8

Dynamic Programming

Dynamic programming is a design strategy that involves constructing a solution S to a given problem by building it up dynamically from solutions $S_1,S_2,...,S_m$ to smaller (or simpler) instances of the problem. The solution S_i to any given smaller problem instance is itself built up from the solutions to even smaller (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 $S_1,S_2,...,S_m$, some additional information is usually required. We let *Combine* denote the function that combines $S_1,S_2,...,S_m$, using the additional information to obtain S, so that

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

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 utilize 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. Also, unlike many instances of divide-and-conquer, dynamic programming algorithms typically never consider a given problem instance than once.

Dynamic programming algorithms for optimization problems also can avoid generating suboptimal problem instances when the *Principle of Optimality* holds, thereby leading to increased efficiency. In this chapter we consider a number of well-known problems that are amenable to the method of dynamic programming.

8.1 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 S to the given optimization problem; that is, solutions S 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 $S_1,S_2,...,S_m$ such that $S = Combine(S_1,S_2,...,S_m)$. For many problems it is computationally infeasible to examine all such solutions, because often there are exponentially many possibilities. Fortunately we can drastically reduce the number of problem instances that we need to consider if the Principle of Optimality holds.

Definition 8.1.1

Given an optimization problem and an associated function *Combine*, the *Principle of Optimality* holds if the following is always true: If $S = Combine(S_1,S_2,...,S_m)$ and S is an

optimal solution to the problem instance, then $S_1,S_2,...,S_m$, 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 (i) the bottom-up resolution of the recurrence, thereby eliminating redundant recalculations, and (ii) 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 $M_0,...,M_{n-1}$ that minimizes the total number of (scalar) multiplications over all possible parenthesizations. If $(M_0 ...M_k)(M_{k+1}...M_{n-1})$ is the "first cut" set of parentheses (and the last product performed), then the matrix products $M_0...M_k$ and $M_{k+1}...M_{n-1}$ 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 $K_0 < ... < K_{n-1}$ 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 8.5). If K_i is the key in the root, then the left subtree L of the root contains $K_0,...,K_{i-1}$ and the right subtree R of the root contains $K_{i+1},...,K_{n-1}$. Given a binary search tree T for keys $K_0,...,K_{n-1}$, 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 $K_0,...,K_{i-1}$ and R (solution S_2) is a binary search tree for keys $K_{i+1},...,K_{n-1}$. 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.

8.2 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 (scalar) 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 (chain) of matrices M_0, $M_1,...,M_{n-1}$, consider the product $M_0M_1...M_{n-1}$, where the matrix M_i has dimension $d_i \times d_{i+1}$, $i = 0,...,n$, for a suitable sequence of positive integers $d_0,d_1,...,d_n$. Since matrix product is an associative operation, there are a number of ways to evaluate the chained product 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, namely, $(AB)X$ 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 (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 8.2.1

Given the sequence of matrices $M_0,M_1,...,M_{n-1}$, P is a *fully-parenthesized* matrix product of $M_0,M_1,...,M_{n-1}$, which for convenience we simply call a *parenthesization of* $M_0M_1...M_{n-1}$, 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 $M_0M_1...M_k$ and $M_{k+1}M_{k+2}...M_{n-1}$, 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.

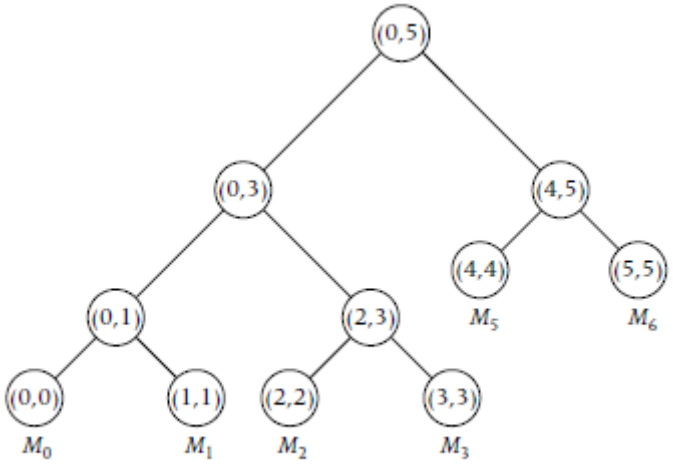
The table in Figure 8.1 shows all the different parenthesizations of the matrices M_0,M_1,M_2, and M_3 having dimensions 20×10, 10×50, 50×5, and 5×30, respectively, with the optimal parenthesizations highlighted.

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

Number of multiplications performed for each full parenthesization are 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.

Figure 8.1

There is one-to-one correspondence between parenthesizations of $M_0M_1...M_{n-1}$ and 2-trees having n leaf nodes. Given a parenthesization P of $M_0M_1...M_{n-1}$, 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 8.2).



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 $M_i,M_{i+1},...,M_j$

Figure 8.2

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

$$p_n = \frac{1}{n}\binom{2n-2}{n-1} \geq \frac{4^{n-1}}{2n^2-n} \in \Omega\left(\frac{4^n}{n^2}\right). \tag{8.2.1}$$

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, consider any optimal parenthesization P for $M_0M_1...M_{n-1}$. 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 $M_iM_{i+1}...M_j$. By the Principle of Optimality, we have the following recurrence for the numbers m_{ij} based on making an optimal choice for the first cut index

$$\begin{aligned} m_{ij} &= \min_k \{m_{ik} + m_{k+1,j} + d_i d_{k+1} d_{j+1} : \ 0 \leq i \leq k < j \leq n-1\} \\ &\textbf{init.cond. } m_{ii} = 0, \quad i = 0,\ldots,n-1. \end{aligned} \tag{8.2.2}$$

The value $m_{0,n-1}$ corresponds to the minimum number of multiplications performed when computing $M_0M_1...M_{n-1}$. A divide-and-conquer algorithm *ParenthesizeRec* can be based directly on a top-down implementation of the recurrence relation (8.2.2). 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 (8.2.2) (and 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 $m_{ii} = 0$, $i = 0,...,n-1$. The qth diagonal consists of the elements $m_{i,i+q}$, $q = 0,...,n-1$. In Figure 8.3 we illustrate the computation of the m_{ij} for the example given in Figure 8.1. When computing m_{ij}, we also generate a table c_{ij} of indices k where the minimum in (8.2.2) occurs, that is c_{ij} is where the first cut in $M_iM_{i+1}...M_j$ 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.

