0{:}j - 1]$. Also, suppose d_j is the length of the longest prefix of $P[0{:}j - 2]$ that also occurs as a suffix of $P[1{:}j - 1]$. Then the next position where a match can occur is at position $i - d_j$. Moreover, to see whether we actually have a match starting at position $i - d_j$, we can avoid checking the characters that we already know agree with those in T—that is, the characters in the substring $T[s - d_j{:}i - 1]$. Hence, we need only check the characters in the substring $T[i{:}i + m - d_j - 1]$ with those in the substring $P[d_j{:}\, m - 1]$ to see if a match occurs (see Figures 20.2 and 20.3).

$$\begin{array}{llllll} \text{position} & i-j & i-d_j & & i-1 & i \\ T & \ldots P[0]\;P[1]\ldots & P[j-d_j] & P[j-d_j+1]\ldots & P[j-1] & T[i]\neq P[j]\ldots \\ & & \| & \| & \| & \\ & & P[0] & P[1]\ldots & P[d_j-1] & P[d_j]\ldots \end{array}$$

FIGURE 20.2 The state of T when $T[i] \neq P[j]$, but $T[i-j:i-1] = P[0:j-1]$ and d_j is the length of the longest prefix of $P[0:j-2]$ that agrees with a suffix of $P[1:j-1]$.

T	0010010020001002012200	0010010020001002012200
P	00100201	00100201
	mismatch at $i = j = 5$	$5 - 2$ is next index where a match could occur (it doesn't occur there)

FIGURE 20.3 Interpretation of Figure 20.2 for specfic P and T.

By preprocessing the string P, we can compute the array $Next[0:m-1]$, where $Next[j]$ is the length of longest prefix of $P[0:j-2]$ that agrees with a suffix of $P[1:j-1]$, $j = 2, \ldots, m$. We set $Next[0] = Next[1] = 0$. For example, for the string P = "00100201", we have the corresponding array $Next[0:7] = [0, 0, 1, 0, 1, 2, 0, 1]$. Then for this P and the T discussed earlier, Figure 20.3 shows the situation described in Figure 20.2 for $i = 5, j = 5$, and $d_j = 2$.

The following key fact summarizes our discussion and is the key to the efficiency of the KMP string matching algorithm.

Key Fact

Suppose in our scan of T looking for a match with P that we have a mismatch at position s in T, where $T[i] \neq P[j]$, but $P[0:j-1] = T[i-j:i-1]$. Setting $d_j = Next[j]$, the next position where a match of P can occur is at position $i - d_j$. Moreover, we need only check the substring $T[i:i+m-d_j-1]$ with the substring $P[d_j:m-1]$ to see if a match occurs there.

In the pseudocode *KMPStringMatcher*, we look for matches using a variable i that scans the text T from left to right one position at a time, and a second variable j that scans the pattern P in a slightly oscillatory manner, as dictated by the key fact (see Figure 20.4). The variable i never backs up, so that when a match occurs, it is actually at position $i - m + 1$ (in other words, s is the position of the last character in P corresponding to the matching). The pseudocode is elegant but somewhat subtle, because when a mismatch $P[j] \neq T[i]$ occurs, there is no need to explicitly place the pattern P at the next position $i - Next[j]$; we merely need to check $T[i : i + m - Next[j] - 1]$ against $P[Next[j] : m - 1]$ to see if a match of P occurs at $i - Next[j]$. We perform this check by continuing the scan of T by i and replacing j by $Next[j]$ before continuing the scan of P by j.

```
function KMPStringMatcher(P[0:m - 1], T[0:n - 1])
Input:    P[0:m - 1] (a pattern string of length m)
          T[0:n - 1] (a text string of length n)
Output:   returns the position in T of the first occurrence of P, or -1
        if P does not occur in T
  i ← 0                           //i runs through text string T
  j ← 0                           //j runs through pattern string P in manner
                                    dictated by key fact
  CreateNext(P[0:m - 1], Next[0:m - 1])
  while i < n do
      if P[j] = T[i] then
          if j = m - 1 then       //match found at position i - m + 1
              return(i - m + 1)
          endif
          i ← i + 1               //continue scan of T
          j ← j + 1               //continue scan of P
      else                        //P[j] ≠ T[i]
          j ← Next[j]             //continue looking for a match of P which
                                    now could begin at position i - Next[j]
                                    in T
          if j = 0 then
              if T[i] ≠ P[0] then //no match at position i
                  i ← i + 1
              endif
          endif
      endif
  endwhile
  return(-1)
end KMPStringMatcher
```

In Figure 20.4, we illustrate the action of *KMPStringMatcher* for the pattern string P and text string T discussed earlier, by tracing the values of s and j for each iteration of the **while** loop. While *NaiveStringMatcher* used 37 comparisons to find a match, *KMPStringMatcher* only used 21 (not counting the comparisons made by *CreateNext* in preprocessing P).

iteration						!						!		!							
i	0	1	2	3	4	5	5	6	7	8	9	10	10	11	11	12	13	14	15	16	17
j	0	1	2	3	4	5	2	3	4	5	6	7	1	2	1	2	3	4	5	6	7

(a)

```
T    0 0 1 0 0 1 0 0 2 0 0 0 1 0 0 2 0 1 2 2 0 0
P    0 0 1 0 0 2 0 1
           0 0 1 0 0 2 0 1
                       0 0 1 0 0 2 0 1
                         0 0 1 0 0 2 0 1  match
```

(b)

FIGURE 20.4 (a) A trace of the values of *i* and *j* for each iteration of the **while** loop in *KMPStringMatcher* for *P* = "00100201" and *T* = "0010010020001002012200". Positions marked with ! are where $T[i] \neq P[j]$ and *j* is reassigned with the value *Next*[*j*]. (b) The implicit shifting of *P* until a match is found.

KMPStringMatcher has linear complexity because of the following key fact.

Key Fact

The while loop in *KMPStringMatcher* is executed at most 2*n* times.

To verify the key fact, note that the loop executes n times when i is incremented within the loop. If i is not incremented in the loop, then the pattern P is implicitly shifted to position $i - Next[j]$, which is at least one more than the last implicit placement of P. Therefore, this implicit shift can happen at most n times, which verifies the key fact.

It remains to design the algorithm *CreateNext*. Again, there is a naive algorithm *NaiveCreateNext* that computes each value of *Next*[i] from scratch, without using any of the information gleaned from computing *Next*[k], $k < i$. It is easy to see that the worst case of *NaiveCreateNext* is in $\Omega(m^2)$. However, similar to the design of *KMPStringMatcher*, using information gleaned about *Next*[$i - 1$] leads to a more efficient way to compute *Next*[i] and yields an $O(m)$ algorithm. We leave the complexity analysis and correctness of *CreateNext* to the exercises.

```
function CreateNext(P[0:m − 1], Next[0:m − 1])
Input:   P[0:m − 1] (a pattern string of length m)
Output:  Next[0:m − 1] (Next[i] is length of longest prefix of P [0:i − 2] that is a suffix
         of P[1:i − 1], i = 0, ... , m − 1)
```

```
        $Next[0] \leftarrow Next[1] \leftarrow 0$
        $i \leftarrow 2$
        $j \leftarrow 0$
        while $i < m$ do
            if $P[j] = P[i - 1]$ then
                $Next[i] \leftarrow j + 1$
                $i \leftarrow i + 1$
                $j \leftarrow j + 1$
            else
                if $j > 0$ then
                    $j \leftarrow Next[j - 1]$
                else
                    $Next[i] \leftarrow 0$
                    $i \leftarrow i + 1$
                endif
            endif
        endwhile
    end CreateNext
```

20.3 The Boyer-Moore String-Matching Algorithm

Similar to the KMP algorithm, the BM algorithm uses preprocessing of the pattern string to facilitate shifting the pattern string, but it is based on a right-to-left scan of the pattern string instead of the left-to-right scan made by the KMP algorithm. We present a simplified version of the BM algorithm that compares the rightmost character of the pattern with the character in the text corresponding to the current shift of the pattern and uses this comparison to determine the next pattern shift (if any). The full version of the BM algorithm is developed in the exercises.

In the BM algorithm, the pattern $P[0:m - 1]$ is first placed at the beginning of the text, and we check for a match by scanning the pattern from right-to-left. If we find a mismatch, then we have two cases to consider, depending on the character x of the text in index position $m - 1$ that is compared against the last character of P. If x does not occur in the first $m - 1$ positions of P, then clearly we can shift P by its entire length m to continue our search for a match. If x does occur in the first $m - 1$ positions of P, then we shift P so that the rightmost occurrence of x in $P[0:m - 1]$ is now at index position $m - 1$ in the text, and we repeat the process of scanning P from right-to-left at this new position. Again, if we find a mismatch, we shift the pattern again based on the text character that was aligned at the rightmost character of P. Also, in this simplified version, we ignore any information gleaned about partial matchings in our previous placement of P (this information *is* used in the full version of the BM algorithm).

The shifts associated with the two cases can be computed easily by preprocessing the pattern string P. In the following pseudocode for *CreateShift*, for convenience, we assume that the array *Shift* is indexed by the alphabet A from which the characters for the pattern P and text T are drawn.

```
procedure CreateShift(P[0:m − 1], Shift[0: |A| − 1])
Input:    P[0:m − 1] (a pattern string)
Output:  Shift[0: |A| − 1] (the array of character-based shifts)
   for i ← 0 to |A| − 1 do                //initialize Shift to all m's.
      Shift[i] = m
   endfor
   for i ← 0 to m − 2 do                  //compute shifts based on rightmost
                                            occurrence of P[i] in P[0:m − 1]
      Shift[P[i]] = m − i − 1
   endfor
end CreateShift
```

For example, suppose $P[0:8]$ is the string "character". Then the values for the shifts of the characters e, t, c, a, r, h are 1, 2, 3, 4, 5, 7, respectively. The shift value for all other characters is 9. In Figure 20.5, we illustrate how the simplified BM algorithm uses these shifts to find a match of the pattern "character" in the text "BMmatcher_shift_character_example".

The worst-case performance of the simplfied BM algorithm is the same as that of the naive algorithm, $\Theta(nm)$ (see Exercise 20.8). However, it can be shown that its average behavior is linear in n and often works as well as the full version of the BM algorithm. The full version of the BM algorithm works identically to the simplified version when there is a mismatch between the rightmost character of the pattern and the text character corresponding to this rightmost character in the current shift of the pattern. If these two characters agree, however, the full version acts differently by exploiting the information gained by the matching of a suffix of P with the corresponding characters in the text for the given placement of P (see the discussion preceding Exercise 20.9).

FIGURE 20.5

Action of simplified BM algorithm, with positions where mismatches first occur in the right-to-left scan of the pattern indicated by !.

```
            !             !                 !
B M m a t c h e r _ s h i f t _ c h a r a c t e r _ e x a m p l e
c h a r a c t e r                                 S h i f t ( r ) = 5
          c h a r a c t e r                       S h i f t ( r ) = 9
                            c h a r a c t e r     S h i f t ( r ) = 2
                                c h a r a c t e r   m a t c h
```

20.4 The Karp-Rabin String-Matching Algorithm

In this section, we assume without loss of generality that our strings are chosen from the k-ary alphabet $A = \{0,1, \dots , k - 1\}$. Each character of A can be thought of as a digit in radix-k notation, and each string $S \in A^*$ can be identified with the base k representation of an integer $\overline{S}$. For example, when $k = 10$, the string of numeric characters "6832355" can be identified with the integer 6832355. Given a pattern string $P[0{:}m - 1]$, we can compute the corresponding integer using m multiplications and m additions by employing Horner's rule.

$$\begin{aligned}\overline{P} = P[m - 1] + k(P[m - 2] + \\ k(P[m - 2] + k(P[m - 3] + \cdots + k(P[1] + kP[0])\dots))\end{aligned} \tag{20.4.1}$$

Given a text string $T[0{:}n - 1]$ and an integer s, we find it convenient to denote the substring $T[s, s + m - 1]$ by T_s. A string-matching algorithm is obtained by using Horner's rule to successively compute $\overline{T_0}$, $\overline{T_1}$, $\overline{T_2}$, … , where the computation continues until $\overline{P} = \overline{T_s}$ for some s (a match) or until we reach the end of the text T. Of course, this is no better than the naive string-matching algorithm. However, the following key fact is the basis of a linear algorithm.

Given the integers $\overline{T_{s-1}}$ and k^{m-1}, we can compute the integer $\overline{T_s}$ in constant time.

The key fact follows from the following recurrence relation:

$$\overline{T_s} = k(\overline{T_{s-1}} - k^{m-1}T[s - 1]) + T[s + m - 1] \quad s = 1,\dots, n - m. \tag{20.4.2}$$

For example, if $k = 10$, $m = 7$, $\overline{T_{s-1}} = 7937245$, and $\overline{T_s} = 9372458$, then recurrence relation (20.4.2) becomes

$$\overline{T_s} = 10[7937245 - (1000000 \times 7)] + 8 = 9372458.$$

The constant $c = k^{m-1}$ in (20.4.2) can be computed in time $O(\log m)$ using the binary method for computing powers. Once c is computed, it does not need to be recomputed when Formula (20.4.2) is applied again. Thus, assuming that the arithmetic operations in (20.4.2) take constant time, each application of (20.4.2) takes constant time. Hence, the $n - m + 2$ integers $\overline{P}$ and $\overline{T_s}$, $s = 0, 1, \dots , n - m$, can be computed in total time $O(n)$.

The problem with the preceding approach is that the integers $\overline{P}$ and $\overline{T_s}$, $s = 0, 1, \ldots, n - m$, may be too large to work with efficiently, and the assumption that Formula (20.4.2) can be performed in constant time becomes unreasonable. To get around this difficulty, we reduce these integers modulo q for some randomly chosen integer q. To avoid multiple-precision arithmetic, q is often chosen to be a random prime number such that kq fits within one computer word.

We now let

$$\begin{aligned} \overline{P}^{(q)} &= \overline{P} \bmod q, \\ \overline{T_s}^{(q)} &= \overline{T_s} \bmod q. \end{aligned} \qquad \textbf{(20.4.3)}$$

The values $\overline{T_s}^{(q)}$ and $\overline{P}^{(q)}$ can be computed in time $O(n)$ using exactly the same algorithm described earlier for computing $\overline{T_s}$ and $\overline{P}$, except that all arithmetic operations are performed modulo q. Clearly, if $\overline{T_s}^{(q)} \neq \overline{P}^{(q)}$ then $T_s \neq P$. However, if $\overline{T_s}^{(q)} = \overline{P}^{(q)}$, we are not guaranteed that $P = T_s$. When a shift s has the property that $\overline{T_s}^{(q)} = \overline{P}^{(q)}$, but $T_s \neq P$ we have a *spurious match*. However, for sufficiently large q, the probability of a spurious match can be expected to be small. We check whether a match is spurious by explicitly checking whether $T_s = P$, and continuing our search for a match if $T_s \neq P$.

```
function KarpRabinStringMatcher(P[0:m - 1], T[0:n - 1], k, q)
Input:   P[0:m - 1] (a pattern string of length m)
         T[0:n - 1] (a text string of length n)
         k (A is the k-ary alphabet {0, 1, . . . , k - 1})
         q (a random prime number q such that kq fits in one computer word)
Output:  returns the position in T of the first occurrence of P, or -1 if P does not occur
         in T
   c ← $k^{m-1}$ mod q
   $\overline{P}^{(q)}$ ← 0
   $\overline{T_0}^{(q)}$ ← 0
   for i ← 0 to m - 1 do                         //apply Horner's rule to compute
                                                    $\overline{P}^{(q)}$ and $\overline{T_0}^{(q)}$
      $\overline{P}^{(q)}$ ← (k*$\overline{P}^{(q)}$ + P[i]) mod q
      $\overline{T_0}^{(q)}$ ← (k*$\overline{T_0}^{(q)}$ + T[i]) mod q
   endfor
   for s ← 0 to n - m do
      if s > 0 then
            $\overline{T_s}^{(q)}$ ← (k*($\overline{T_{s-1}}^{(q)}$ - T[s - 1]*c) + T[s + m - 1]) mod q
      endif
```

```
            if $\overline{T}_s^{(q)} = \overline{P}^{(q)}$ then
                if $T_s = P$ then                              //match is not spurious
                    return(s)
                endif
            endif
        endfor
        return(0)
    end KarpRabinStringMatcher
```

Figure 20.6 illustrates the action of function *KarpRabinStringMatcher* for a sample pattern string $P[0:6]$ = "6832355", text string $T[0:20]$ = "895732102583 235544031", and prime modulus $q = 11$. To illustrate the calculations of $\overline{T}_s^{(11)}$ in Figure 20.6, we show how *KarpRabinStringMatcher* computes $\overline{T}_3^{(11)}$from $\overline{T}_2^{(11)}$. Using Formula (20.4.2) and doing all arithmetic modulo $q = 11$, we have

$$\begin{aligned} 7321026 &\equiv 10(5732102 - 1000000 \times 5) + 6 \pmod{11} \\ &\equiv 10(2 - 1 \times 5) + 6 \pmod{11} \\ &\equiv 9 \pmod{11}. \end{aligned}$$

Because *KarpRabinStringMatcher* terminates after finding the first nonspurious match, the worst-case performance occurs for an input pair (P, T), where the pattern string P occurs precisely at the end ($s = n - m$) of the text string T. With q chosen at random, we can expect different behavior for different choices of q, so that we now consider the expected number $\tau_{\exp}(P, T)$of string comparisons made by the algorithm. For $s \neq n - m$, we make the assumption that $\overline{T}_s^{(q)}$ takes on a particular value $i \in \{0, 1, \dots, q - 1\}$ with equal probability $1/q$. Because a spurious match occurs only when $\overline{T}_s^{(q)} = \overline{P}^{(q)}$, $s = 0, \dots, n - m - 1$, it follows that

FIGURE 20.6 Action of *KarpRabinStringMatcher* for a sample pattern string $P[0:6]$, text string $T[0:20]$, and prime modulus $q = 11$.