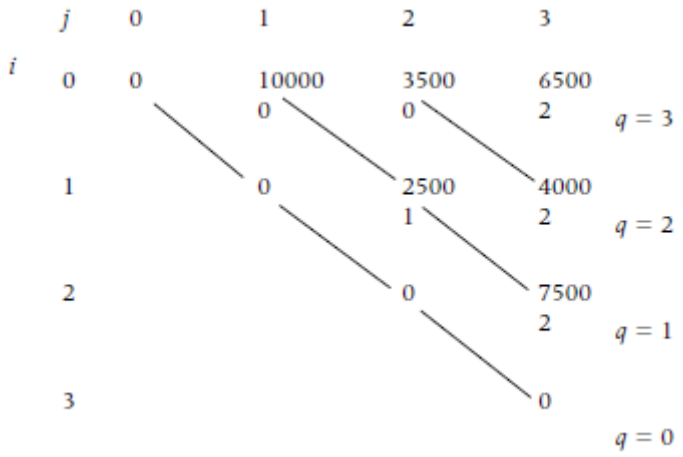


Table showing values m_{ij}, $0 \le i \le j \le 3$, computed diagonal by diagonal from $q = 0$ to $q = 3$ using the botton-up resolution of (8.2.2) for matrices M_0, M_1, M_2, and M_3 having dimensions 20×10, 10×50, 50×5, and 5×30, respectively. The values of c_{ij} are shown underneath each m_{ij}, $0 \le i \le j \le 3$.

Figure 8.3

The following procedure *OptimalParenthesization* accepts as input the *dimension sequence* $d[0{:}n]$, where matrix M_i has dimension $d_i \times d_{i+1}$, $i = 0,...,n-1$. 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 (8.2.2). *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 $M_i...M_j$.

```
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 M0, M1,...,Mn-1)
Output:   m[0:n – 1,0:n – 1] (m[i,j] = number of multiplications performed in an optimal
                              parenthesization for computing Mi...Mj, 0 ≤ i ≤ j ≤ n – 1)
          FirstCut[0:n – 1,0:n – 1]  (index of first cut in optimal parenthesization of
                                      Mi...Mj, 0 ≤ i ≤ j ≤ 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 mij according to (8.2.2)
            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 $M_0...M_{n-1}$ using an optimal parenthesization. We assume that the matrices $M_0,...,M_{n-1}$ and the matrix *FirstCut*[0:n – 1,0:n – 1] are global variables to the procedure *ChainMatrixProd*. The chained matrix product $M_0...M_{n-1}$ 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 Mi,...,Mj)
          M0,...,Mn-1 (global matrices)
          FirstCut[0:n – 1,0:n – 1] (global matrix computed by
                                     OptimalParenthesization)
Output:   Mi...Mj (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
```

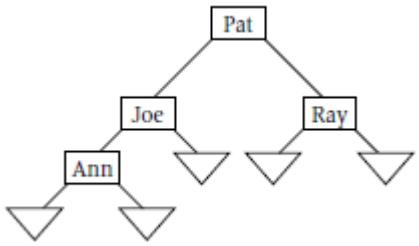
For the example given in Figure 8.1, invoking *ChainMatrixProd* with M_0, M_1, M_2, M_3 computes the chained matrix product $M_0M_1M_2M_3$ according to the parenthesization $((M_0(M_1M_2))M_3)$.

8.3 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 an occurrence (if any) of a key X. First, X is compared to the key K associated with the root. If X is found there, then we are done. Otherwise, if X is less than K, then we search the left subtree, else we search the right subtree.

Consider, for example, the binary search tree given in Figure 8.4 involving the four keys 'Ann', 'Joe', 'Pat', '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. Then, the average number of comparisons made by *SearchBinSrchTree* for the tree T of Figure 8.4 is given by

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



Search tree with leaf nodes drawn representing unsuccessful searches

Figure 8.4

Now consider a general binary search tree T whose internal nodes correspond to a fixed set of n keys $K_0, K_1,...,K_{n-1}$ with associated probabilities $\mathbf{p} = (p_0, p_1,..., p_{n-1})$, 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 $\mathbf{q} = (q_0, q_1,...,q_n)$. (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, $i = 0,...,n-1$. Similarly, let e_i denote the depth of the leaf node corresponding to the interval I_i, $i = 0, 1,...,n$. 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

$$A(T,n,\mathbf{p},\mathbf{q}) = \sum_{i=0}^{n-1} p_i (d_i + 1) + \sum_{i=0}^{n} q_i e_i . \tag{8.3.1}$$

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 $p_0,...,p_{n-1}$ and $q_0,...,q_n$ 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 6. That is, we solve the problem

$$\underset{T}{\text{minimize}}\ A(T,n,\mathbf{p},\mathbf{q}) \tag{8.3.2}$$

over all binary search trees T of size n, where $p_0,...,p_{n-1}$ and $q_0,...,q_n$ 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:

$$\sigma(\mathbf{p},\mathbf{q}) = \sum_{i=0}^{n-1} p_i + \sum_{i=0}^{n} q_i . \tag{8.3.3}$$

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

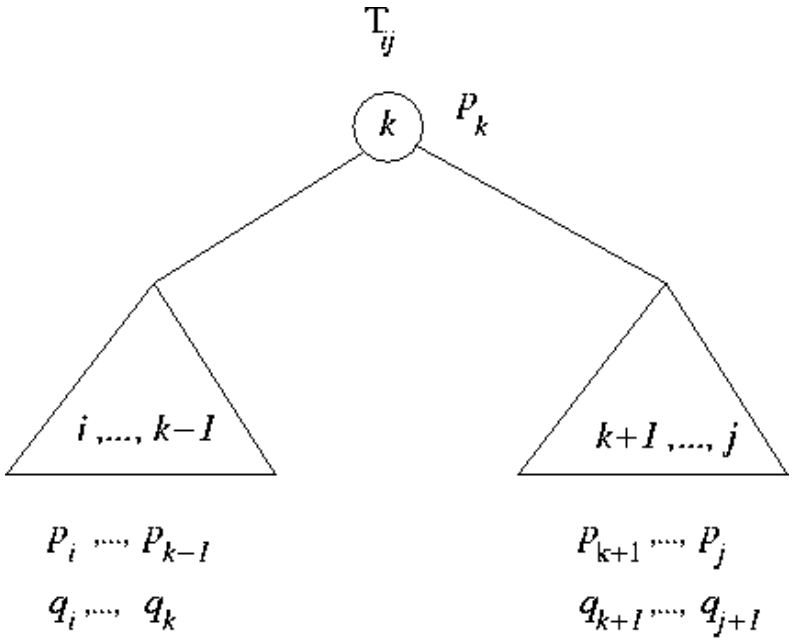
Let K_i denote the key associated with the root of T, and let L and R denote the left and right subtrees (of the root) of T, respectively. As we remarked earlier, L is a binary search tree for the keys $K_0,...,K_{i-1}$, and R is a binary search tree for the keys $K_{i+1},...,K_{n-1}$. For convenience, let $A(T) = A(T,n,\mathbf{p},\mathbf{q})$, $A(L) = A(L,i,p_0,...,p_{i-1},q_0,...,q_i)$, and $A(R) = A(R,n-i-1,p_{i+1},...,p_{n-1},q_{i+1},...,q_n)$. Clearly, each node of T 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 (8.3.1) that

$$A(T) = A(L) + A(R) + \sigma(\mathbf{p},\mathbf{q}). \tag{8.3.4}$$

We now employ (8.3.4) 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 $K_0,...,K_{i-1}$ 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 (8.3.4) 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.