$\overline{T}[0:20]$ 895732102683235544031
$\overline{P}[0:6]$ 6832355

s	$\overline{T}_s$	$\overline{T}_s^{(11)}$
0	8 9 5 7 3 2 1	10
1	9 5 7 3 2 1 0	9
2	5 7 3 2 1 0 2	2 Spurious match
3	7 3 2 1 0 2 6	9
4	3 2 1 0 2 6 8	6
5	2 1 0 2 6 8 3	0
6	1 0 2 6 8 3 2	4
7	0 2 6 8 3 2 3	0
8	2 6 8 3 2 3 5	5
9	6 8 3 2 3 5 5	2 Match: return (9)

a spurious match occurs at shift s with probability $1/q$. Let r denote the expected number of spurious matches. A test for a spurious match involves m comparisons in the worst case, and $r + 1$ such tests are performed (including the test at shift $s = n - m$); thus, the expected performance $\tau_{\text{exp}}(P, T)$ of *KarpRabinStringMatcher* for the input (P, T) is

$$\tau_{\text{exp}}(P, T) = (r + 1)m + (n - m + 1). \qquad \textbf{(20.4.4)}$$

The value r is the expectation of the binomial distribution with $n - m$ trials (shifts), where success (a spurious match) occurs with probability $1/q$. Thus, from Formula (E.3.9) of Appendix E, we have

$$r = \frac{n - m}{q}. \qquad \textbf{(20.4.5)}$$

Substituting Formula (20.4.5) into Formula (20.4.4), we obtain

$$\tau_{\text{exp}}(P, T) = \left(\frac{n - m}{q} + 1\right)m + (n - m + 1). \qquad \textbf{(20.4.6)}$$

If we assume that q is bounded above by a fixed constant, then *KarpRabinStringMatcher* achieves a worst-case complexity in $\Theta(nm)$, which is no better than the naive algorithm. However, in practice, it is reasonable to assume that q is much larger than m, in which case *KarpRabinStringMatcher* has complexity in $O(n)$. The Karp-Rabin string-matching algorithm has the additional feature that it is readily adapted to the problem of finding $m \times m$ patterns in $n \times n$ texts (see Exercise 20.16).

20.5 Approximate String Matching

In practice, there are often misspellings when creating a text, and it is useful when searching for a pattern string P in a text to find words that are approximately the same as P. In this section, we formulate a solution to this problem using dynamic programming. We have already discussed a solution to a similar problem in Chapter 9—namely, the problem of finding the longest common subsequence of two strings.

We first consider the problem of determining whether a pattern string $P[0{:}m - 1]$ is a k-approximation of a text string $T[0{:}n - 1]$. Later, we look at the problem of finding occurrences of substrings of T for which P is a k-approximation. The pattern string P is a *k-approximate matching* of the text string T if T can be converted to P using at most k operations involving one of the following

1. Changing a character of T (substitution)
2. Adding a character to T (insertion)
3. Removing a character of T (deletion)

For example, when P is the string "algorithm", one of the following might occur:

1. elgorithm $\rightarrow$ algorithm (*substitution* of e with a)
2. algorthm $\rightarrow$ algorithm (*insertion* of letter i)
3. lalgorithm $\rightarrow$ algorithm (*deletion* of letter l)

In this example, each string T differs from P by at most one character. Unfortunately, in practice, more serious mistakes are made, and the difference involves multiple characters. We define the *edit distance D(P, T)* between P and T to be the minimum number of operations of substitution, deletion, and insertion needed to convert T to P. For example, the strings "algorithm" and "logarithm" have edit distance 3.

logarithm $\rightarrow$ alogarithm $\rightarrow$ algarithm $\rightarrow$ algorithm

Let $D[i,j]$ denote the edit distance between the substring $P[0{:}i-1]$ consisting of the first i characters of the pattern string P and $T[0{:}j-1]$ consisting of the first j characters of the text string T. If $P[i] = T[j]$, then $D[i,j] = D[i-1,j-1]$. Otherwise, consider an optimal intermixed sequence involving the three operations substitution, insertion, and deletion that converts $T[0{:}j-1]$ into $P[0{:}i-1]$. The number of such operations is the edit distance between these two substrings. Note that in transforming T to P, inserting a character into T is equivalent to deleting a character from P. For convenience, we will perform the equivalent operation of deleting characters from P rather than adding characters to T. We can assume without loss of generality that the sequence of operations involving the first $i-1$ characters of P and the first $j-1$ characters of T are operated on first. To obtain a recurrence relation for $D[i,j]$, we examine the last operation. If the last operation is substitution of $T[j]$ with $P[i]$ in T, then $D[i,j] = D[i-1,j-1] + 1$. If the last operation is the deletion of $P[i]$ from P, then $D[i,j] = D[i-1,j] + 1$. Finally, if the last operation is deletion of $T[j]$ from T, then $D[i,j] = D[i,j-1] + 1$ The edit distance is realized by computing the minimum of these three possibilities. Observing that the edit distance between a string of size i and the null string is i, we obtain the following recurrence relation for the edit distance:

$$D[i,j] = \begin{cases} D[i-1,j-1], & \text{if } P[i] = T[j], \\ \min\{D[i-1,j-1]+1,\ D[i-1,j]+1,\ D[i,j-1]+1\}, & \text{otherwise.} \end{cases}$$

init. cond. $D[0,i] = D[i,0] = i$.

(20.5.1)

The design of a dynamic programming algorithm based on this recurrence and its analysis is similar to that given for the longest common subsequence problem discussed in Chapter 9, and we leave it to the exercises. We also leave as an exercise designing an algorithm for finding the first occurrence or all occurrences of a substring of the text string T that is a k-approximation of the pattern string P.

20.6 Tries and Suffix Trees

By preprocessing the pattern string, the KMP and BM algorithms achieved improvement over the naive algorithm. Another approach that can be applied when the text is fixed is to preprocess the strings in the text using a data structure such as a tree. In this section, we discuss two important tree-based data structures, tries and suffix trees, for preprocessing the text to allow for very efficient pattern matching and information retrieval.

20.6.1 Standard Tries

Consider a collection C of strings from an alphabet A of size k, where no string in S is a prefix of any other string. We can then construct a tree T whose nodes are labeled with symbols from A, such that the strings in C correspond precisely to the paths in T from the root R to a leaf node as follows. We construct T such that the labels of the children of each node are unique and occur in increasing order as the children are scanned from left to right. Starting with the tree T consisting of a single root node R, we inductively incorporate a new string $S[0{:}p - 1]$ from C into T as follows. If no child of the root R is labeled $S[0]$, then we simply add a new branch at the root consisting of a path of length p whose node at level $i + 1$ is labeled $S[i]$, $i = 0, \ldots,$ $p - 1$. Otherwise, we follow a path from the root by first following the edge from the root to the unique child of the root labeled $S[0]$, then following the edge from that node to its child labeled $S[1]$, and so forth until we reach a node v at level i labeled $S[i - 1]$ having no child (at level $i + 1$) labeled $S[i]$. We then add a new branch at v consisting of a path of length $p - i - 1$, such that node in the path at level j (in the tree) is labeled $S[j - 1]$, $j = i + 1, \ldots, p - 1$. A tree T constructed in this way is called a *standard trie* for the string collection C. Figure 20.7 shows a standard trie for the sample string collection $C = \{$"internet", "interview", "internally", "algorithm", "all", "web", "world"$\}$.

The leaf nodes of the trie T can be used to store information about the string S corresponding to the leaf, such as the location in the text of P, the number of occurrences of P in the text, and so forth. The term *trie* comes from the word re*trie*val, because a trie can be used to retrieve information about P. In addition to pattern matching, tries can be used for word matching, where the pattern is matched only to substrings of the text corresponding to words. This is useful for creating a forward index of words in a web document.

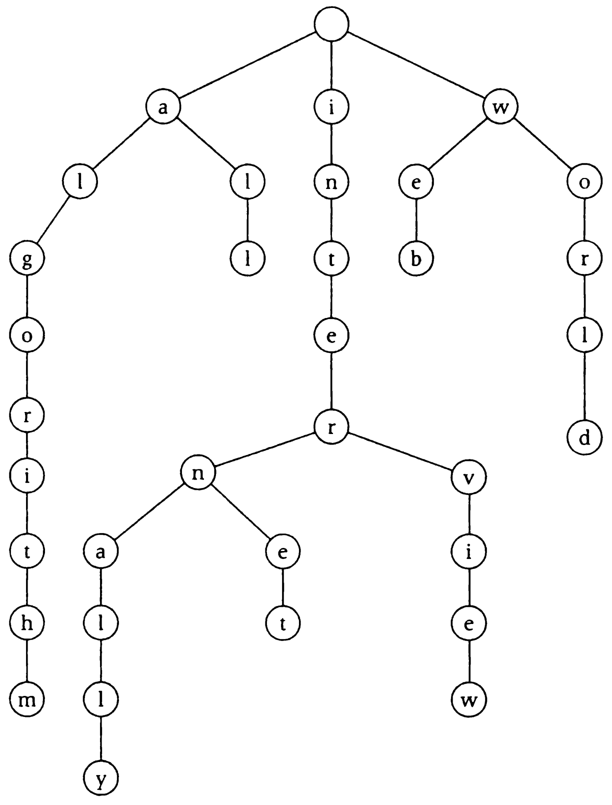


FIGURE 20.7

Standard trie for the collection of strings C = {"internet", "interview", "internally", "algorithm", "all", "web", "world"}.

It is immediate that a standard trie T has the following three properties: (1) Each nonleaf node has at most k children, where k is the size of the alphabet A; (2) the number of leaf nodes equals the number of strings in S; and (3) the depth of T equals the length of the longest string in S. The following proposition about the space requirements for storing T is easily verified (see Exercise 20.22).

Proposition 20.6.1 Let T be a standard trie for a collection C of strings, and let s denote the total length over all the strings in C. Then the number of nodes $N(T)$ of T satisfies

$$N(T) \in O(s).$$

We can efficiently test whether a given pattern string $P[0{:}m-1]$ belongs to C by scanning the string P and successively following the child in the trie labeled with the current symbol that has been scanned until either no child of the current node has a label equal to the symbol or a leaf node labeled with the last symbol of P has been reached. Because we have made the assumption that no string in C is a prefix of any other string, it follows that the pattern string

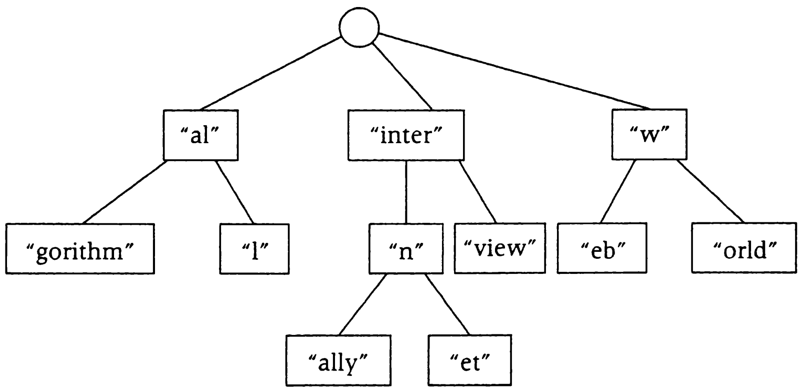
$P[0{:}m-1]$ belongs to C if and only if a leaf node labeled with the last symbol in P is reached. Because each node has at most k children, this procedure has complexity $O(km)$. Thus, if the alphabet has constant size, the complexity of searching for P is linear in its length.

20.6.2 Compressed Tries

The $O(s)$ space requirement of a standard trie T can be reduced if there are nodes in T that have only one child. Consider any such node v, and let c denote its only child. Let x and y denote the labels of v and c, respectively, and let u denote the parent of v. Then the path generated by a string S from C that contains v must also contain c. Thus, without affecting our ability to match S, we can compress the trie by removing v, making c a child of u, and replacing the label y of c with the string xy. This operation can be repeated for other nodes having only one child, except that x and y may themselves be strings instead of just single symbols. After repeatly performing this compression operation until all internal nodes have at least two children, we obtain a tree labeled with strings, which we call the *compressed trie* for S. A compressed trie is also called a PATRICIA (practical algorithm to retrieve information coded in alphanumeric) tree . Note that the compressed trie for S can also be obtained by replacing every path $uu_1 \dots u_k v$ from a node u to a node v in T whose internal nodes $u_1, u_2, \dots, u_k$ are bivalent (have exactly one child), but whose end nodes u and v are not, with the edge uv and replacing the label y of v with the string $x_1x_2 \dots x_k y$, where x_i denotes the label of u_i, $i = 1, \dots, k$. The compressed trie for the trie given in Figure 20.7 is shown in Figure 20.8.

FIGURE 20.8

Compressed trie for the same string set C = {"internet", "interview", "internally", "algorithm", "all", "web", "world"} of Figure 20.7



The following proposition about the number of nodes of a compressed trie T is easily verified.

oposition 20.6.2 Let T be a compressed trie for a collection C of c strings. Then the number of nodes $N(T)$ of T satisfies

$$N(T) \in O(c).$$

Comparing this result with Proposition 20.6.1, we see that compressing the standard trie has reduced the space requirements from $O(s)$ to $O(c)$. This becomes significant when the strings in C are long. A compressed trie can be created directly from the set of strings C without first constructing a standard trie for C and then compressing it. We leave as an exercise designing an algorithm for constructing a compressed trie directly from C. Using a slight modification of the technique used for a standard trie, we can search a compressed trie to efficiently test whether a given pattern string belongs to C (see Exercise 20.24).