Since 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 (8.3.4), is the basis of the following dynamic programming algorithm for constructing an optimal binary search tree.
For $i,j \in \{0,...,n-1\}$, we let T_{ij} denote an *optimal* search tree involving the consecutive keys $K_i, K_{i+1},...,K_j$, 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 8.5). Also, note that $T = T_{0,n-1}$ is an optimal search tree involving all n keys. For convenience, we define

$$\sigma(i, j) = \sum_{k=i}^{j} p_k + \sum_{k=i}^{j+1} q_k.$$



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

Figure 8.5

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

$$A(T_{ij}) = A(T_{ij}, j-i+1, p_i, p_{i+1}, \ldots, p_j, q_i, q_{i+1}, \ldots, q_{j+1}). \tag{8.3.5}$$

Since the keys are sorted in nondecreasing order, it follows from the Principle of Optimality and (8.3.4) that

$$A(T_{ij}) = \min_k \{A(T_{i,k-1}) + A(T_{k+1,j})\} + \sigma(i, j), \tag{8.3.6}$$

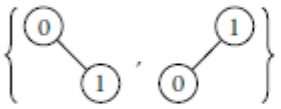
where the minimum is taken over all $k \in \{i, i+1, \ldots, j\}$.

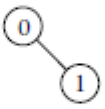
Recurrence relation (8.3.6) yields an algorithm for computing an optimal search tree T. The algorithm begins by generating all single-node binary search trees, which are trivially optimal. Namely, $T_{00}, T_{11}, \ldots, T_{n-1,n-1}$. Using (8.3.6), the algorithm can then generate optimal search trees $T_{01}, T_{12}, \ldots, T_{n-2,n-1}$. In general, at the kth stage in the algorithm, the recurrence relation (8.3.6) is applied to construct the optimal search trees $T_{0,k-1}, T_{1,k}, \ldots, T_{n-k,n-1}$, using the previously generated optimal search trees as building blocks. Figure 8.6 illustrates the algorithm for a sample instance involving $n = 4$ keys. Note that there are two possible choices for T_{02} in Figure 8.6, each having a minimum cost of 1.1. The tree with the smaller root key was selected.

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

$T_{00} =$ (0) $T_{11} =$ (1) $T_{22} =$ (2) $T_{33} =$ (3)

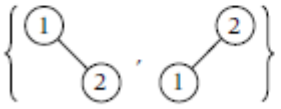
$A(T_{00}) = .25$ $A(T_{11}) = .15$ $A(T_{22}) = .25$ $A(T_{33}) = .45$

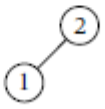


$\Rightarrow T_{01} =$ 

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

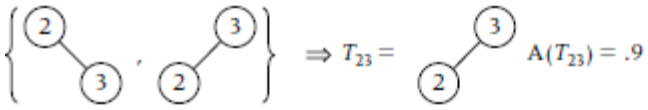
$\min\{0 + .15 = .15, .25 + 0 = .25\} + .35 = .5$



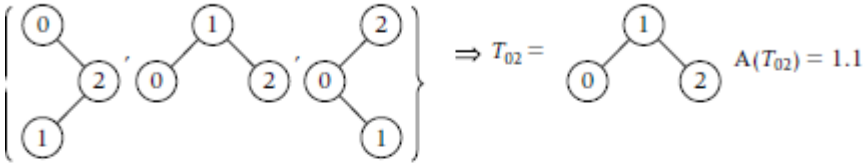
$\Rightarrow T_{12} =$ 

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

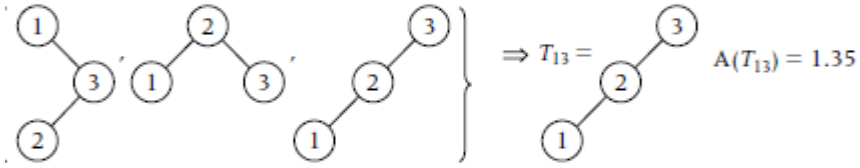
$\min\{0 + .25 = .25, .15 + 0 = .15\} + .4 = .55$ Continued



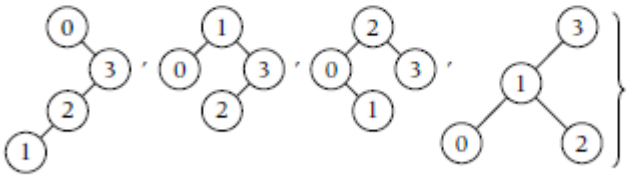
nin {0 + .45 = .45, .25 + 0 = .25} + .65 = .9



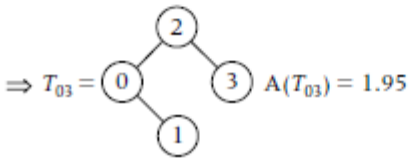
nin {0 + .55 = .55, .25 + .25 = .5, .5 + 0 = .5} + .6 = 1.1



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



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



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

Figure 8.6

We now give pseudocode for the algorithm *OptimalSearchTree*, which 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 σ(p_i,...,p_j,q_i,...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 8.7, we illustrate the action of *OptimalSearchTree* for the optimal binary search tree described above. For convenience, we will 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 up 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 $Sigma[i,j] = p_i + ... + p_j + q_i + ... + q_{j+1} = Sigma[i,j – 1] + p_j + q_{j+1}$:

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

A[*i*,*j*]

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

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

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

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

Root[*i*,*j*]

i \ *j*	0	1	2	3
0	0	*	*	*
1		1	*	*
2			2	*
3				4

i \ *j*	0	1	2	3
0	1	1	*	*
1		2	3	*
2			3	4
3				4

i \ *j*	0	1	2	3
0	1	1	2	*
1		2	3	4
2			3	4
3				4

i \ *j*	0	1	2	3
0	1	1	2	3
1		2	3	4
2			3	4
3				4

Sigma[*i*,*j*]

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

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

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

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

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*

Figure 8.7

Since *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 all 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

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

8.4 Longest Common Subsequence and Edit Distance

In this section we consider the problem of determining how close two given character strings are to one another. For example, in a spell checking situation, one of the strings might be a string contained in a text being created on a word processor, and another string might be a pattern string from a stored dictionary. If there is no exact match between the text string and any pattern string, then a number of pattern strings might be presented to the operator that are fairly close, in some sense, to the text string. As another example, we might be comparing two DNA strings to measure how close they match. There are a number of ways to define closeness of two strings. The *edit distance* is commonly used by search engines to find approximate matchings for a given text string entered by the user in a query in situations where an exact match is not 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, namely, the longest common subsequence (LCS) contained in a text string and a particular pattern string. Both computing LCS and computing the edit distance are optimization problems satisfying the Principle of Optimality, and both can be solved using dynamic programming. However, the solution to the LCS problem is easier to understand since it has a simpler recurrence relation, which we now describe.