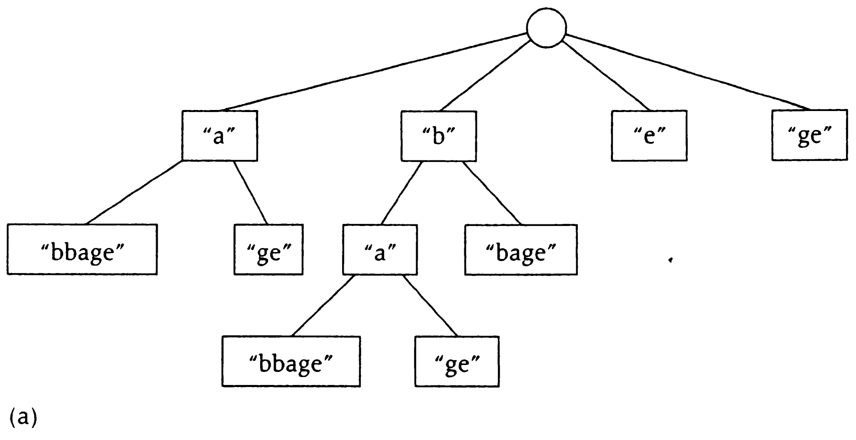
20.6.3 Suffix Trees

A *suffix tree* (also called a *suffix trie*) with respect to a given text string T is a compressed trie for the string collection C consisting of all suffixes of T. This definition requires that no suffix be a prefix of any other suffix. For strings in which this occurs, we simply add a special symbol to the end of every suffix in C. Suffix trees are useful in practice because they can be used to determine whether a pattern string P is a substring of a given text string T. The suffix tree for the string T = "babbage" is shown in Figure 20.9(a). Because the string label on each node corresponds to a substring $T[i:j]$ of T, it can be represented more compactly using just the pair (i, j). Figure 20.9(b) shows the more compact representation of the node labels for the suffix tree in part (a).

Given a pattern $P[0:m - 1]$ in string P, it is easy to design an $O(km)$ algorithm that traces a path in the suffix tree corresponding to test T to determine whether P occurs as a substring of T. We leave the design of such an algorithm as an exercise.

FIGURE 20.9

(a) Suffix tree for string T = "babbage".



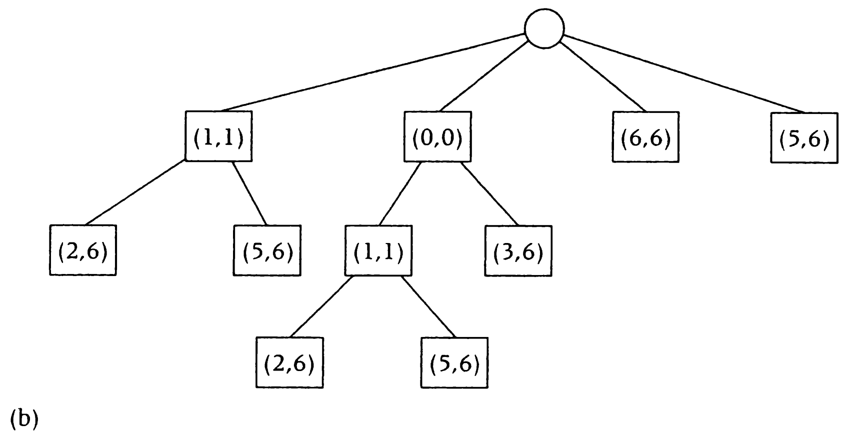


FIGURE 20.9 Continued. (b) more compact representation of the node labels for the suffix tree in part (a).

20.7 Closing Remarks

String and pattern matching have been areas of interest for a long time, and string algorithms have recently received increased attention because of their role in Web searching as well as in computational biology. In this chapter, we have introduced several of the most important string-matching algorithms, but the subject is vast. The interested reader should consult the references for more extended treatments of this important topic and its applications.

References and Suggestions for Further Reading

Books on string matching:

Aoe, J. *Computer Algorithms: String Pattern Matching Strategies*. Wiley-IEEE Computer Society Press, 1994.

Crochemore, M. *Text Algorithms*. New York: Oxford University Press, 1994.

Gusfield, D. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge: Cambridge University Press, 1997.

Navarro, G., and Raffinot, M. *Flexible Pattern Matching in Strings*. Cambridge: Cambridge University Press, 2002.

Stephen, G. A. *String Searching Algorithms*. London: World Scientific Publishing, 1994.

Chen, D., and Cheng, X., eds. *Pattern Recognition and String Matching*. Dordrecht, The Netherlands: Kluwer Academic Publishers, 2002. A collection of 28 articles contributed by experts on pattern recognition and string matching.

Survey articles on string matching:

Baeza-Yates, R. A. "Algorithms for String Matching: A Survey." *ACM SIGIR Forum* 23 (1989): 34–58.

Navarro, G. "A Guided Tour to Approximate String Matching." *ACM Computing Surveys* 33, no. 1 (2001): 31–88.

XERCISES

Section 20.2 The Knuth-Morris-Pratt Algorithm

20.1. Suppose $P[0{:}m-1]$ and $T[0{:}n-1]$ are the strings $0^{m-1}1$ and 0^n that were the worst-case strings of length m and n for *NaiveStringMatcher*.

a. Show that $P[0{:}m-1]$ and $T[0{:}n-1]$ are best-case strings of length m and n for *KMPStringMatcher* for the case where P does not occur in T.

b. Find worst-case strings $P[0{:}m-1]$ and $T[0{:}n-1]$, respectively, for *KMPStringMatcher*, and thereby determine $W(m,n)$ for *KMPStringMatcher.*

20.2 Compute the array *Next*[0:10] for the pattern string P = "abracadabra".

20.3 Trace the action of *KMPStringMatcher* as in Figure 20.4 for the pattern string P = "cincinnati" and the text string T = "cincinatti_is_cincinnati_misspelled".

20.4 Verify the correctness of the algorithm *CreateNext.*

20.5 Show that *CreateNext* has $O(m)$ complexity.

20.6 Write programs implementing *NaiveStringMatcher* and *KMPStringMatcher*, and run them for various inputs for comparison.

Section 20.3 The Boyer-Moore String-Matching Algorithm

20.7 Design and analyze pseudocode for the simplified BM algorithm.

20.8 Show that the worst case of the simplified BM algorithm is as bad as the naive algorithm.

Exercises 20.9 through 20.14 involve the full version of the BM algorithm. As mentioned, when a mismatch occurs in the last position of the pattern string, the full version works the same way as the simplified version using the value *Shift*[c] to shift the pattern based on the mismatched text character c. The difference arises when we have matched the last $k > 0$ characters of the pattern string P, called a *good suffix* (of length k), before a mismatch occurs with a character c from the text. We then shift the pattern by the larger of the following two shifts, called the *bad character shift* s_1 and the *good suffix shift* s_2, respectively. The bad character shift s_1 is simply defined as $s_1 = \max\{Shift[c] - k, 1\}$. We can shift P by s_1 and not miss any matches, for reasons similar to those used when comparing against the last character in P.

The good suffix shift s_2 is also based on reasoning similar to that used to create the array *Shift*, but where we consider suffixes of P instead of a single character. More precisely, we look for the rightmost repeated occurrence of the good suffix of length k (if any) to shift this occurrence by the amount s_2 required to bring it to the end of the pattern. Of course, the character preceding this repeated occurrence (if any) must be different from the character preceding the good suffix; otherwise, a mismatch will occur again. Even when such a repeated occurrence of the good suffix does not happen, we might not be able to shift the pattern by its entire length m because we might miss a match that might occur when a suffix of length j of a good suffix of length k, $0 < j < k$, matches a prefix of length j of P. In the latter case, we shift by the amount required to bring the prefix to the end of the pattern P. For example, if $P = 011101001$, then the good suffix shifts for $k = 1, \dots, 8$, which are given by 5, 3, 7, 7, 7, 7, 7, 7, respectively.

20.9 Design and analyze pseudocode for an algorithm that creates the good-suffix shift values for a given input pattern $P[0{:}m - 1]$.

20.10 Compute the bad-character shifts and the good-suffix shifts for the following patterns:

a. $P =$ "01212121"

b. $P =$ "001200100"

20.11 Trace the action of the full version of the BM algorithm for the pattern $P =$ "amalgam" and text $T =$ "ada_gamely_amasses_amalgam_information".

20.12 Design and analyze pseudocode for the full version of the BM algorithm.

20.13 Write programs implementing the simplified and full versions of the BM algorithm, and compare their performance for various inputs.

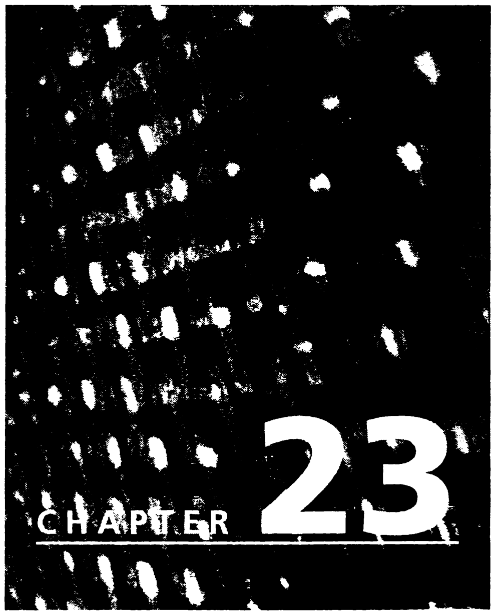
20.14 Compare the performance of the programs written in the previous exercise to programs implementing *NaiveStringMatcher* and *KMPStringMatcher* (see Exercise 20.6) for various inputs.

Section 20.4 The Karp-Rabin String-Matching Algorithm

20.15 Trace the action of *KarpRabinStringMatcher* for the alphanumeric strings $P =$ "108" and $T =$ "002458108235" for the following values of q:

a. $q = 7$

b. $q = 11$



HEURISTIC SEARCH STRATEGIES: A*-SEARCH AND GAME TREES

heuristic: ... *providing aid or direction in the solution of a problem, but otherwise unjustified or incapable of justification. ... of or relating to exploratory problem-solving techniques that utilize self-learning techniques to improve performance.*

Webster's New Collegiate Dictionary

Search strategies such as backtracking, LIFO and FIFO branch-and-bound, breadth-first search, and depth-first search are blind in the sense that they do not look ahead, beyond a local neighborhood, when expanding a node. In this chapter, we show how using heuristics can help narrow the scope of otherwise blind searches. We introduce a type of heuristic search strategy known as A*-search, which is widely used in artificial intelligence (AI). We then discuss strategies for playing two-person games. The alpha-beta heuristic for two-person games is based on assigning a heuristic value to positions reached by looking ahead a certain fixed number of moves. Then an estimate for the best move is obtained by working back to the current position using the so-called *minimax* strategy.

23.1 Artificial Intelligence: Production Systems

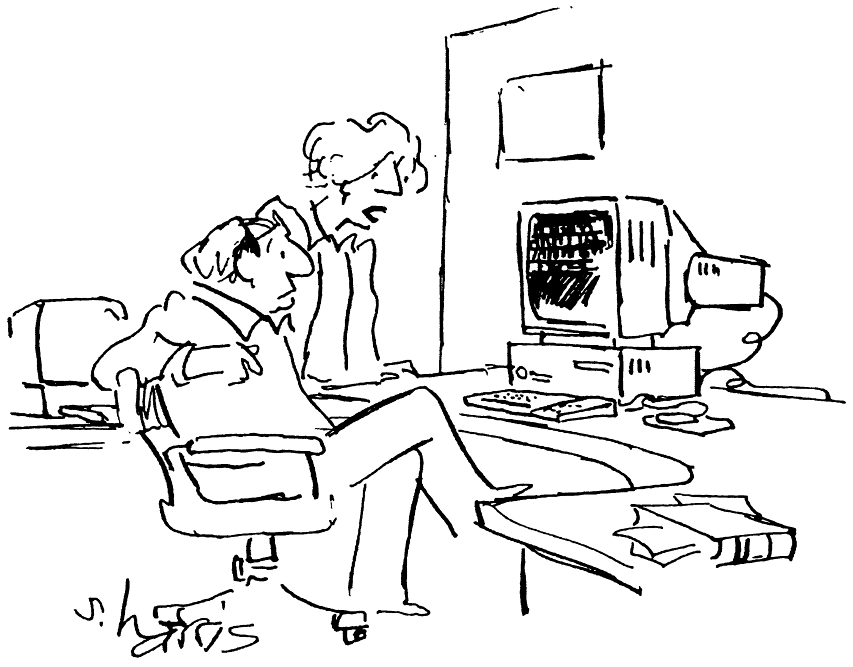
The subject of AI is concerned with designing algorithms that allow computers to emulate human behavior (see Figure 23.1). The major areas of AI include natural language processing, automatic programming and theorem proving, robotics, machine vision and pattern recognition, intelligent data retrieval, expert systems, and game playing. Certain activities that are child's play, such as using a natural language, present theoretically and computationally difficult problems that are beyond the reach of current technology. It is true that voice recognition computer programs are currently available that can properly interpret a limited set of spoken instructions. However, the time when we can carry out ordinary conversations with a computer, such as those between the spaceship crew and the computer Hal in the movie *2001: A Space Odyssey*, has yet to be fully realized.

Many problems in AI involve production systems. An AI production system is characterized by *system states* (also called *databases*), *production rules* that allow the system to change from one state to another, and a *control system* that manages the execution of the production rules and allows the system to evolve according to some desired scenario. For example, a system state might be the position of a robotic arm. A production rule allows the robotic arm to change its position. A control strategy is an algorithm that controls the movement of the arm from a given initial position to a final goal position. Here again, a two-year-old child can move his or her arms without much thinking, but designing an algorithm that allows a robot to accomplish the same thing is a complicated task.

Given any AI production system, there is a positive cost associated with the application of each production rule. A production system can be modeled as a positively weighted digraph, called a *state-space digraph,* where a node in the graph is the state of the system, and a directed edge from node v to node w is assigned the cost $Cost(v, w)$ of the production rule that transforms state v into state w. Given some initial state r (in which the root vertex is in the directed graph), we are interested in whether or not we can find a directed path from r to a goal state. A control system for the problem is then simply a search strategy for reaching a goal state starting from r. As usual, we wish to find control systems that perform searches efficiently.

FIGURE 23.1

© Sidney Harris. *Reprinted with permission. All rights reserved.*



"IT FIGURES. IF THERE'S ARTIFICIAL INTELLIGENCE, THERE'S BOUND TO BE ARTIFICIAL STUPIDITY."

23.2 8-Puzzle Game

We illustrate the ideas of an AI production system by the 8-puzzle game. The 8-puzzle game is a smaller version of the 15-puzzle game invented by Sam Lloyd in 1878. In the 8-puzzle game, there are eight tiles numbered 1 through 8 occupying eight of the nine cells in a 3 × 3 square board. The objective is to move from a given initial state in the board to a goal state. The only moves (production rules) allowed are to move a tile into an adjacent empty cell. It is convenient to characterize such a move as a movement of the empty cell. Thus, there are exactly four rules for moving the empty cell: move left, move right, move up, move down. Of course, the only states allowing all four rules to be applied are when the empty cell is in the center. When the empty cell is at a corner location in the board, then only two of the four rules can be applied. The remaining locations allow the application of three of the four rules.

We will let *D* be the state-space digraph whose vertex set consists of all possible board configurations in the 8-puzzle game. In *D*, a directed edge (*v*, *w*) exists if a move in the game transforms *v* into *w*. Note that if (*v*, *w*) is an edge, then (*w*, *v*) is also an edge. Thus, *D* can be considered a state-space graph, where the two directed edges (*v*, *w*) and (*w*, *v*) are replaced with the single undirected edge {*v*, *w*}. A portion of the state-space graph is shown in Figure 23.2. Suppose we use the breadth-first search control strategy to search for a path leading from the initial state to the goal state. We assume that our control strategy always generates the children of a node in the following order: move left, move right, move up, move down. In Figure 23.3, we have taken an initial position that is only four moves from the goal state. However, 28 states (not counting the initial state) are generated by breadth-first search.

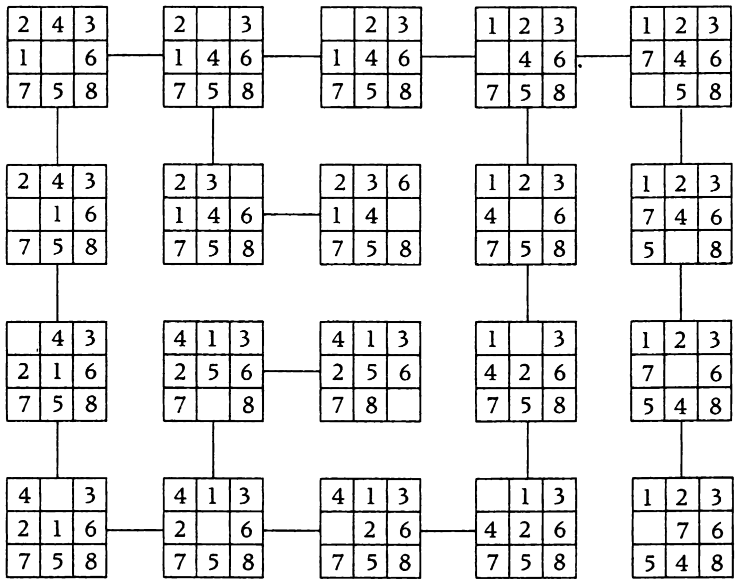


FIGURE 23.2

A portion of the state-space graph for the 8-puzzle game.

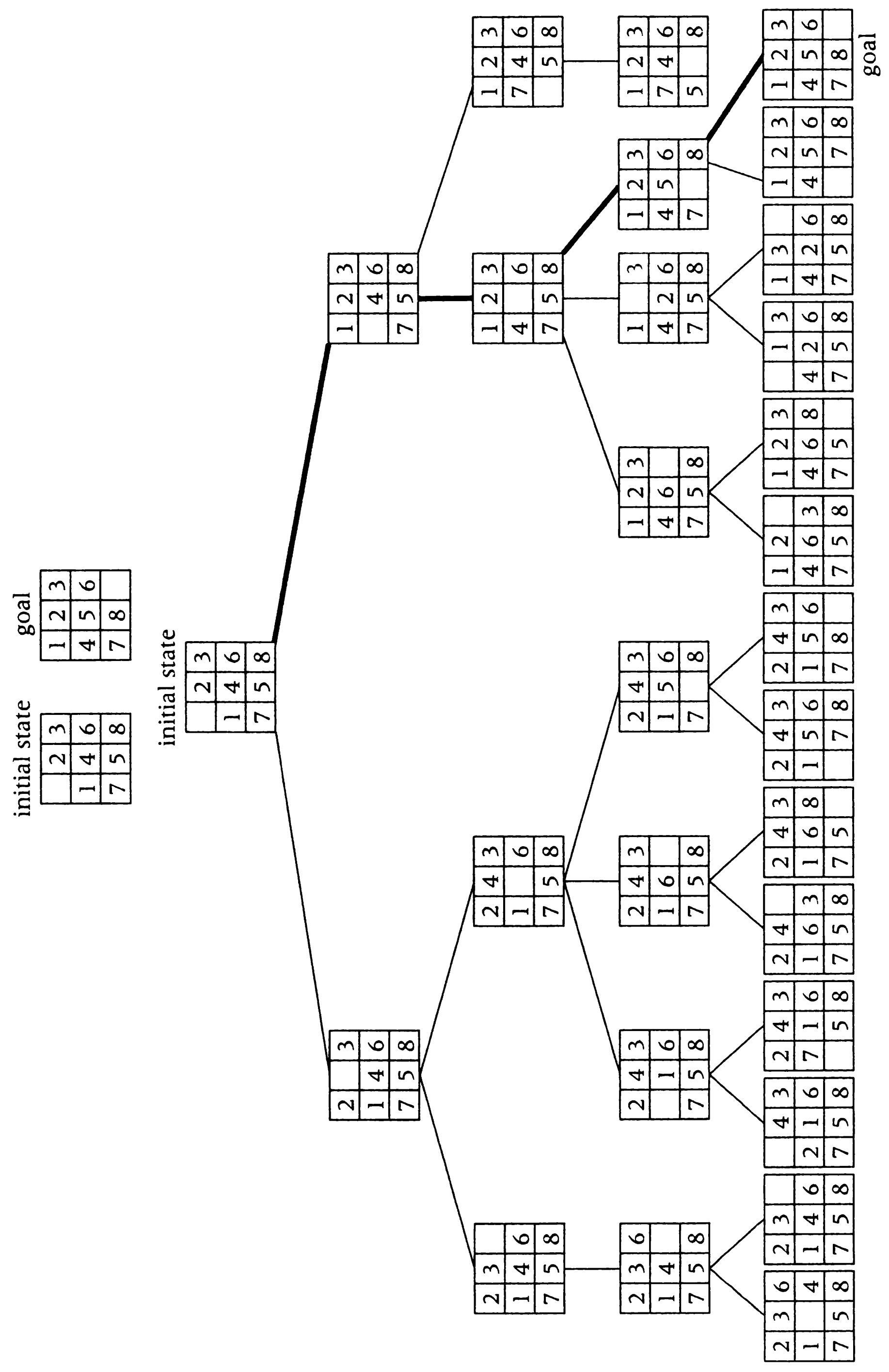


FIGURE 23.3

The states generated by the breadth-first search for a path from the given initial state to the goal state in the 8-puzzle game. The path to the goal is shown in bold.

In general, the number of nodes generated by a breadth-first search for the n-puzzle game is exponential in the minimum number of moves required to reach a goal. Thus, we must look for a better search strategy to solve the problem for initial states requiring many moves to reach the goal. We now describe such a strategy.

23.3 A*-Search

Given a root vertex r and a set of goal states in a state-space digraph, an A*-search is a strategy for finding a shortest path from r to a nearest goal. Such a path is called an *optimal path.* An A*-search finds an optimal path using a generalization of Dijkstra's algorithm. In the discussion of Dijkstra's algorithm in Chapter 12, we maintained an array $Dist[0{:}n - 1]$. At each stage, the vertex v minimizing $Dist[v]$ over all vertices not in the tree was added to T. The values of $Dist[w]$ were then updated for all vertices w in the out-neighborhood of v. The operations performed on the array $Dist[0{:}n - 1]$ were essentially those of a priority queue. To aid in the description of the A*-search strategy, we now give a high-level description of Dijkstra's algorithm based on maintaining a priority queue of vertices. We also modify Dijkstra's algorithm to terminate once a goal is dequeued.

We denote the priority of a vertex v in the queue by $g(v)$, where the smaller values of g have higher priority. At initialization, the root vertex r is enqueued with priority $g(r) = 0$. When a vertex v is enqueued, a parent pointer from v to $Parent(v)$ is also stored. The parent pointers determine a tree. We call the subtree of the tree spanned by the vertices that have been dequeued the *dequeued tree T.* The tree T contains a shortest path in D from r to each vertex in T.

At each stage in the algorithm, a vertex v in the priority queue is dequeued, and the out-neighbors of v are examined. If an out-neighbor w of v is already in the tree T, then nothing is done to w. If the out-neighbor w has not been enqueued, then it is enqueued with priority $g(w) = g(v) + c(v, w)$, and a parent pointer from w to v is set. Finally, if the out-neighbor w is already on the queue, then the priority of vertex w is updated to $g(w) = \min\{g(w), g(v) + c(v, w)\}$. If $g(w)$ is changed to $g(v) + c(v, w)$, then the parent pointer of w is reset to point to v. The algorithm terminates once a goal is dequeued. The path in the final tree T from r to the goal is an optimal path. The action of the algorithm is shown in Figure 23.4 for a sample digraph.

Dijkstra's algorithm is too inefficient for most AI applications because the shortest-path tree can grow to be huge, even exponentially large. The reason for

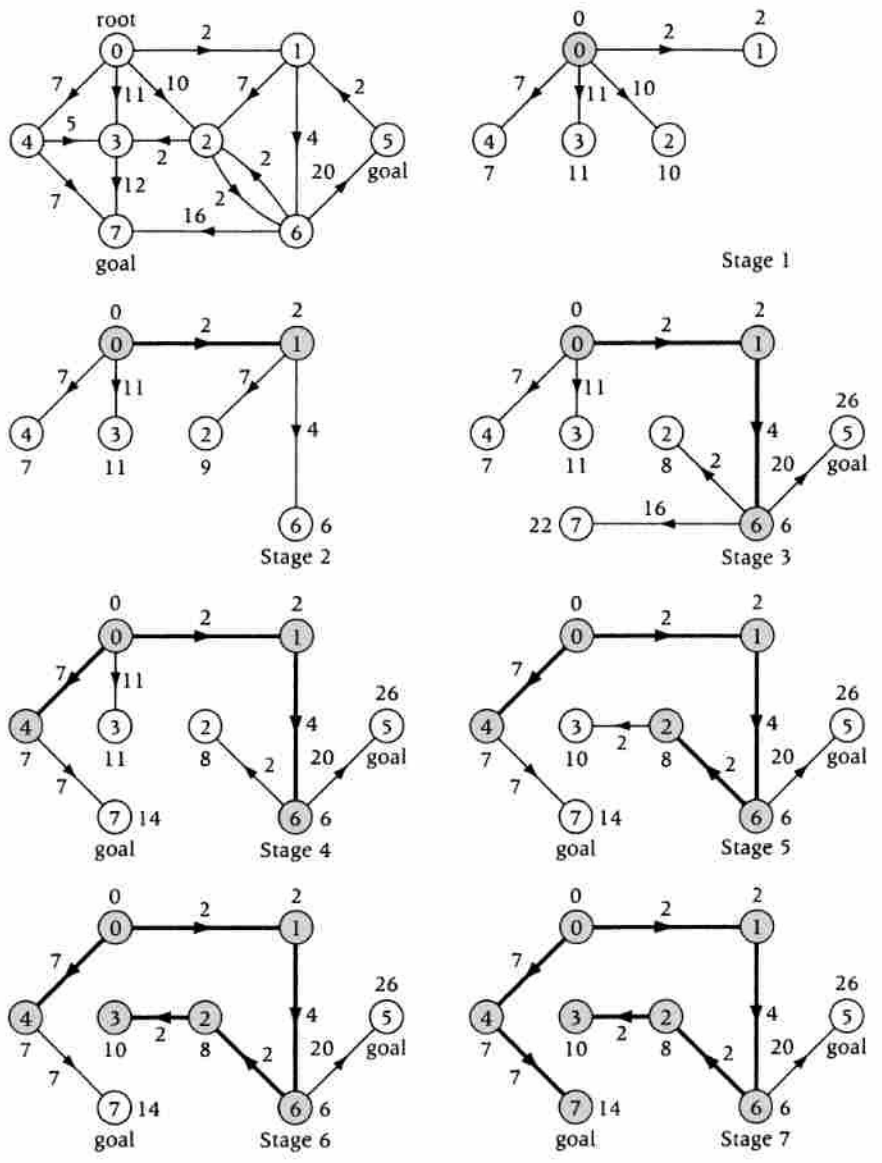


FIGURE 23.4

The action of Dijkstra's algorithm is shown for a sample weighted digraph *D*. The distance *g*(*v*) is shown outside each node *v*. The vertices and the edges of the dequeued (shortest-path) tree are shaded. The priority queue at each stage consists of vertices *w* not in the dequeued tree, where the priority of *w* is *g*(*w*).

its inefficiency is that only local information is assumed when looking ahead to a goal. No global information is used that can help the shortest-path tree send out branches in a promising direction. In other words, in Dijkstra's algorithm, the shortest-path tree tends to grow fat (a "shotgun approach" to a goal) rather than grow skinny (a "beeline" approach to a goal).

23.3.1 Heuristics

When information beyond merely the costs of the edges in the digraph is available, Dijkstra's algorithm can be improved so that the shortest-path tree is less expansive and the search is more efficient. The idea is that the priority value $g(v)$ of a vertex v in the queue, which is the cumulative distance (cost) from the root r via the current path determined by the parent pointers, can be replaced by an overall estimate of the cost of the shortest path from r to a goal constrained to go through v. In Dijkstra's algorithm, the priority value of v is $g(v)$, but now we define the priority value of v to be the *cost function*

$$f(v) = g(v) + h(v), \qquad \textbf{(23.3.1)}$$

where $h(v)$ is some estimate of the cost of a shortest path from v to a goal vertex. Because the shortest path from v to a goal has not been found, the best that we can do is use a *heuristic* value for $h(v)$.

When no restriction is placed on the heuristic h in Formula (23.3.1), h is merely a heuristic for a greedy algorithm. When the vertices in the dequeued tree T are reexamined and their parent pointers updated when shorter paths for them are found, the algorithm based on (23.3.1) is called an *A-search*. There is no guarantee that the first path found to a goal is optimal using an A-search. When an A-search uses a heuristic $h(v)$ that is a lower bound of the cost of the shortest path from v to a goal, then the algorithm is called an *A*-search*. We assume that an A*-search terminates when it dequeues a goal, or when the queue is empty. The proof of the following theorem is left to the exercises.

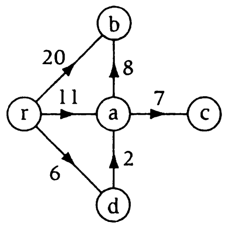
Theorem 23.3.1 Given a positively weighted digraph G (finite or infinite), if a goal is reachable from a root vertex r, then an A*-search terminates by finding an optimal path from r to a goal. □

In an A*-search, when a node v is dequeued, some of its neighbors may already be in the tree T. Unlike Dijkstra's algorithm, an A*-search must check these neighbors to see if shorter paths to them now exist via the vertex v just dequeued. If shorter paths are found, then T must be adjusted to account for them (see Figure 23.5).

Considerable computational cost may be incurred by an A*-search when adjusting the dequeued tree T. This computational cost can be avoided by placing a rather natural and mild restriction on the heuristic h used by A*-search. The heuristic value $h(v)$ is an estimate of the cost of going from v to the nearest goal.

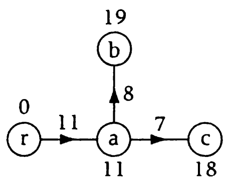
FIGURE 23.5

The number shown outside each node v in T is the value $g(v)$. This example shows how the dequeued tree T can change during an A*-search. Note that the values of g at vertices a, b and c required updating.

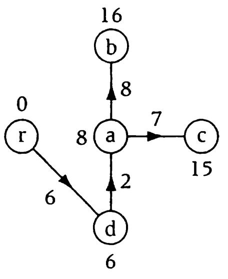


v	r	a	b	c	d
$h(v)$	–	23	20	15	29

Portion of state space induced by vertices r, a, b, c, d, and an associated function h



The dequeued tree T after vertices r, a, b, c have been dequeued



The dequeued tree T after d has been dequeued

If the edge (v, w) exists, then one estimate is $c(v, w) + h(w)$. The restriction on a heuristic, called the *monotone restriction*, says that $h(v)$ should be at least as good as this estimate.

DEFINITION 23.3.1 A heuristic $h(v)$ for an A*-search for a given digraph with cost function c on the edges is said to satisfy the *monotone restriction*, if

$$\begin{array}{ll} h(v) \leq c(v, w) + h(w), & \text{whenever the edge } (v, w) \text{ exists,} \\ h(v) = 0, & \text{whenever } v \text{ is a goal.} \end{array} \qquad \textbf{(23.3.2)}$$

If h satisfies the monotone restriction, then we merely say that h is monotone.

If h is a monotone heuristic, then $h(v)$ is a lower bound of the cost of the shortest path from v to a goal. The following proposition states that an A*-search using a monotone heuristic does not need to update parent pointers of a vertex v already in the dequeued tree T, because the path in T from r to v is already a shortest path in D from r to v. The result is consistent with the fact that the A*-search algorithm reduces to Dijkstra's algorithm when $h(v) \equiv 0$ (and the identically zero function trivially satisfies the monotone restriction).

oposition 23.3.2 Suppose an A*-search uses a monotone heuristic. Then the dequeued tree T is a shortest-path tree in the state-space digraph D. In particular, parent pointers for vertices in the dequeued tree T never need updating.