8.4.1 Longest Common Subsequence

Suppose $T = T_0T_1...T_{n-1}$ is a text string that we wish to compare to a pattern string $P = P_0 P_1...P_{m-1}$, 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 $T_{i_1}T_{i_2}...T_{i_k}$, where $0 \leq i_1 < i_2 < \ldots < i_k \leq n-1$. 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 longest common subsequence of T and P. For simplicity of notation, we will 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 $LCS[i,j]$

$$\begin{aligned} LCS[i,j] &= LCS[i-1,j-1] + 1 && \text{if} \quad T[i-1] = P[j-1], \\ &= \max\{LCS[i,j-1]\ ,\ LCS[i-1,j]\} && \text{otherwise.} \end{aligned} \qquad (8.4.1)$$

To verify (8.4.1), 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

$$LCS[i,j] = \max\{LCS[i,j-1], LCS[i-1,j]\} \quad \text{if} \quad T[i-1] \neq P[i-1]. \qquad (8.4.2)$$

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 this common value C to a given subsequence we would increase the length of the subsequence by one. 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 $P[j-1]$. But 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:

$$LCS[i,j] = LCS[i-1,j-1] + 1 \quad \text{if} \quad T[i-1] = P[i-1]. \qquad (8.4.3)$$

which, together with (8.4.2), completes the verification of (8.4.1).

The following algorithm is the straightforward row-by-row computation of the array $LCS[0{:}n, 0{:}m]$ based on the recurrence (8.4.1).

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 $i \leftarrow 0$ to n **do** //initialize for boundary conditions
        $LCS[i,0] \leftarrow 0$
    endfor
    for $j \leftarrow 0$ 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 (8.4.1)
            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 $\Theta(nm)$, which is a rather dramatic improvement over the exponential complexity $O(2^n m)$ 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 $P[0{:}m-1]$.

In Figure 8.8 we show 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, it was not of much value to know the minimum number of multiplications required without determining the actual paranthesization that did the job. This is why we needed to compute the array $FirstCut[0{:}n-1,0{:}n-1]$ in order to be able to construct the optimal paranthesization. Similarly, in the optimal binary search tree problem, it was not of much value to know the average search complexity of the optimal search tree without actually determining the optimal search tree itself. That is why we kept track of the key $Root[i,j]$ in the root of the optimal binary search tree containing the keys $K_i < \ldots < K_j$. However, in the LCS problem, knowing the actual common subsequence is not as important as knowing its length. For example, in a situation like spell checking, the subsequence is not as important as its length, since typically one would be given a list of pattern strings for correcting a misspelled word in the text that share subsequences exceeding a threshold length (dependent of the length of the strings), as opposed to exhibiting common subsequences. Nevertheless, it is interesting that a longest common sequence 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 to generate a longest common subsequence is to start at the bottom right-hand corner position (n,m) of the array *LCS*, and work your way backwards through the array to build the subsequence in reverse order. The moves are dictated by looking at how you get the value assigned to a given position when you used (8.4.1) to build the array *LCS*. More precisely, if you 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 you move to position $(i-1,j-1)$ in *LCS*. On the other hand, if $T[i-1] \neq P[j-1]$, then you move to position $(i-1,j)$ or $(i,j-1)$ depending on

whether $LCS[i-1,j]$ is greater than $LCS[i,j-1]$ or not. In the case where $LCS[i-1,j]$ is equal to $LCS[i,j-1]$, then either move can be made. In the latter case, the two different choices might not only generate different longest common subsequences, but may also yield different longest common strings corresponding to these subsequences. For example, in Figure 8.8a we show the path generated using the go-left rule, where we always move to position $(i-1,j)$ when $T[i-1] \neq P[j-1]$, whereas Figure 8.8b shows the path resulting by 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*, a longest common subsequence is obtained when you reach a position where $i = 0$ or $j = 0$.

			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

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 on ties rule

Figure 8.8a

			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

The path in *LCS* generating the longest common string *useune* the move-up on ties rule

Figure 8.8b

8.4.2 Edit Distance and Approximate String Matching

In practice, there are often misspellings in the 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 and solving this problem using dynamic programming. We will first consider the problem of determining whether a pattern string $P = p_1 \ldots p_p$ is an *k*-approximation of a text string $T = t_1 \ldots t_t$. Later, we will 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

(i) changing a character of *T* (substitution)
(ii) adding a character to *T* (insertion)
(iii) removing a character of *T* (deletion)

For example suppose *P* is the string "algorithm"

(i) elgortihm → algorithm (*substitution* of *e* with *a*)
(ii) algorthm → algorithm *(insertion* of letter *i*)
(iii) lagorithm → algorithm (*deletion* of letter *l*)

In the above example each string *T* differs from *P* in at most one character. Unfortunately, in practice more serious mistakes are made where the difference may involve 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 string "algorithm" and "logarithm" have edit distance 3

logarithm → alogarithm → algarithm → algorithm.

Let *D*[*i*,*j*] denote the editing distance between the substring *P*[1:*i*] consisting of the first *i* characters of the pattern string P and *T*[1:*j*] 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 $T[1{:}j]$ into $P[1{:}i]$. The number of such operations is the edit distance. 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:

$$D[i, j] = \begin{cases} D[i-1, j-1], & p[i] = t[j], \\ \min\{D[i-1, j-1]+1, D[i-1, j]+1, D[i, j-1]+1\}, & \textit{otherwise}. \end{cases} \qquad (8.4.4)$$

$$\textbf{init. cond. } D[0,\mathrm{i}] = D[0,\mathrm{i}] = i.$$

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, and we leave it to the exercises. We also leave it to the exercise to design an algorithm for finding the first occurrence or all occurrence of a substring of the text string T that is a k-approximation of the pattern string P.

8.5 Floyd's Algorithm

Dijkstra's shortest-path algorithm find paths from a single source vertex r to all other vertices of a weighted digraph $D = (V,E)$ and has complexity $\Theta(n^2)$. By repeated calls to either of Dijkstra's algorithm we obtain an algorithm for a shortest path between every pair of vertices u and v, having $\Theta(n^3)$ complexity. We now use dynamic programming to obtain a $\Theta(n^3)$ algorithm, Floyd's algorithm, (also known as the Floyd-Warshall algorithm), for finding a shortest path between every pair of vertices of a weighted digraph D. Unlike the all-pairs shortest paths algorithm based on repeated applications of Dijkstra's algorithm, Floyd's algorithm works even if some of the edges have negative weights, provided that there is no negative cycle (sum of the weights of the edges in the cycle is negative).

Consider a digraph D with vertex set $V = \{0,...,n-1\}$ and edge set E, whose edges have been assigned a weighting w. Let P be a path joining two distinct vertices, $i, j \in V$, and suppose v is an interior vertex of P (that is, any vertex of P different from either i or j). As discussed earlier, the Principle of Optimality holds, so that the subpath P_1 from i to v is a shortest path from i to v, and the subpath P_2 from v to j is a shortest path from v to j.