PROOF

For any vertex $v \in V$, let $g^*(v)$ denote the length of the shortest path in D from r to v (so that $g(v) \geq g^*(v)$ at every stage in the execution of an A*-search). We wish to show that $g(v) = g^*(v)$ at the time when v is dequeued. If $v = r$, then we have $g(v) = g^*(v) = 0$; thus, we can suppose that $v \neq r$. Let $P = v_0, v_1, \ldots, v_j$ be a shortest path in D from $r = v_0$ to $v = v_j$. Let vertex v_k be the last vertex in P such that $v_0, v_1, \ldots, v_k$ were all in the tree T when v was dequeued (v_k exists since $r = v_0 \in T$). Then v_{k+1} was in the queue Q at the time when v was dequeued. For any pair of consecutive vertices v_i, v_{i+1} in P, using the monotone restriction, we have

$$g^*(v_i) + h(v_i) \leq g^*(v_i) + h(v_{i+1}) + c(v_i, v_{i+1}). \qquad \textbf{(23.3.3)}$$

Now v_i and v_{i+1} are in a shortest path in G, so that

$$g^*(v_{i+1}) = g^*(v_i) + c(v_i, v_{i+1}). \qquad \textbf{(23.3.4)}$$

Substituting Formula (23.3.4) in Formula (23.3.3), we obtain

$$g^*(v_i) + h(v_i) \leq g^*(v_{i+1}) + h(v_{i+1}). \qquad \textbf{(23.3.5)}$$

Iterating Formula (23.3.5) and using the transitivity of $\leq$ yields

$$g^*(v_{k+1}) + h(v_{k+1}) \leq g^*(v_j) + h(v_j) = g^*(v) + h(v). \qquad \textbf{(23.3.6)}$$

Now v_{k+1} is on a shortest path P, and $v_0, v_1, \ldots, v_k$ all belong to T, so that $g(v_{k+1}) = g^*(v_{k+1})$. Hence, Formula (23.3.6) implies

$$f(v_{k+1}) = g(v_{k+1}) + h(v_{k+1}) \leq g^*(v) + h(v) \leq g(v) + h(v) = f(v) \qquad \textbf{(23.3.7)}$$

Thus, we must have had $g(v) = g^*(v)$ when v was dequeued; otherwise, $f(v_{k+1}) < f(v)$, and v would not have been dequeued in preference to v_{k+1}. ■

The following proposition helps explain the terminology *monotone restriction*. We leave the proof of Proposition 23.3.3 as an exercise.

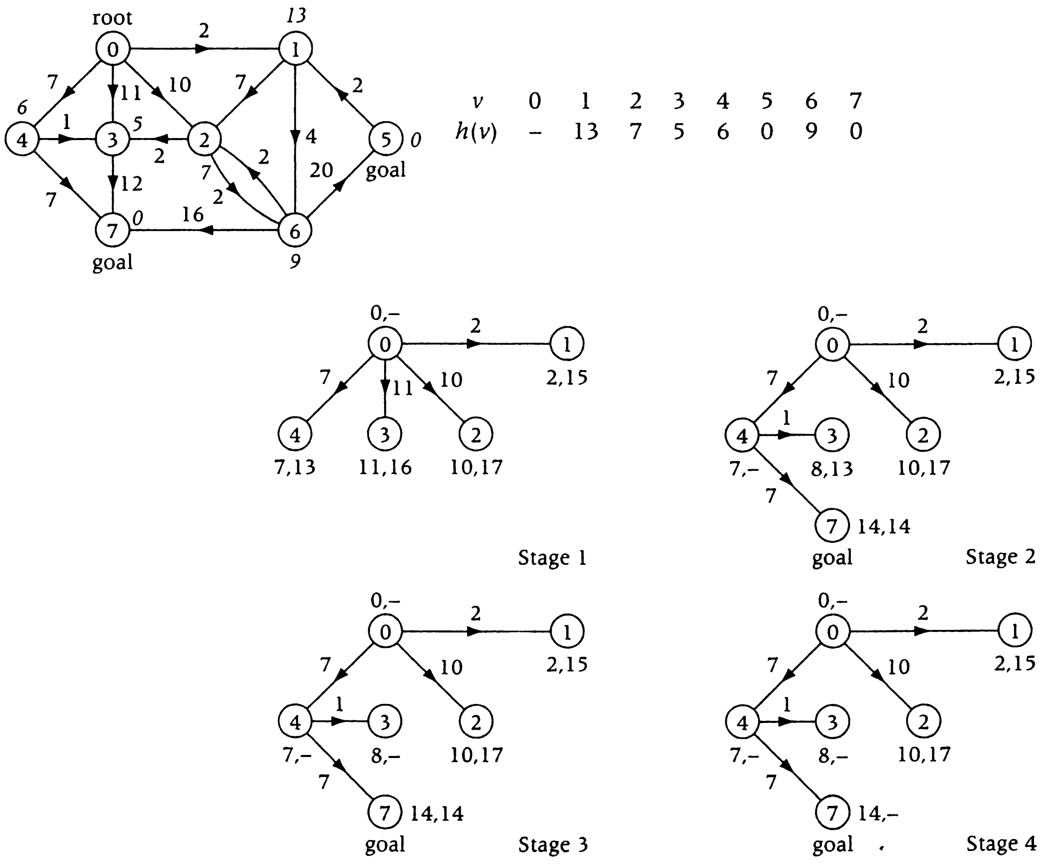
Proposition 23.3.3 The f-values of the vertices dequeued by an A*-search using a monotone heuristic are nondecreasing. □

In any problem using an A*-search, the digraph and associated cost function are either implicitly or explicitly input to the algorithm. Here we give examples of both scenarios. When the digraph is very large, it is usually implicitly defined, and only the part of the digraph generated by the execution of the A*-search is made explicit. The following is a high-level description of A*-search using a monotone heuristic. At any given point in the execution of procedure *A*-SearchMH*, T is a subtree of D rooted at the root vertex r containing a path from r to each vertex that has been dequeued by the algorithm. Assuming that a path from r to a goal exists, Theorem 23.3.1 and Proposition 23.3.2 show that when *A*-SearchMH* terminates after dequeuing a goal, the corresponding path to the goal is optimal.

```
procedure A*-SearchMH(D, c, r, GoalSet, h, T)
Input:   $D = (V, E)$ (a digraph, either implicitly or explicitly defined)
                  $c$ (a positive cost function on $E$)
                  $r$ (a root vertex in $D$)
                  $h$ (a heuristic function satisfying monotone restriction)
                  GoalSet (a set of goal vertices in $D$)
Output:  a shortest-path out-tree $T$ rooted at $r$ containing an optimal path to a goal
         vertex, if one exists
  $Q$ (a priority queue of vertices, with $v$ having priority value $f(v) = g(v) + h(v)$, where
  $g(v)$ is the cost of shortest path $P(v)$ from $r$ to $v$ currently generated. $Q$ also contains
  a parent pointer from $v$ to $w \in T$, where edge $(w, v)$ belongs to $P(v)$)
  while $Q$ is not empty do
      dequeue vertex $v$ in $Q$ with minimum priority value $f(v)$
      add vertex $v$ to $T$ using parent pointer
      if $v \in$ GoalSet then
          return
      endif
      for all vertices $w \notin T$ and adjacent to $v$ do
          if $w \notin Q$ then
              enqueue $w$ with parent $v$ and priority value $f(w) = g(w) + h(w)$
              where $g(w) = g(v) + c(v, w)$
```

```
            else
                if $f(w) \geq g(v) + c(v, w) + h(w)$ then
                    reset parent pointer of $w$ to $v$ and update priority value of
                    $w$ to $f(w) = g(w) + h(w)$, where $g(w) = g(v) + c(v, w)$
                endif
            endif
        endfor
    endwhile
    return "failure"
end A*-SearchMH
```

The action of procedure *A*-SearchMH* is illustrated in Figure 23.6 for a sample digraph D.



v	0	1	2	3	4	5	6	7
$h(v)$	–	13	7	5	6	0	9	0

FIGURE 23.6

The action of A*-search with root vertex $r = 0$ and a monotone heuristic $h(v)$ is shown for the same weighted digraph D as in Figure 22.4.

We now illustrate procedure *A*-SearchMH* with two examples. First, we revisit the 8-puzzle problem. Then we consider the problem of finding shortest paths between cities in the United States using the freeway system. In the 8-puzzle problem, the state-space graph G is implicitly defined. In the freeway problem, the state-space graph is explicitly input to the algorithm.

23.3.2 A*-Search and the 8-Puzzle Game

Consider the heuristic

$$h(v) = \text{ the number of tiles not in correct cell in the state } v.$$

It is easy to verify that $h(v)$ satisfies the monotone restriction. Using $h(v)$, Figure 23.7 shows the shortest-path tree generated by the A*-search for the same input as shown in Figure 23.3.

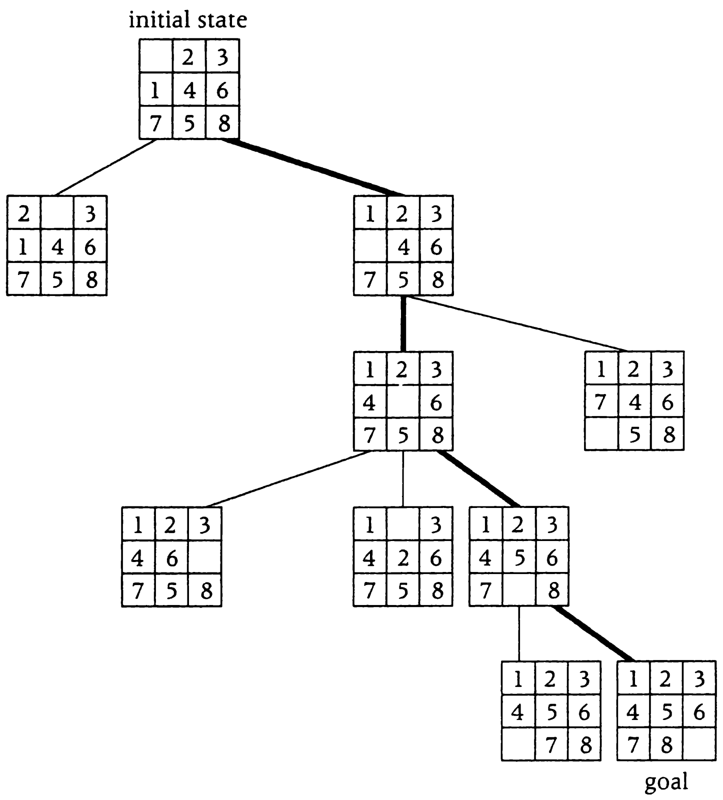


FIGURE 23.7 Shortest-path tree generated by the A*-search for the 8-puzzle game with the same initial state and goal state as in Figure 23.3.

Note that the A*-search only generated 9 states compared to the 28 states generated by the breadth-first search for the same input. Other monotone heuristics exhibit even better behavior in general than the one used in Figure 23.7. For example, for any given state, the sum of the Manhattan distances (vertical steps plus horizontal steps) from the tiles to their proper positions in the goal state exhibits good performance.

23.3.3 Shortest Paths in the Freeway System

Our second example of an A*-search is for the problem of finding a shortest path on freeways between two cities in the continental United States (see Figure 23.8). The heuristic $h(v)$ we use will be a lower bound of the geographical (great-circle) distance between v and the destination city t. We assume that the distances between adjacent cities is available to the algorithm via a suitable adjacency cost matrix. The lower-bound estimate is computed using the longitude and latitude of each city, which we assume are both input to the algorithm as additional information. A lower bound of 50 miles is used for the longitude distance of one degree apart. A lower bound of 70 miles is used for the latitude distance of one degree apart. The square root of the sum of the squares of the longitude distance and the latitude distance between cities v and t is used as $h(v)$. The heuristic $h(v)$ is monotone because the cost (mileage on a freeway between adjacent cities) cannot be smaller than the geographical distance between them. Indeed, if the cost between adjacent cities v and w is $c(v, w)$, then $h(v)$, $h(w)$, and $c(v, w)$ form an almost planar triangle, and we have $h(v) \leq h(w) + c(v, w)$, which is the monotone restriction.

Figure 23.8 shows the graph of the United States as input to a Prolog program, as well as a shortest path from Cincinnati to Houston. Figure 23.9 shows the shortest-path tree generated by Dijkstra's algorithm ($h \equiv 0$), whereas Figure 23.10 shows the shortest-path tree generated by the A*-search. The number of vertices in the shortest path between Cincinnati and Houston is 9. The number

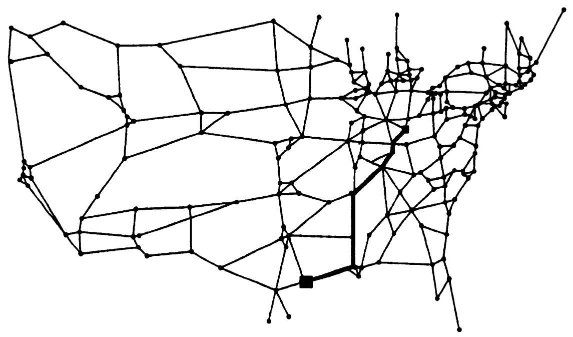


FIGURE 23.8 U.S. freeway system.

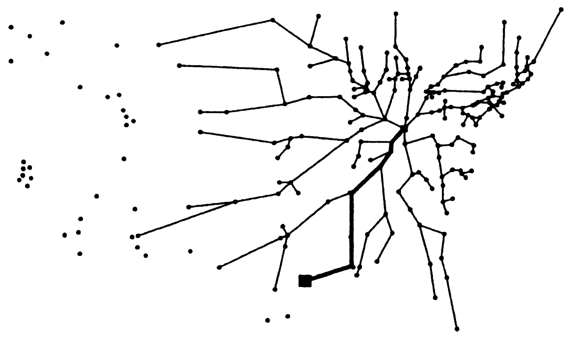


FIGURE 23.9 Shortest-path tree generated by Dijkstra's algorithm.

of vertices expanded when the heuristic is used is 34, compared with 213 vertices when expanded by Dijkstra's algorithm. Thus, the portion of the graph expanded using the A*-search is only 16 percent of that expanded using Dijkstra's algorithm for this example. The contrast between the trees grown in Figure 23.9 and in Figure 23.10 shows rather nicely the difference between the "shotgun" approach to a goal made by Dijkstra's algorithm versus the "beeline" approach made by A*-search.

The selection of a good heuristic is crucial to the success of an A*-search. The closer $h(v)$ is to the actual cost of the shortest path from v to a goal, the fewer nodes will be expanded during the A*-search. However, determining heuristics close to the actual cost is usually too expensive computationally, because determining close estimates is as hard as the original problem. It sometimes speeds the search to use a function for h that does not have the lower-bound property—that is, using an A-search instead of an A*-search. For example, there are better heuristics leading to A-searches for the 8-puzzle game than the monotone heuristic $h(v)$ that was the basis for our A*-search.

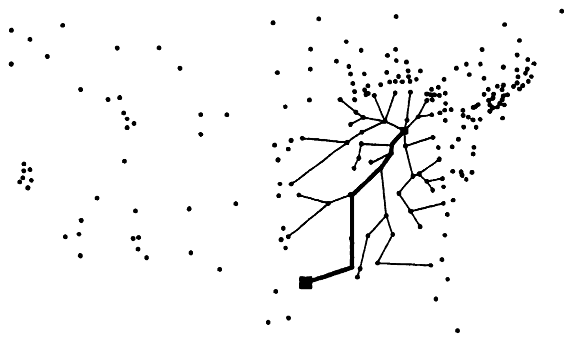


FIGURE 23.10 Shortest-path tree generated by A*-search.

23.4 Least-Cost Branch-and-Bound

Least-cost branch-and-bound is basically an A*-search applied to a state-space tree with the additional use of a bounding function. We use the same notation when describing the state-space tree T as we used in Chapter 10. Least-cost branch-and-bound applies to problems involving minimizing an objective function φ over each solution state in the state-space tree. The cost $c(v)$ of a given node $v = (x_1, \ldots, x_k)$ in the state-space tree is taken to be a lower-bound estimate for

$$\varphi^*(v) = \min\{\varphi(w) | w \in T_v \text{ and } w \text{ is a solution state}\}, \quad \textbf{(23.4.1)}$$

where T_v is the subtree of the state-space tree rooted at v, so that

$$c(v) \leq \varphi^*(v). \quad \textbf{(23.4.2)}$$

Often, the cost function c has the same form as with an A*-search,

$$c(v) = g(v) + h(v), \quad \textbf{(23.4.3)}$$

where $g(v)$ is the cost associated with going from the root to the node v, and $h(v)$ is a heuristic lower-bound estimate of the incremental cost of going from v to a solution state v^* in T_v where φ is minimized (over T_v). Typically, $g(v)$ is $\varphi(v)$, where $\varphi(v)$ is an extension of the objective function φ to all problem states.