For k a nonnegative integer $k \leq n-1$, let V_k denote the subset of vertices $\{0,...,k\}$ (by convention, $V_{-1} = \varnothing$). Let $S_k(i,j)$ denote the weight of a shortest path P from i to j, whose interior vertices (if any) all lie in V_k (if no such path P exists, then $S_k(i,j) = \infty$). Since, by

definition, $S_{-1}(i,j)$ is the weight of a shortest path from i to j containing no interior vertices, S_{-1} is equal to the *weight matrix* W defined as follows: $W(i,j) = w(e)$ if D contains an edge e from vertex i to vertex j, 0 if $i = j$, and ∞ otherwise. Note also that $S_{n-1}(i,j)$ is the weight of a shortest path in G from i to j.

If k is not an interior vertex of P, then P is a shortest path from i to j whose interior vertices lie in V_{k-1}, so that $S_k(i,j) = S_{k-1}(i,j)$. On the other hand, if k is an interior vertex of P, then the interior vertices (if any) of the path P_1 from i to k and the interior vertices (if any) of the path P_2 from k to j all lie in V_{k-1}. Since P_1 is a shortest path from i to k and P_2 is a shortest path from k to j, we have $S_k(i,j) = S_{k-1}(i,k) + S_{k-1}(k,j)$. Thus, either $S_k(i,j) = S_{k-1}(i,j)$ or $S_k(i,j) = S_{k-1}(i,k) + S_{k-1}(k,j)$, depending on whether or not P contains vertex k. Since P is a shortest path i to j, it follows that the weight of P is equal to the minimum of these two values, yielding the following recurrence relation for $S_k(i,j)$.

$$\begin{aligned} &S_k(i,j) = \min\{S_{k-1}(i,j), S_{k-1}(i,k) + S_{k-1}(k,j)\} \\ &\textbf{init.cond.} \quad S_{-1}(i,j) = W(i,j) \text{ for all } i, j \in V. \end{aligned} \tag{8.5.1}$$

Floyd's algorithm is based on recurrence relation (8.5.1). The matrix S_k keeps track of the weight of the shortest paths and not the shortest paths themselves. To keep track of the paths, we maintain another matrix $P_k(i,j)$ defined by

$$\begin{aligned} &P_k(i,j) = \begin{cases} P_{k-1}(i,j) & \text{if } S_k(i,j) = S_{k-1}(i,j) \\ k & \text{otherwise} \end{cases}, \\ &\textbf{init.cond.} \; P_{-1}(i,j) = 0 \text{ for all } i, j \in V. \end{aligned} \tag{8.5.2}$$

Floyd's algorithm is illustrated in Figure 8.9 for a sample digraph D and weight function w.

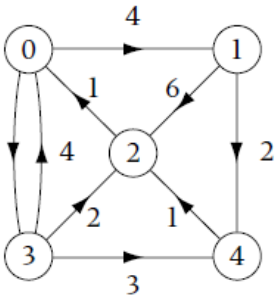
A shortest path from i to j in G can then be reconstructed from P_{n-1} as follows: If $P_{n-1}(i,j) = 0$, then a shortest path from i to j is directly along the edge (i,j); otherwise, if $P_{n-1}(i,j) = k$, then k is an intermediate vertex of a shortest path from i to j and any other intermediate vertices in this shortest path can be obtained by recursively examining $P_{n-1}(i,k)$ and $P_{n-1}(k,j)$. For example, in Figure 8.9, a shortest path S from vertex 2 to vertex 4 can be reconstructed from the final matrix P_4 as follows. Since $P_4(2,4) = 1$, vertex 1 is an internal vertex of S. Now, $P_4(1,4) = -1$, so that there are no intermediate vertices of S between 1 and 4. Since $P_4(2,1) = 0$, vertex 0 is an intermediate vertex of S between 1 and 1. Now, $P_4(2,0) = -1$ and $P_4(0,1) = -1$, so that there are no intermediate vertices between 2 and 0 or between 0 and 1. Thus, a shortest path S from 2 to 4 is given by 2, 0, 1, 4. We now give pseudocode for Floyd's algorithm.

```
procedure Floyd(W[0:n – 1,0:n – 1],P[0:n – 1,0:n – 1],S[0:n – 1,0:n – 1])
Input:    W[0:n – 1,0:n – 1]    (weight matrix for a weighted digraph D)
Output:   P[0:n – 1,0:n – 1] (matrix implementing shortest paths)
                S[0:n – 1,0:n – 1] (distance matrix where S[u,v] is the weight of a
shortest path from u to v in G)
for i ← 0 to n – 1 do          // initialize P and S
          for j ← 0 to n – 1 do
                P[i,j] ← –1
```

```
                S[i,j] ← W[i,j]
            endfor
        endfor
        for k ← 0 to n – 1 do     // update S and P using (8.5.1) and (8.5.2)}
            for i ← 0 to n – 1 do
                for j ← 0 to n – 1 do
                    if S[i,j] > S[i,k] + S[k,j] then
                        P[i,j] ← k
                        S[i,j] ← S[i,k] + S[k,j]
                    endif
                endfor
            endfor
        endfor
end Floyd
```

Remark

An alternate way to maintain the shortest paths is to store in P[*i,j*] the first vertex encountered on the current path from *i* to *j* (see Exercise 8.57).



$$D = \begin{pmatrix} & 4 & & & \\ & & & & 2 \\ 3 & & & & \\ & & 2 & 1 & \\ & & & & \end{pmatrix} \qquad S_0 = W = \begin{pmatrix} 0 & 4 & \infty & 3 & \infty \\ \infty & 0 & 6 & \infty & 2 \\ 1 & \infty & 0 & \infty & \infty \\ 4 & \infty & 2 & 0 & 3 \\ \infty & \infty & 1 & \infty & 0 \end{pmatrix}$$

$$S_1 = \begin{pmatrix} 0 & 4 & \infty & 3 & \infty \\ \infty & 0 & 6 & \infty & 2 \\ 1 & 5 & 0 & 4 & \infty \\ 4 & 8 & 2 & 0 & 3 \\ \infty & \infty & 1 & \infty & 0 \end{pmatrix} \qquad P_1 = \begin{pmatrix} -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & 0 & -1 & 0 & -1 \\ -1 & 0 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{pmatrix}$$

$$S_2 = \begin{pmatrix} 0 & 4 & 10 & 3 & 6 \\ \infty & 0 & 6 & \infty & 2 \\ 1 & 5 & 0 & 4 & 7 \\ 4 & 8 & 2 & 0 & 3 \\ \infty & \infty & 1 & \infty & 0 \end{pmatrix} \qquad P_2 = \begin{pmatrix} -1 & -1 & 1 & -1 & 1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & 0 & -1 & 0 & 1 \\ -1 & 0 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{pmatrix}$$

$$S_3 = \begin{pmatrix} 0 & 4 & 10 & 3 & 6 \\ 7 & 0 & 6 & 10 & 2 \\ 1 & 5 & 0 & 4 & 7 \\ 3 & 7 & 2 & 0 & 3 \\ 2 & 6 & 1 & 5 & 0 \end{pmatrix} \qquad P_3 = \begin{pmatrix} -1 & -1 & 1 & -1 & 1 \\ 2 & -1 & -1 & 2 & -1 \\ -1 & 0 & -1 & 0 & 1 \\ 2 & 2 & -1 & -1 & -1 \\ 2 & 2 & -1 & 2 & -1 \end{pmatrix}$$