For example, in the coin-changing problem (see Chapter 7), $g(v)$ is the number of coins used in the problem state v. A natural heuristic $h(v)$ is obtained in a manner similar to the greedy method. Let $r(v)$ denote the remaining change required. We use as many of the largest-denomination coin as possible, then as many of the next-largest-denomination coin as possible, and continue in this manner as long as we do not exceed $r(v)$. The number of coins so obtained is our heuristic $h(v)$ used for the lower bound $c(v) = g(v) + h(v)$.

As a second example, consider the 0/1 knapsack problem formulated as a minimization problem (see Chapter 10). We then have

$$g(x_1, \ldots, x_k) = \mathit{LeftOutValue}(x_1, \ldots, x_k)$$

and

$$h(x_1, \ldots, x_k) = -\mathit{Greedy}(C', B'),$$

where *LeftOutValue*$(x_1, \ldots, x_k)$ is the sum of the values of the objects not in the set $\{b_{x_1}, \ldots, b_{x_k}\}$, B' is the set of objects $\{b_{x_k+1}, \ldots, b_n\}$, $C' = C - (w_{x_1} + \ldots + w_{x_k})$, and *Greedy*$(C', B')$ is the value of the greedy solution to the (C', B') knapsack problem.

In least-cost branch-and-bound, the live nodes in the state-space tree are maintained as a priority queue with respect to the cost function c. In contrast to our method in A*-search, here we maintain a global variable *UB*, which is the smallest value of the objective function over all solution states already generated. Then a node v can be bounded if $c(v) \geq UB$; moreover, we have the following key fact.

Key Fact

Given a lower-bound cost function, if a node of least cost among the live nodes is bounded, then the algorithm can terminate, having already generated an optimal solution state (goal node).

The following paradigm for a least-cost branch-and-bound search strategy uses the same notation and implementation details for the state-space tree as in Chapter 10.

```
procedure LeastCostBranchAndBound
Input:   function $D_k(x_1, \ldots, x_{k-1})$ (determining state-space tree $T$ associated with the
           given problem)
         objective function $\varphi$ defined on the solution states of $T$
         cost function $c(v)$ such that:
         $c(v) \leq \varphi^*(v) = \min\{\varphi(w) \mid w \in T_v$ and $w$ is a solution state$\}$
Output:  a solution state (goal) where $\varphi$ is minimized
  LiveNodes is initialized to be empty
  AllocateTreeNode(Root)
  Root→Parent ← null
  AddPriorityQueue(LiveNodes, Root)         //add root to priority queue of live
                                              nodes
  Goal ← Root                               //initialize goal to root
  UB ← ∞
  Found ← .false.
  while LiveNodes is not empty .and. .not. Found do
      Select(LiveNodes, E-node, k)          //select E-node of smallest cost
                                              from live nodes
      if c(E-node) ≥ UB then                //Goal points to optimal solution
                                              state
          Found ← .true.
```

```
            else
                if E-node is a solution state and $\varphi$(E-Node) < UB then
                                                            //update UB
                    UB ← $\varphi$(E-Node)
                    Goal ← E-Node
                endif
                for each X[k] ∈ $D_k$(E-node) do          //for each child of the E-node do
                    if c(X[k]) < UB .and. .not. StaticBounded (X[1], . . . , X[k]) then
                        AllocateTreeNode(Child)
                        Child→Info ← X[k]
                        Child→Parent ← E-node
                        AddPriorityQueue(LiveNodes, Child)
                                                            //add child to list of live nodes
                    endif
                endfor
            endif
        endwhile
        Path(Goal)                              //output path from goal node to root
    end BranchAndBound
```

23.5 Game Trees

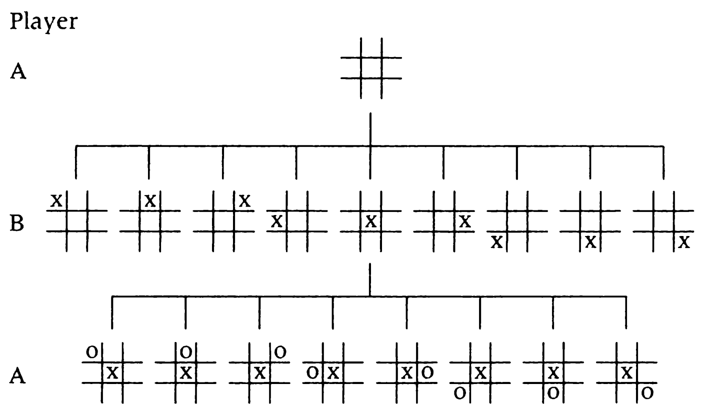
Since the invention of the electronic computer, there has been interest in computerized strategies for playing two-player games. For example, particular interest has been focused on designing computer programs to play chess. The first computer programs written for playing chess were not very sophisticated, partly because early computers were not powerful enough to store the vast amounts of information necessary to play a good game. Computer programs implemented on levels for such games as backgammon and computer programs for playing bridge, have exhibited excellent performance.

Computerized game-playing strategies are usually based on the efficient partial search of the enormous game tree modeling all possible legal moves for a given two-player game. The size of the game tree of all possible moves is generally much too large to admit complete searches. Thus, computerized game-playing strategies use heuristics to estimate the value of various moves based on looking down a limited number of levels in the game tree.

Game trees can become quite large even for simple games. For example, consider the game tree associated with tic-tac-toe on a 3×3 board (see Chapter 10). Suppose two players, A and B, are placing Xs and Os, respectively, and player A moves first. Player A has nine possible choices to place the first X. (Of course, using symmetries of the square, only three of these moves are nonequivalent. We

FIGURE 23.11

A portion of the game tree for tic-tac-toe on the 3 × 3 board. Children of a single node at level 1 are shown.



choose to ignore this reduction for the moment.) After player A moves, then player B has eight possible choices for placing the first O. Continuing in this fashion, we see that the game tree has an upper bound of 9! = 362,880 nodes, although the actual number of nodes is smaller because the game terminates whenever a player achieves three Xs or three Os in a row. Figure 23.11 shows all the nodes at levels 0 and 1 of the game tree, but only that portion of the nodes at level 2 corresponding to the eight possible moves from a particular single node at level 1. Figure 23.12 shows all the nodes in the levels 0, 1, and 2 of the pruned game tree that results by pruning symmetric board configurations.

Now consider the game tree modeling an arbitrary game between two players, A and B, who alternately make moves, with each player having complete

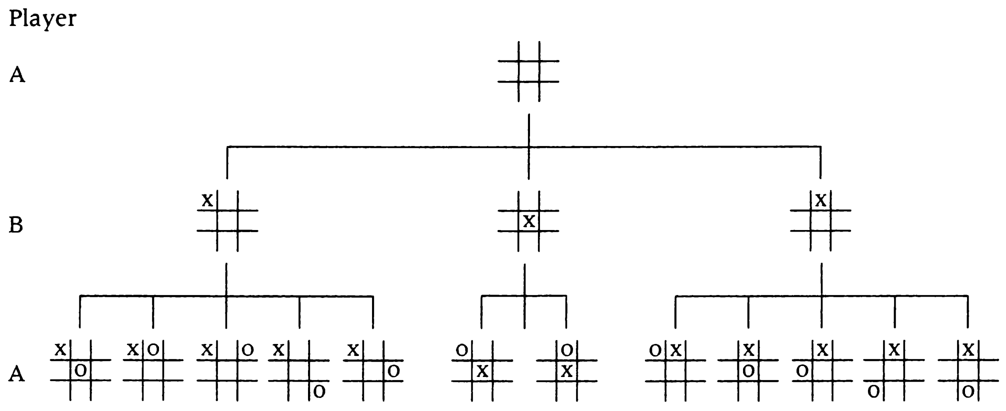


FIGURE 23.12

A portion of the game for tic-tac-toe on the 3 × 3 board pruned by symmetric board configurations. All nodes at the first three levels are shown.

knowledge of the moves of the other (a *perfect information* game). The root of the game tree corresponds to the opening move in the game, which is assumed to be made by player A. Nodes at even levels in the game tree correspond to configurations where it is A's move and are called *A-nodes*. Nodes at odd levels correspond to configurations where it is B's move and are called *B-nodes*. The children of an A-node (respectively, B-node) correspond to all admissible moves available to player A (respectively, player B) from the node. Similar to the game tree for tic-tac-toe, for general games we assume a particular ordering of all admissible moves from a given node, so our game trees are always ordered trees. A leaf node in the game tree is called a *terminal* node and corresponds to the end of the game. In general, a terminal node corresponds to a win, loss, or tie for player A, although certain games such as nim cannot end in a tie.

Given a game tree, the value to player A is assigned to each terminal node (outcome of the game). We also assume that the game is a *zero-sum* game, so that the value to player B of a terminal node is the negative of its value to player A. *However, until further notice, when we speak of the value of a node it is always the value to player A.* We wish to design an algorithm that determines player A's optimal first move. In other words, we want to determine the move that player A should make so that the game will end at a terminal node of maximum value for player A, assuming that each player plays perfectly. Of course, if player B does not play perfectly, then the outcome for player A could be even better.

To analyze the entire game, it is enough to design a procedure to determine the optimal opening move. Indeed, after player A makes the opening move, we would simply repeat (from player B's point of view) the optimal strategy at the game subtree rooted at the node corresponding to this move, and so forth.

If the game tree is small enough to be completely traversed in a reasonable amount of time, then there is a simple *minimax procedure* for player A to determine the optimal opening move. We simply perform a postorder traversal of the game tree, in which a visit at an A-node corresponds to identifying a child of maximum value for a move from the A-node and assigning that value to the A-node. Similarly, a visit at a B-node corresponds to identifying a child of minimum value for a move from the B-node and assigning that value to the B-node.

Key Fact

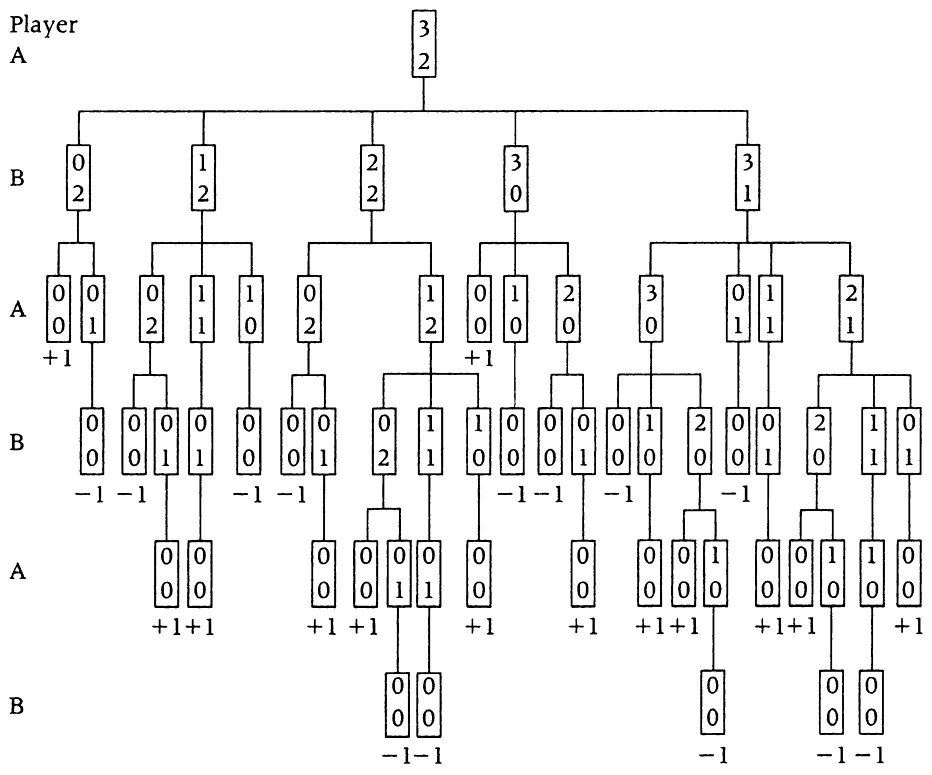
The minimax strategy is nothing more than the definition of perfect play for the two players.

Note that postorder traversal is necessary because the value of each child of a node must be determined before the value of the node itself can be determined. When the postorder traversal is complete, the opening move, together with the value of the game to player A, will be determined.

We illustrate the minimax procedure for the game tree corresponding to a small instance of the game of nim. In the general game of nim, there are n piles of sticks, where the ith pile contains m_i sticks, $i = 1, \dots, n$. Each player alternately chooses a nonempty pile and removes some or all of the sticks from this pile. There is usually a restriction made on how many sticks a player is allowed to remove in a given move. The last player to remove a stick loses. For large $m_1 + m_2 + \cdots + m_n$, the game tree modeling nim would be enormous. To keep things in sight, consider the instance $n = 2, m_1 = 3, m_2 = 2$. The game tree for this instance is shown in Figure 23.13, where the numbers inside each node correspond to the number of sticks left in each pile. Thus, terminal nodes correspond to 0, 0. We assign the value of +1 to a terminal A-node (A wins) and −1 to a terminal B-node (B wins). In Figure 23.14, we have done some pruning of the complete game tree to eliminate generating symmetric child configurations of the two nodes [1, 1] and [2, 2]. For example, in the complete game tree, the node [2, 2] generates the four nodes [0, 2], [1, 2], [2, 0], and [2, 1]. Using symmetry, we need only display the first two nodes in Figure 23.13 when drawing the (pruned) game tree.

FIGURE 23.13

Game tree for [3, 2] nim, pruned to eliminate symmetric children of [1, 1] and [2, 2]. Terminal node values are +1 when A wins and −1 when B wins.



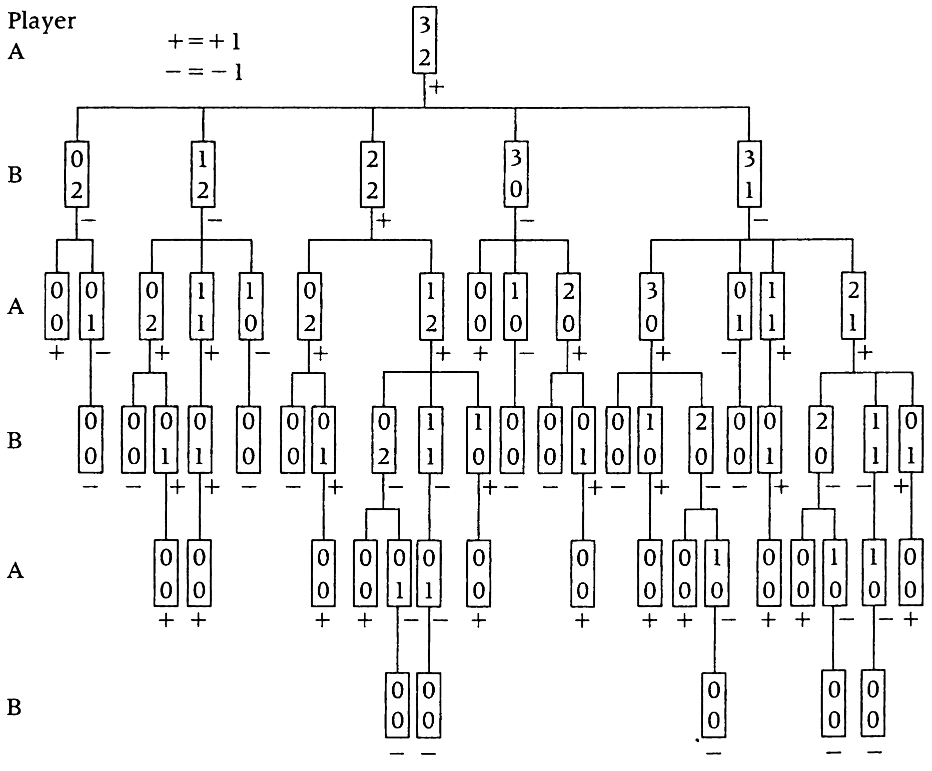


FIGURE 23.14

Values + = +1 and − = −1 are assigned to each node by the minimax postorder traversal of the [3, 2] game of nim.