$$S_4 = \begin{pmatrix} 0 & 4 & 5 & 3 & 6 \\ 7 & 0 & 6 & 10 & 2 \\ 1 & 5 & 0 & 4 & 7 \\ 3 & 7 & 2 & 0 & 3 \\ 2 & 6 & 1 & 5 & 0 \end{pmatrix} \qquad P_4 = \begin{pmatrix} -1 & -1 & 3 & -1 & 1 \\ 2 & -1 & -1 & 2 & -1 \\ -1 & 0 & -1 & 0 & 1 \\ 2 & 2 & -1 & -1 & -1 \\ 2 & 2 & -1 & 2 & -1 \end{pmatrix}$$

$$S_5 = \begin{pmatrix} 0 & 4 & 5 & 3 & 6 \\ 4 & 0 & 3 & 7 & 2 \\ 1 & 5 & 0 & 4 & 7 \\ 3 & 7 & 2 & 0 & 3 \\ 2 & 6 & 1 & 5 & 0 \end{pmatrix} \qquad P_5 = \begin{pmatrix} -1 & -1 & 3 & -1 & 1 \\ 4 & -1 & 4 & 4 & -1 \\ -1 & 0 & -1 & 0 & 1 \\ 2 & 2 & -1 & -1 & -1 \\ 2 & 2 & -1 & 2 & -1 \end{pmatrix}$$

Stages of Floyd's algorithm for a sample weighted digraph G

Figure 8.6

8.6 The Bellman-Ford Algorithm

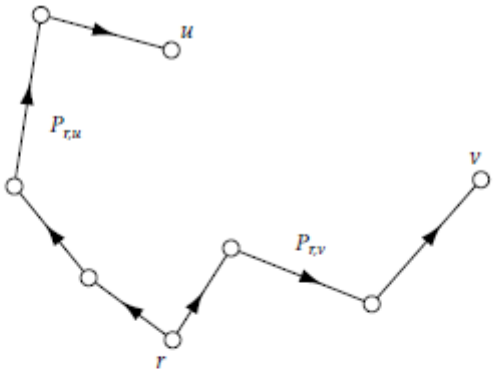
The Bellman-Ford algorithm, as with Dijkstra's algorithm discussed in Chapter 6, finds the best (smallest-weight) path in a directed weighted graph from a given vertex r to all other vertices. While Disjkstra's algorithm has better worst-case performance, the Bellman-Ford algorithm does not require that the weights on the edges are non-negative. The Bellman-Ford algorithm is not directly based on the dynamic programming paradigm, but has a similar flavor. The algorithm is based on making successive scans through all of the edges of the digraph, keeping track of the distance $Dist[v]$ from r to v using the best (smallest cost) path from r to v generated so far**.** After the kth stage of the algorithm, it turns out that $Dist[v]$ is no greater than the value of the smallest-cost path from r to v that has at most k edges. This optimality condition mirrors the Principle of Optimality that characterizes dynamic program solutions.

We initialize the array $Dist[0{:}n-1]$ by setting $Dist[r] = 0$, and $Dist[v] = \infty$ for $v \neq r$. We also keep track of the shortest paths so far generated using a parent array implementation $Parent[0{:}n-1]$ initialized by $Parent[r] = 0$, and $Parent[v] = \infty$ for $v \neq r$. The following key idea is the basis for the Bellman-Ford algorithm.

Key Fact

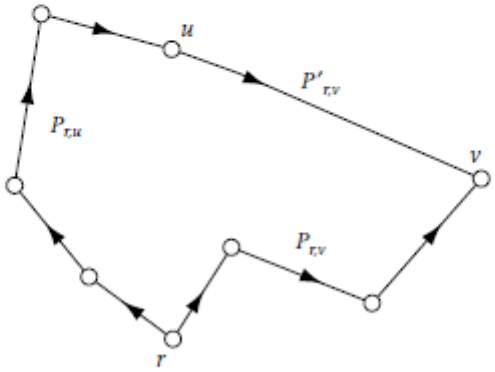
During a given scan through the edges, when we examine the edge uv, $Dist[v]$ is updated by checking whether or not a shorter path from r to v is obtained by replacing the current path by the path consisting of the current path from r to u together with the edge uv. This process of updating $Dist[v]$ is called *relaxing* the edge uv.

The relaxing of an edge is illustrated by Figure 8.7. In Figure 8.7a, we show the currently shortest path $P_{r,u}$ from r to u having cost $Dist[u]$, and the currently shortest path $P_{r,v}$ from r to v having cost $Dist[v]$, just before we consider the directed edge uv. In Figure 8.7b, by considering the edge uv, we now have a second possible path from r to v, namely, the path $P'_{r,v}$ obtained by adding the edge uv to the path $P_{r,u}$.



In Bellman–Ford Algorithm just before considering edge *uv*

(a)



In Bellman–Ford Algorithm after considering edge *uv*.
We compare weight of $P_{r,v}$ with weight of $P'_{r,v}$
and update *Dist*[*v*] with smaller of the two

(b)

Relaxing the edge *uv* in the Bellman-Ford algorithm

Figure 8.7

Note that the cost of $P'_{r,v}$ is the cost *Dist*[*u*] of the path $P_{r,u}$ plus the cost *c*(*uv*) of the edge *uv*. Hence, to see if we now have a shorter path from *r* to *v*, we simply need to check whether the cost of the path $P'_{r,v}$ is smaller than the cost of the path $P_{r,v}$, that is, we simply set

$$Dist[v] = \min\{\text{cost}(P_{r,v}) , \text{cost}(P'_{r,v})\} = \min\{Dist[v] , Dist[u] + c(uv)\}.$$

If we have now found a shorter path, that is, if $\text{cost}(P'_{r,v}) < \text{cost}(P_{r,v})$, then we set $Dist[v] = \text{cost}(P'_{r,v}) = Dist[u] + c(uv)$, and $Parent[v] = u$. As usual, the path from *r* to *v* is given (in reverse order) by the sequence

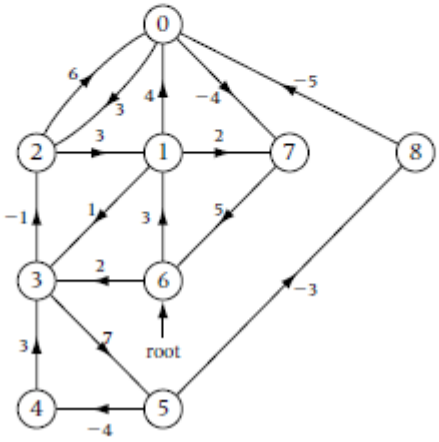
$$v, Parent[v], Parent[Parent[v]], \ldots, Parent^k[v] = r.$$

Note that the current path from r to v may be updated many times during the same pass, since there may be many edges uv in D. If D has no negative cycles, it turns out that shortest paths will have been computed after $n - 1$ scans. After completion of the first $n - 1$ scans, procedure *BellmanFord* makes a final scan of the edges to check for negative cycles. If there is any edge uv such that $Dist[v] > Dist[u] + c(uv)$, then a negative cycle exists. In the case where D has negative cycles, that is, a cycle such that the total cost of the directed paths making up the cycle is negative, then by repeatedly going around the cycle we can make the cost of paths from r to vertices in this cycle as small as we desire. The following pseudocode for the Bellman-Ford algorithm follows directly from our discussion.

```
procedure BellmanFord(D,r,c,Dist[0:n – 1],Parent[0:n – 1],NegativeCycle)
Input:    D (a digraph with vertex set V = {0,...,n – 1} and edge set E)
              c (a weighting of the edges with real numbers)
              r (a vertex of D)
Output:   Parent[0:n – 1] (an array implementing a shortest path tree rooted at r)
              Dist[0:n – 1] (an array of distances from r)
              NegativeCycle   (a Boolean variable having the value .true. if, and
                                                only if, there exists a negative cycle)
    for i ← 0 to n – 1 do          {initialize Dist[0:n – 1] and Parent[0:n – 1]}
        Dist[i] ← ∞
        Parent[i] ← ∞
    endfor
    Dist[r] ← 0
    Parent[r] ← 0
    for Pass ← 1 to n – 1 do                  // update Dist[0:n – 1] and Parent[0:n – 1]
        for each edge uv ∈ E do          // by scanning all the edges
            if Dist[u] + c(uv) < Dist[v] then
                Parent[v] ← u
                Dist[v] ← Dist[u] + c(uv)
            endif
        endfor
    endfor
    NegativeCycle ← .false. {check for negative cycles}
    for each edge uv ∈ E do
        if Dist[v] > Dist[u] + c(uv) then
            NegativeCycle ← .true.
        endif
    endfor
end BellmanFord
```

In procedure *BellmanFord* we did not specify the order in which the edges are scanned. In fact, any order will do, but the efficiency can be dramatically different for different orders. In Figure 8.8, during each pass, we scan the edges (u,v) in lexicographical order.

Pass	Index	0	1	2	3	4	5	6	7	8		0	1	2	3	4	5	6	7	8
0	*Dist*:	∞	∞	∞	∞	∞	∞	0	∞	∞	*Parent*:	∞	∞	∞	∞	∞	∞	−1	∞	∞
1	*Dist*:	∞	3	∞	2	∞	∞	0	∞	∞	*Parent*:	∞	6	∞	6	∞	∞	−1	∞	∞
2	*Dist*:	1	3	1	2	5	9	0	5	6	*Parent*:	8	6	3	6	5	3	−1	1	5
3	*Dist*:	1	3	1	2	5	9	0	−3	6	*Parent*:	8	6	3	6	5	3	−1	0	5
4	*Dist*:	1	3	1	2	5	9	0	−3	6	*Parent*:	8	6	3	6	5	3	−1	0	5
5	*Dist*:	1	3	1	2	5	9	0	−3	6	*Parent*:	8	6	3	6	5	3	−1	0	5
6	*Dist*:	1	3	1	2	5	9	0	−3	6	*Parent*:	8	6	3	6	5	3	−1	0	5
7	*Dist*:	1	3	1	2	5	9	0	−3	6	*Parent*:	8	6	3	6	5	3	−1	0	5
8	*Dist*:	1	3	1	2	5	9	0	−3	6	*Parent*:	8	6	3	6	5	3	−1	0	5



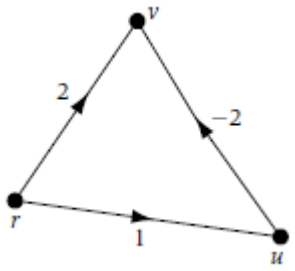
Action of procedure *BellmanFord* for a sample weighted digraph and root vertex $r = 6$, where edges (u,v) are scanned in lexicographical order.

Figure 8.8

Since procedure *BellmanFord* examines each edge n times, it has (best-case, average, and worst-case) complexity belonging to $\Theta(nm)$. The best-case and average performance of procedure *BellmanFord* can be improved by adding a flag to test whether any changes to *Dist* occur during a given pass. If no change occurs in any particular pass, then the algorithm can terminate. For example, for the digraph illustrated in Figure 8.8, only four passes are required. By adding the flag, the performance of procedure *BellmanFord* can be drastically different for different orderings of passing through the edges. To see why, consider the 3-vertex digraph shown in Figure 8.8, and note the difference between *Dist*[v] after the first pass for the two different orderings of the edges. In fact, it is not hard to give any example of a weighted digraph D on n vertices and two orderings of the edges such that procedure *BellmanFord* performs only two passes in one ordering, and the full n passes in the other ordering.

Key Fact

The correctness of either version (flag or no flag) of the procedure *BellmanFord* does not depend on the order in which the edges are scanned. However, with the flag added, the efficiency can vary drastically depending on the order in which the edges are scanned.



First pass of procedure *BellmanFord*, where edges are considered in the order *rv*, *uv*, *ru* would yield *Dist*[*v*] = 2, but the order *ru*, *uv*, *rv* would yield *Dist*[*u*,*v*] = − 1.

Figure 8.9

The correctness of procedure *BellmanFord* involves establishing the loop invariant stated in the following lemma.

Lemma 8.6.1

After completing *k* iterations of the *Pass* **for** loop, for each vertex *v*, *Dist*[*v*] is no larger than the minimum length $L_{v,k}$ of a path from *r* to *v* having at most *k* edges (where $L_{v,k} = \infty$ if no such path exists), $k = 0,...,n-1$.

Proof We prove the lemma by induction on the number *k* of passes that have been completed. Clearly, the lemma is true before any pass has been completed, so that the basis step is established. Now assume that after *k* passes have been completed, for each vertex *v*, *Dist*[*v*] is no larger than the minimum length $L_{v,k}$ of a path from *r* to *v* having at most *k* edges. Consider any vertex *v* for which there is a path from *r* to *v* having at most *k* + 1 edges (the lemma is trivially true if there is no such path), and let *P* be such a path of minimum length $L_{v,k}$. Let *uv* be the terminal edge of *P*, and let *Q* be the subpath of *P* from *r* to *u*. Since *Q* has at most *k* edges, the length of *Q* is at least $L_{u,k}$. Thus, the length of *P* is at least $L_{u,k} + c(uv)$. But, by induction assumption, after the *k*th pass, *Dist*[*u*] is at most $L_{u,k}$. Therefore, after the (*k* + 1)st pass, we have $Dist[v] \le Dist[u] + c(uv) \le L_{u,k} + c(uv) \le$ length of $P = L_{v,k}$. Thus, the loop invariant holds after *k* + 1 passes. ■
When there are no negative cycles, a shortest path contains at most *n* – 1 edges. Hence, the correctness of *BellmanFord* when there are no negative cycles follows immediately from Lemma 8.6.1. We leave the correctness proof of *BellmanFord* when there are negative cycles as an exercise.