Figure 23.14 shows the results of a postorder traversal of the game tree in Figure 23.13, where visiting a node executes the minimax procedure described previously. In Figure 23.14, we show the value (to player A) of each node outside the node. Note that the [3, 2] game of nim is a win for player A. The complete traversal of the game tree shows that [2, 2] is a unique opening move that guarantees a win for player A. If a single winner strategy is desired, then the postorder traversal could be terminated as soon as it is determined that the root node has value +1 (with the opening move [2, 2]). Of course, similar termination can be done in any game where we simply have the values +1, 0, and −1 for win, tie, and loss, respectively.

A game in which a complete traversal of the game tree is feasible is usually too small to be interesting. Even the ordinary game of 3 × 3 tic-tac-toe has a rather large game tree. For games like chess, the game tree has been estimated to contain more than 10^{100} nodes. Thus, rather than attempting to traverse the entire game tree when determining an optimal move, in practice the minimax procedure is usually limited to looking ahead a fixed number of levels *r* in the game tree (*r-level search*). Terminal nodes encountered within *r* levels are assigned the value of the outcome of the game corresponding to this node. Nonterminal nodes at level r are assigned some estimate of the value of the node based on the best available knowledge, typically using some heuristic. Nodes that look more promising are given higher values. Of course, the better the estimate, the better the strategy generated by the r-level search.

Suppose we consider 3×3 tic-tac-toe with a two-level search. None of the nodes in the first two levels is a terminal node, so we need to come up with some estimate of the value of each node at level 2. A natural choice would be to assign to a given node (configuration of the board) the number of winning lines completable in Xs minus the number of winning lines completable by Os. For example, Figure 23.15 shows the value of each node at level 2 in the game tree (pruned by symmetries). The figure also shows the result of applying the two-level search (minimax procedure) to the game tree for 3×3 tic-tac-toe, which gives the values of -1, 1, and -2 to the nodes C_1, C_2, and C_3 at level 1, respectively, and gives the root node a value of 1.

We see from Figure 23.15 that player A's opening move would be to place an X in the center position. Then the minimax procedure is continued from the subtree rooted at the latter node. Unfortunately, continuing with the same two-level heuristic search method may lead to a loss for player A (see Exercise 23.24). The fairly obvious fix to this problem is to assign an appropriately large value to terminal positions when encountered in the search We simply give terminal nodes that are wins for player A (that is, three Xs in a row) any value greater than 8, which is the total number of winning lines for the 3×3 game. For example, we could assign the value 9 to terminal nodes that are wins for player A and -9 to terminal nodes that are wins for player B. Terminal nodes corresponding to tie games (cat's games) are assigned the value 0. With the values 9,

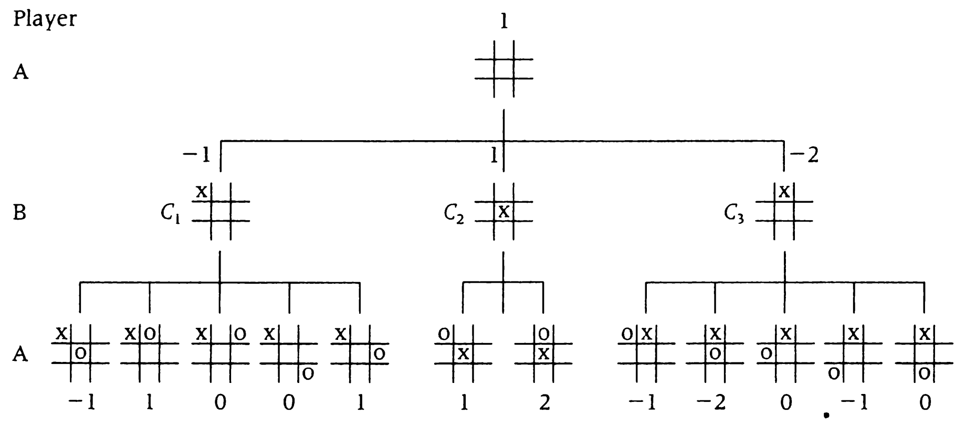


FIGURE 23.15 Value of each node at level 2 as computed using the number of winning lines completable by Xs minus the number of winning lines completable by Os. The values of nodes at levels 1 and 0 are computed using the minimax strategy.

0, and −9 so assigned to terminal nodes, a two-level search leads to a tie game. A tie game for the 3 × 3 board is the best that either player can hope for when both players play perfectly.

There is a heuristic strategy called *alpha-beta pruning* that can result in a significant reduction in the amount of nodes required to visit during an n-level search and still correctly compute the value of a given node. The easiest way to explain alpha-beta pruning is by example. Consider again the two-level search made in the game tree in Figure 23.15. After returning to the root from the middle child C_2, we know that player A can make a move to a node having value 1. Then we move to the third child C_3 of the root and begin visiting the children of C_3 (grandchildren of the root). The first child of C_3 has value −1, so we can immediately cut off our examination of the remaining children of C_3. The reason is simple: The value of C_3 is the minimum value of its children, so the value −1 of the first child of C_3 places an upper bound of −1 on the value of C_3. Because the lower bound on the value of the root is already known to be 1, the value of C_3 cannot possibly affect the value of the root. The cutoff just described is illustrated in Figure 23.16. A cutoff of the search of the grandchildren of an A-node (respectively, B-node) is called *alpha-cutoff* (respectively, *beta-cutoff*).

We formalize the notion of alpha-beta pruning as follows. A lower bound for the value of an A-node is called an *alpha value* of the A-node. Note that during an r-level search, an alpha value of a parent A-node is determined when we return to the A-node from its first child, and the alpha value can be updated, as appropriate, when we return from subsequent children. For example, after returning

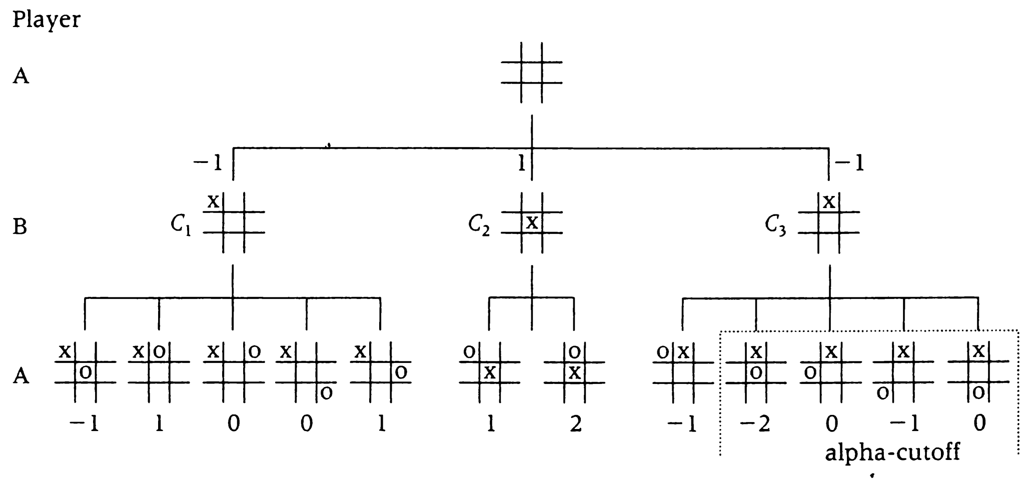


FIGURE 23.16

An alpha-cutoff.

to the root from child C_1 in Figure 23.16, we knew that -1 was an alpha value of the root. However, on returning to the root from the second child C_2, we could update the alpha value of the root to 1.

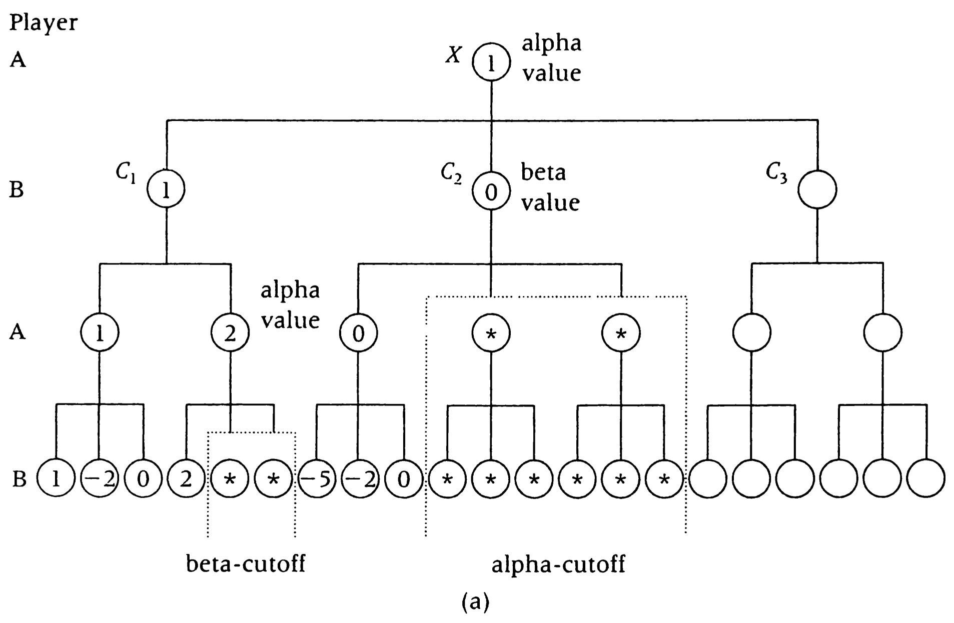
In general, suppose during an r-level search we are examining the children of the ith child of C_i, where the parent of C_i is an A-node X. If we encounter a child of C_i (grandchild of X) whose value is not larger than an alpha value of X, then we can cut off (alpha-cutoff) our search of the remaining children of C_i, because the value of C_i cannot affect the value of X.

An entirely symmetric discussion holds for B-nodes. Specifically, an upper bound for the value of a B-node is called a *beta value* of the B-node. Given any grandparent B-node Y, if during an r-level search we encounter a child (grandchild of Y) of the ith child D_i of Y whose value is not smaller than a beta value of Y, then we can cut off (beta-cutoff) our search of the remaining children of D_i, because the value of D_i cannot affect the value of Y.

Figure 23.17 illustrates a sample game tree and the effect of alpha-beta pruning for a complete search (that is, a three-level search) of the tree. To illustrate the dynamic nature of alpha and beta values, in Figure 23.17a, we show the indicated alpha and beta values of nodes just after returning to the node X from its second child. The value inside a given node is either the actual value of the node or an alpha or beta value as appropriate. Those nodes containing an alpha or a beta value are flagged as such. A value in a node that is shown as * means that the value is irrelevant because the node is never reached due to alpha-beta pruning. To emphasize the stage of the search in Figure 23.17a, no values are shown inside the nodes of the subtree rooted at the third child of X. In Figure 23.17b, values are supplied for the nodes in the latter subtree, where we show the results of the completed search.

When writing pseudocode implementing the minimax procedure, it is convenient to consider the value of a B-node to be the value to player B, not A. In other words, we simply change the signs of the values given to B-nodes in our previous discussion. These changes simplify the pseudocode by turning the minimax procedure into a max procedure. Note that in the max procedure, the value of either an A-node or a B-node is the maximum of the negatives of the values of their children. Thus, the identical max procedure is executed at an A-node or a B-node.

In the new scenario, our cutoff rule takes the same form whether we are examining the children of an A-node or those of a B-node. In either case, suppose *LB* is a lower bound for the value of the node v whose children are being evaluated, and suppose *ParentValue* is a lower bound for the value of the parent node. If we ever determine that $-LB \leq ParentValue$, then we can cut off further examination of the children of v because the value of v cannot affect the value of the parent of v.



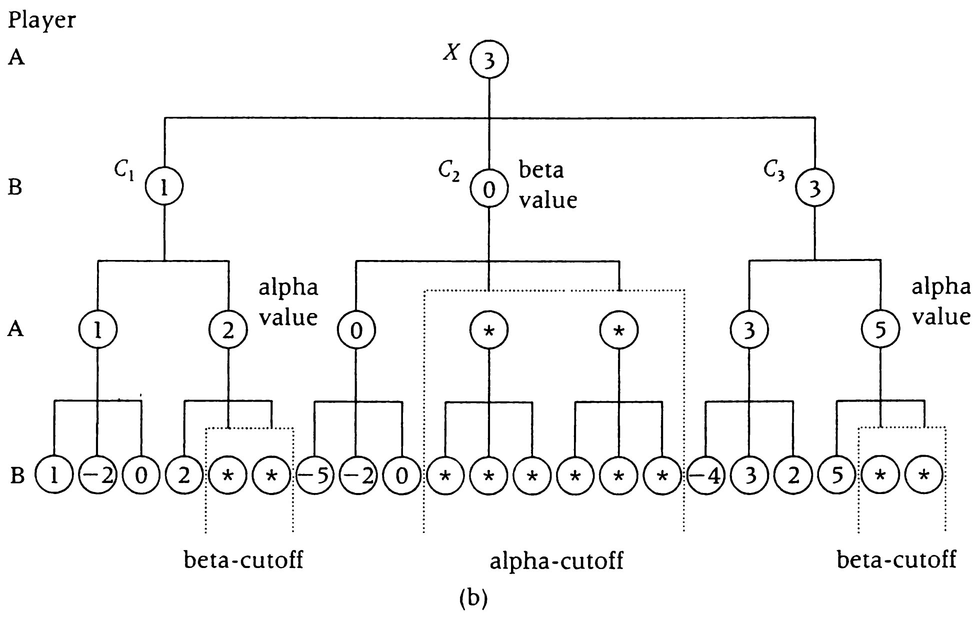


FIGURE 23.17

(a) The results of the search just after returning to X from its second child C_2; b) the results of the completed search.

The following pseudocode for the recursive function *ABNodeValue* returns the value of a node X in the game tree using a *NumLevels*-search, where the parameter *ParentValue* is a lower bound for the value of the parent of X. If X is a terminal node, or if *NumLevels* $= 0$, then we assume that X has been given an appropriately defined value (denoted by *Val*(X)) as described earlier. Given a node X and an integer r, the value of X would be calculated using an r-search by invoking *ABNodeValue* initially with arguments X, r, ∞.

```
function ABNodeValue(X, NumLevels, ParentValue) recursive
Input:   X (a node in the game tree having children $C_1, C_2, \ldots, C_k$),
         NumLevels (number of levels to search)
         ParentValue (lower bound on the value of the parent of X)
Output:  returns the value of X
   if X is a terminal node .or. NumLevels = 0 then
      return(Val(X))
   else
      LB ← −NodeValue($C_1$, NumLevels − 1, ∞) //initial lower bound for value of X
      for i ← 2 to k do
         if LB ≥ ParentValue then                 //cutoff
            return(LB)
         else
            LB ← max(LB, −NodeValue($C_i$, NumLevels − 1, −LB)
         endif
      endfor
   endif
   return(LB)
end ABNodeValue
```

When measuring the efficiency of *ABNodeValue*, the quantity of interest is the number of nodes cut off from the straight minimax r-search of the game tree that does not use the cutoff rule. Of course, not much can be said in general unless some assumptions are made about the regularity of the game tree. Even with strong regularity conditions imposed on the game tree, the analysis is difficult. We merely state one result in this direction. Perl has shown that for game trees in which each parent has the same number of children, and in which the terminal nodes are randomly ordered, *ABNodeValue* permits a search depth greater by a factor of 4/3 than that allowed by the straight minimax procedure in the same amount of time.

Alpha-beta pruning can be enhanced by adding an additional parameter *NodeValueLowBnd* into the algorithm *ABNodeValue*. *NodeValueLowBnd* is maintained as a lower bound on the value on the input parameter X. Additional pruning of the game tree results from the following key fact.

Key Fact

If the value of a grandchild of X is not larger than *NodeValueLowBnd*, then all remaining children of the grandchild can be pruned.

Whereas alpha-beta pruning only uses information from the parent, *NodeValueLowBnd* carries information deep into the tree, and the resulting evaluation of the game tree is called *deep* alpha-beta pruning. We leave the design of the recursive function for deep alpha-beta pruning as an exercise.

23.6 Closing Remarks

Seeking solutions to the 8-puzzle game or finding a shortest-length trip along a freeway system are examples of what has been called *single-agent* problems. In general, an A*-search is better suited for a large-scale problem in which the *entire solution* is sought in a reasonable length of time (and then saved for future reference) than for a real-time problem in which the first step (and each successive step) in the solution must be computed very quickly. For example, it might be acceptable for a computer to take weeks or even months to solve a highly important single-agent problem because its solution would then be known and usable in real time thereafter.

An ordinary A*-search as applied to a single-agent problem usually finds the entire path to a goal before even the first move from the starting position is definitely known. Hence, using an A*-search for a single-agent problem becomes too costly for a large-scale application where the optimal decisions along the way must be made quickly and in advance of the final solution. For example, you might be under a short time constraint to make each move in a game like the 8-puzzle game, rather than simply wanting to determine the *entire solution* in a larger but more reasonable length of time.

While most single-agent problems are not subject to intermediate real-time constraints, two-player games usually are. Chess, for example, usually restricts the amount of time a player has to make the next move. Moreover, the game tree for chess is so enormous that generating complete solutions is out of the question. To make real-time decisions, the alpha-beta heuristic is based on attempting to evaluate moves in a limited *search horizon*—that is, looking ahead a fixed number of moves.

For a single-agent problem, a heuristic search method called *real-time A*-search* combines the A*-search strategy with a limited look-ahead search horizon. A real-time A*-search uses an analog of minimax alpha-beta pruning called minimin *alpha pruning*. Alpha pruning drastically improves the efficiency of A*-search without affecting the decisions made. Like an A*-search, a real-time A*-search can find the entire solution to a fairly large-scale problem in a reasonable amount of time. However, a real-time A*-search has the advantage of generating the optimal moves along the way quickly and before the entire solution is known. Refer to the references for further information on the real-time A*-search.

Suppose that a perfect-information game involving alternate moves by two players A and B must end in a finite number of moves and a win for one of the two players. Then one of the two players must have a *winning strategy*—that is, a strategy that guarantees a win regardless of the moves made by the other player. The reason is simple: If neither player had a winning strategy, there would be a sequence of alternate moves made by A and B that never ends in a loss for either player. Because that sequence would not terminate, we would obtain a contradiction of the finiteness assumption of the game.

A two-person *positional* game is determined by a collection of sets A_i, $i = 1, \ldots, m$. The players alternately choose an element (which they keep) from $\bigcup_{i=1}^{n} A_i$. The first player to choose *all* the elements from one of the sets wins. Tic-tac-toe is an example of a positional game, where the sets A_i are the winning lines in the board. For positional games that cannot end in a tie, such as the $3 \times 3 \times 3$ game of tic-tac-toe, the first player always has a winning strategy. Indeed, we have just seen that one of the two players must have a winning strategy, so suppose it is the second player. Then the first player makes a random opening move, and thereafter assumes the role of the second player (basically ignoring the opening move). More precisely, when the first player moves, he chooses the move dictated by the second player's winning strategy, or moves randomly if he has previously made this move. Since having made an extra move in a positional game cannot possibly hurt, the first player is thus led to a win! This contradicts the assumption that the second player has a winning strategy and shows that the first player has a winning strategy. The same argument shows that if there is a winning strategy for a positional game, then it must belong to the first player.

For positional games that cannot end in a tie, the fact that the first player has a winning strategy does not mean that there is an efficient algorithm to generate the strategy. Also, for positional games that *can* end in a tie (given, perhaps, imperfect play)., there still might exist a winning strategy for the first player. For example, tie positions exist for the $4 \times 4 \times 4$ game of tic-tac-toe (winning sets being four in a row). However, it was conjectured for a long time that the first player has a winning strategy in $4 \times 4 \times 4$ tic-tac-toe. This conjecture was finally established by Patashnik using clever bounding arguments that allowed a pruning of the enormous game tree for $4 \times 4 \times 4$ tic-tac-toe, reducing it to a size that was amendable to computer search.

References and Suggestions for Further Reading

Kanal, L., and V. Kumar, eds. *Search in Artificial Intelligence.* New York: Springer-Verlag, 1988. Contains numerous articles on search in artificial intelligence, including a discussion on the optimality of the A*-search.

Korf, R. E. "Real-Time Heuristic Search," *Artificial Intelligence* 42 (1990): 189–211. A paper devoted to the real-time A*-search.

Nilsson, N. J. *Principles of Artificial Intelligence.* Palo Alto, CA: Tioga, 1980. A detailed account of the A*-search, which was originally developed by Hart, Nilsson, and Raphael.

Pearl, J. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Reading, MA: Addison-Wesley, 1984. A text devoted to heuristic searching.

Two books on artificial intelligence that contain extensive discussions of search strategies and game playing:

Rich, E., and K. Knight. Artificial Intelligence. 2nd ed. New York: McGraw-Hill, 1991.

Russell, S. J., and P. Norvig. Artificial Intelligence: A Modern Approach. Englewood Cliffs, NJ: Prentice Hall, 1995.

Patashnik, O. "Qubic: 4 × 4 × 4 Tic-Tac-Toe," Mathematics Magazine 53 (1980): 202–223. Survey discussion of n-dimensional tic-tac-toe, as well as the proof that the 4 × 4 × 4 is a first-player win.

Two papers containing detailed analyses of alpha-beta and deep alpha-beta pruning:

Baudet, G. "An Analysis of the Full Alpha-Beta Pruning Algorithm," Proceedings of the 10th Annual ACM Symposium on Theories of Computing, San Diego, CA: Association for Computing Machinery, 1978, pp. 296–313.

Knuth, D. "An Analysis of Alpha-Beta Cutoffs," Artificial Intelligence 6 (1975): 293–323.

Berlekamp, E. R., J. H. Conway, and R. K. Guy. Winning Ways, for Your Mathematical Plays. Vol. I, II. New York: Academic Press, 1982. Covers strategies for a host of games.

EXERCISES

Section 23.2 8-Puzzle Game

23.1 Draw the first three levels of the state-space tree generated by a breadth-first search for the 8-puzzle game with the following initial and goal states:

goal

1	2	3
4	5	6
7	8	

initial state

4	2	7
1		6
3	5	8

23.2 The $(n^2 - 1)$-puzzle is a generalization of the 8-puzzle to the $n \times n$ board. The goal position is where the tiles are in row-major order (with the empty space in the lower-right corner). For $k \in \{1, \dots, n^2\}$, let $L(k)$ denote the number of tiles t, $t < k$, such that the position of t comes after k in the row-major order in the initial arrangement (the empty space is considered as tile n^2). Show that a necessary and sufficient condition that the goal can be reached is that

$$\sum_{k=1}^{n^2} L(k) \equiv i + j \pmod 2, \qquad \textbf{(23.3.4)}$$

where (i, j) is the position of the empty space in the initial arrangement.

Section 23.3 A*-Search

23.3 Show that if h is a monotone heuristic, then $h(v)$ is a lower bound of the cost of the shortest path from v to a (nearest) goal.

23.4 Prove Proposition 23.3.3.

23.5 Design an algorithm for an A*-search using a heuristic that is not necessarily monotone.

23.6 For the 8-puzzle game, consider the following heuristic:

$h(v)$ = the number of tiles not in correct cell in the state v.

Show that $h(v)$ satisfies the monotone restriction.

23.7 For the 8-puzzle game, consider the following heuristic:

$h(v)$ = the sum of the Manhattan distances (vertical steps plus horizontal steps) from the tiles to their proper positions in the goal state.

Show that h satisfies the monotone restriction.

23.8 For the 8-puzzle game, let h be the monotone heuristic defined in Exercise 23.6. For the following initial and goal states, draw the states generated by making the first three moves in the game using an A*-search with the priority function $f(v) = g(v) + h(v)$ [$g(v)$ is the number of moves made from the initial state to v]. When enqueuing states, assume that the (possible) moves of the empty tile are ordered as follows: move left, move right, move up, move down. Label each state v with its f-value.

goal

1	2	3
4	5	6
7	8	

initial state

4	2	7
1		6
3	5	8

23.9 Repeat Exercise 23.8 for the heuristic h defined in Exercise 23.7.

23.10 Write a program for the n-puzzle game using the Manhattan distance heuristic. Test your program for $n = 8$ and $n = 15$.

23.11 Can you find better heuristics (not necessarily lower bounds) for the n-puzzle game than the Manhattan distance heuristic? Test your heuristic empirically for $n = 8$ and $n = 15$.

23.12 Prove Theorem 23.3.1

Section 23.4 Least-Cost Branch-and-Bound

23.13 Show that the heuristic $h(v)$ given in Section 23.4 for the coin-changing problem is a lower bound for the minimum number of additional coins required to make correct change from the given problem state v.

23.14 Write a program implementing a least-cost branch-and-bound solution to the coin-changing problem.

23.15 Design a heuristic and a least-cost branch-and-bound algorithm for the variation of the coin-changing problem in which we have a limited number of coins of each denomination. Assume the number of coins of each denomination is input along with the denominations.

23.16 Draw the portion of the variable-tuple state-space tree generated by least-cost branch-and-bound for the instance of the 0/1 knapsack problem given in Figure 10.13 in Chapter 10, using the heuristic given in Section 23.4. Label each node with the value of $c(v)$ and the current value of UB.

23.17 Repeat Exercise 23.16 for the fixed-tuple state-space tree.

23.18 Write a program implementing a least-cost branch-and-bound solution to the 0/1 knapsack problem.

23.19 Given the complete digraph $\hat{K}_n$ with vertices $0, 1, \dots, n - 1$ and a nonnegative cost matrix $C = (c_{ij})$ for its edges (we set $c_{ij} = \infty$ if $i = j$ or if the edge ij does not exist), a traveling salesman tour starting at vertex 0 corresponds to a sequence of vertices $0, i_1, i_2, \dots, i_{n-1}, 0$, where $i_1, i_2, \dots, i_{n-1}$ is a permutation of $1, \dots, n - 1$. Consider a state-space tree T for the traveling salesman problem (finding a minimum-cost tour) where a node at level k in T corresponds to a simple path containing $k + 1$ vertices, starting with vertex 0. Thus, T has depth n, and leaf nodes correspond to a sequence of choices $i_1, i_2, \dots, i_{n-1}$, determining the tour $0, i_1, i_2, \dots, i_{n-1}, 0$.

We now describe a cost function $c(v)$ for a least-cost branch-and-bound algorithm for the traveling salesman problem. The definition of $c(v)$ is based on the notion of a reduced-cost matrix. A row (or column) of a nonnegative cost matrix is said to be *reduced* if it contains at least one zero. A nonnegative cost matrix is *reduced* if each row and column of the matrix is reduced (except for rows and columns whose elements are all equal to ∞). Given the cost matrix C, an associated reduced-cost matrix C_r is constructed as follows. First, reduce each row by subtracting the minimum entry in the row from each element in the row. In the resulting matrix, repeat this process for each column. We define $c(r)$ to be the total amount subtracted. The following example illustrates C_r for a sample C.

$$C = \begin{pmatrix} \infty & 23 & 9 & 32 & 12 \\ 21 & \infty & 2 & 16 & 4 \\ 4 & 8 & \infty & 20 & 6 \\ 15 & 10 & 4 & \infty & 2 \\ 9 & 5 & 8 & 10 & \infty \end{pmatrix} \qquad C_r = \begin{pmatrix} \infty & 14 & 0 & 18 & 3 \\ 19 & \infty & 0 & 9 & 2 \\ 0 & 4 & \infty & 11 & 2 \\ 13 & 8 & 2 & \infty & 0 \\ 4 & 0 & 3 & 0 & \infty \end{pmatrix} \qquad c(r) = 27$$

More generally, we define (inductively on the levels of T) a reduced-cost matrix for each nonleaf node by suitably reducing the cost matrix C_u associated with the parent node u of v. Suppose u corresponds to a path ending at vertex i, and v corresponds to adding the edge ij to this path. We then change all the entries in row i and column j of C_u to ∞, as well as the entry in the j^{th} row and first column. We then perform the same subtracting operation on the resulting matrix as we did when computing C_r. Let s_v denote the total amount subtracted, and define $c(v) = c(u) + C_u(i,j) + s_v$.

For leaf nodes v, $c(v)$ is defined as the cost of the tour determined by v.

a. Show that $c(r)$ is a lower bound for the minimum cost of a tour.

b. More generally, show that $c(v) \le \phi^*(v)$ = the minimum cost over all tours determined by the leaf nodes of the subtree of T rooted at v.

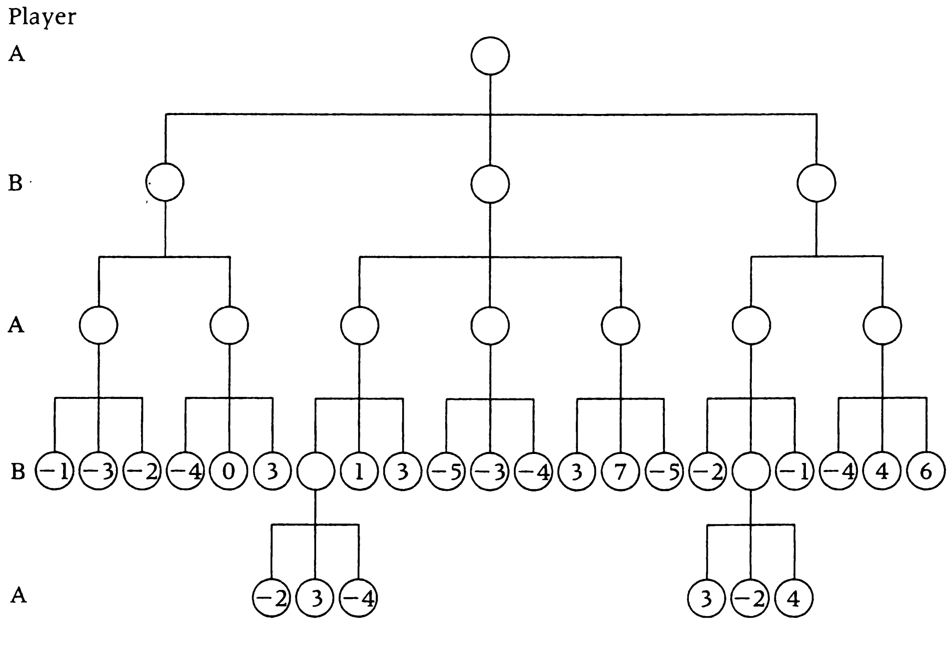
c. Part (b) shows that $c(v)$ is suitable for *LeastCostBranchAndBound*. Design and give pseudocode for *LeastCost BranchAndBound* implementing $c(v)$.

23.20 Draw the portion of the state-space tree T generated by the least-cost branch-and-bound discussed in the Exercise 23.19 for the cost matrix illustrated in that exercise. Label each node v with its cost value $c(v)$. Also, write out the reduced matrix associated with each node generated.

23.21 Discuss other state-space trees and associated cost functions $c(v)$ for the traveling salesman problem.

Section 23.5 Game Trees

23.22 Consider the two-person zero-sum game shown in the figure below. The values in the leaf nodes are values to player A. Use the minimax strategy (postorder traversal) to determine the value of the game to player A. Show clearly where alpha-cutoff and beta-cutoff occur, as well as (final) actual values, alpha values, and beta values of all nodes reached in the traversal.



23.23 Rewrite the recursive function *NodeValue* as a recursive procedure that has the same input parameters, *Y*, *NumLevels*, *ParentValue*, but now returns in output parameters the value V of Y and the child C_i, whose value is $-V$.

23.24 Find a sequence of admissible moves for the two-level heuristic search illustrated in Figure 23.16 that leads to a loss for player A in 3×3 tic-tac-toe.

23.25 Show that by assigning the values 9, 0, and -9 to terminal nodes that are wins, ties, or losses, respectively, for player A, the two-level search illustrated in Figure 23.16 never leads to a loss for player A.

23.26 Because there are no tie positions in the $3 \times 3 \times 3$ tic-tac-toe game, the first player has a winning strategy. Find a winning strategy for the first player.

23.27 Design a recursive function *DABNodeValue(X*, *NumLevels*, *ParentValue*, *NodeValueLowBnd*) for deep alpha-beta pruning. The initial invocation of *DABNodeValue* should have *ParentValue* $= \infty$ and *NodeValueLowBnd* $= -\infty$.

23.28 Redo Exercise 23.22 for deep alpha-beta pruning. Indicate any pruned nodes that were not pruned by alpha-beta pruning.