8.7 Closing Remarks

Floyd's algorithm is also known as the Roy–Warshall algorithm, the Roy–Floyd algorithm, or the WFI algorithm. Floyd published his algorithm in 1962, but it was

essentially the same as algorithms published by Bernard Roy in 1959 and by Stephen Warshall's in 1962 for finding the transitive closure of a graph.

The dynamic programming paradigm was first formulated by Richard Bellman. His interest was in solving optimization problems that involved making a sequence of decisions (resulting in problem states $x_0, x_1, \ldots$) that where guaranteed to yield an optimal solution (highest value) V for a value function F associated with a given problem. He stated the following condition that formed the basis for a recursive solution to the problem.

Principle of Optimality: An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

The *Bellman Equation* results from expressing the optimal value function at a given initial state x_0 in terms of its value at the state resulting from making the initial decision. Suppose $D(x_i)$ denotes the set of possible decisions available at stage x_i, $i = 0,1, \ldots$, and let T denote that transition function that determines the next stage x_{i+1} resulting from making the decision $d_i \varepsilon D(x_i)$ at stage x_i, $i = 0,1, \ldots$. Then the principle of optimality results in the following recursive Bellman Equation for the optimal value V

$$V(x_0) = \max\{ F(x_0, d_0) + V(T(x_0, d_0)) \} \text{ over all } d_0 \varepsilon D(x_0) . \qquad (8.7.1)$$

The recursive expression then is solved in a bottom-up resolution of the recursion in the manner that we have described in this chapter.

The Bellman Equation has been applied in a wide variety of optimization problems. In many applications, especially in financial problems, the term $V(T(x_0, d_0))$ in (8.7.1) is multiplied by a so-called *discount factor* β.

In Exercise 8.16 we develop a dynamic programming solution to the Traveling Salesman Problem (TSP). TSP involves finding a minimum length tour of n cities, starting and ending at a given city. TSP is a rather famous problem, and a good deal of research has been on this problem. Unfortunately, no polynomial algorithm has been found for its solution, and in fact it is NP-hard (so that most researchers believe no polynomial solution for TSP will ever be found). The dynamic programming solution discussed in Exercise 8.16 has complexity in $\Theta(n2^n)$, which at least is rather better than the brute force method of enumerating the $(n - 1)!$ possible tours.

Exercises

Section 8.1 Optimization Problems and the Principle of Optimality

8.1 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.

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

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

Section 8.2 Optimal Parenthesization for Computing a Chained Matrix Product

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

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

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

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

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

Section 8.3 Optimal Binary Search Trees

8.9 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, $q_i = 0$, $i = 0,...,n$:

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

8.10 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

8.11 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 8.9.

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

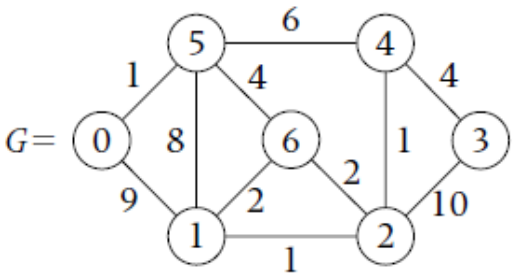
8.4 Longest Common Subsequence

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

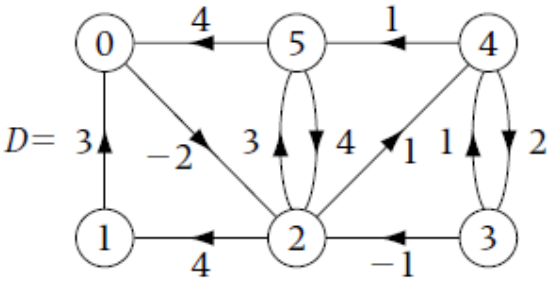
8.14 By following various paths in the array *LCS*[0:8,0:11] given in Figure 8.8, find all longest common subsequences of the strings *usbeeune* and *subsequence*.
8.15 Design and analyze an algorithm that generates all longest common subsequences given the input array *LCS*[0:*n* – 1, 0:*m* – 1].
8.16 Write a program that implements *LongestCommonSubseq* and run it for some sample inputs.

8.5 Floyd's Algorithm

8.17 Show the action of Floyd's algorithm for the following graph.



8.18 Given the following weighted digraph *D*, show the matrices S_k[0:5,0:5], P_k[0:5,0:5], $k = 0,...,5$ as computed by Floyd's algorithm. Describe how the matrix P_5 can be used, for example, to find the shortest path from vertex 1 to vertex 3.



8.19 Give pseudocode for an algorithm that computes the shortest path from *i* to *j* from the matrix *P*[0:*n* – 1,0:*n* – 1] generated by Floyd's algorithm.
8.20 Alter the procedure Floyd to store in *P*[*i*,*j*] the first vertex encountered on the current path from *i* to *j*. Then repeat the previous exercise for this altered matrix *P*[0:*n* – 1,0:*n* – 1].
8.21 Prove that Floyd's algorithm works even if some of the edges have negative weights, provided that there is no negative weight cycle.
8.22 Show how Floyd's algorithm can be used to obtain longest paths between every pair of vertices in a dag.

8.6 The Bellman-Ford Algorithm

8.23 a. Redo the action of procedure *BellmanFord* shown in Figure 8.8 when *r* = 2.

b. Redo the action of procedure *BellmanFord* shown in Figure 8.8 when $r = 6$ and where the cost of the edge (3,2) is changed to –2.

8.24 a. Design a modification of procedure *BellmanFord* that improves the best-case and average performance by terminating if no changes to *Dist* occur during a pass.

b. Analyze the best-case and worst-case complexity of your algorithm in part (a) in terms of the parameters n and m. Explicitly exhibit input digraphs achieving the best-case and worst-case complexities, respectively.

c. Repeat part (b) in terms of the single parameter n.

8.25 A high-level description of procedure *BellmanFord* is given in Section 8.6. Give a description of procedure *BellmanFord* for the following implementations of the digraph.

a. adjacency matrix

b. adjacency lists

8.26 Show that if a weighted digraph contains no negative or zero cycles, then a shortest path contains at most $n - 1$ edges. Conclude the correctness proof of procedure *BellmanFord* when there are no negative cycles.

8.27 Give the correctness proof of procedure *BellmanFord* when there are negative cycles.

8.28 Modify procedure *BellmanFord* to output a negative cycle, when one exists.

8.29 Consider the version of procedure *BellmanFord*, which checks each pass for any change in $Dist[0{:}n-1]$. For this version of procedure *BellmanFord*, the order in which the edges are scanned can significantly affect the worst-case performance of the algorithm. Give an example of a weighted digraph, and two orderings A and B of the edges, such that if ordering A is used during each pass, procedure *BellmanFord* performs only two passes, but if ordering B is used during each pass, procedure *BellmanFord* performs n passes.

Additional Problems

8.30 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

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

8.31 The 0/1 Knapsack problem is NP-hard when the input is measured in binary. However, when the input is measured in unary. 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 $\{b_0, \ldots, b_{i-1}\}$. Use the principle of optimality to find a recurrence relation for $V[i,j]$.

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

8.33 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.