Lecture Notes on
Analysis and Design of Algorithms
BCS401

Module-4 : Dynamic Programming

Contents

1. Introduction to Dynamic Programming

Dynamic programming is a technique for solving problems with **overlapping subproblems**. Typically, these subproblems arise from a recurrence relating a given problem's solution to solutions of its smaller subproblems. Rather than solving overlapping subproblems again and again, dynamic programming suggests solving each of the smaller subproblems only once and recording the results in a table from which a solution to the original problem can then be obtained. *[From T1]*

The Dynamic programming can be used when the solution to a problem can be viewed as the result of **sequence of decisions**. *[From T2]*. Here are some examples.

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

Example 2 The files x_1, x_2, and x_3 are three sorted files of length 30, 20, and 10 records each. Merging x_1 and x_2 requires 50 record moves. Merging the result with x_3 requires another 60 moves. The total number of record moves required to merge the three files this way is 110. If, instead, we first merge x_2 and x_3 (taking 30 moves) and then x_1 (taking 60 moves), the total record moves made is only 90. Hence, the second merge pattern is faster than the first.

An optimal merge pattern tells us which pair of files should be merged at each step. As a decision sequence, the problem calls for us to decide which pair of files should be merged first, which pair second, which pair third, and so on. An optimal sequence of decisions is a least-cost sequence.

Example 3 [Shortest path] One way to find a shortest path from vertex i to vertex j in a directed graph G is to decide which vertex should be the second vertex, which the third, which the fourth, and so on, until vertex j is reached. An optimal sequence of decisions is one that results in a path of least length. □

Example 4 [Shortest path] Suppose we wish to find a shortest path from vertex i to vertex j. Let A_i be the vertices adjacent from vertex i. Which of the vertices in A_i should be the second vertex on the path? There is no way to make a decision at this time and guarantee that future decisions leading to an optimal sequence can be made. If on the other hand we wish to find a shortest path from vertex i to all other vertices in G, then at each step, a correct decision can be made □

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

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

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

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

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

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

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

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

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

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

□

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

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

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

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

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

1.2 Multistage Graphs

A multistage graph $G = (V, E)$ is a directed graph in which the vertices are partitioned into $k \ge 2$ disjoint sets V_i, $1 \le i \le k$. In addition, if $\langle u, v \rangle$ is an edge in E, then $u \in V_i$ and $v \in V_{i+1}$ for some i, $1 \le i < k$. The sets V_1 and V_k are such that $|V_1| = |V_k| = 1$. Let s and t, respectively, be the vertices in V_1 and V_k. The vertex s is the *source*, and t the *sink*. Let $c(i, j)$ be the cost of edge $\langle i, j \rangle$. The cost of a path from s to t is the sum of the costs of the edges on the path. The *multistage graph problem* is to find a minimum-cost path from s to t. Each set V_i defines a stage in the graph. Because of the constraints on E, every path from s to t starts in stage 1, goes to stage 2, then to stage 3, then to stage 4, and so on, and eventually terminates in stage k. Figure 5.2 shows a five-stage graph. A minimum-cost s to t path is indicated by the broken edges.

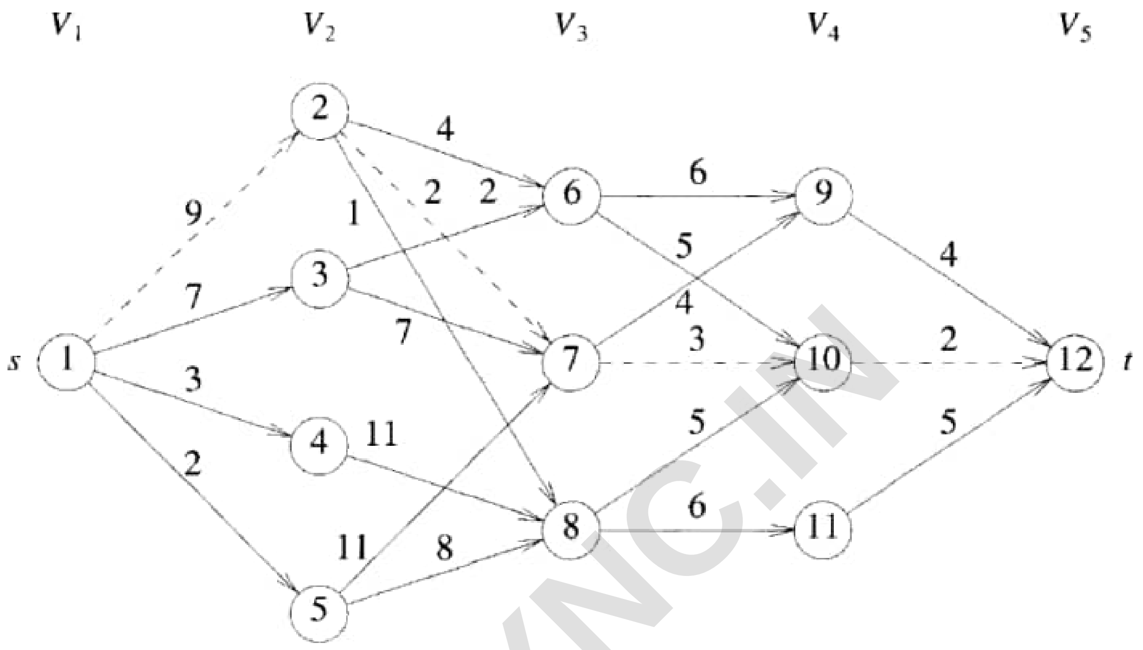


Figure: Five stage graph

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

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

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

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

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

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

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

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

Algorithm 5.1 Multistage graph pseudocode corresponding to the forward approach

```
Algorithm FGraph($G, k, n, p$)
// The input is a $k$-stage graph $G = (V, E)$ with $n$ vertices
// indexed in order of stages. $E$ is a set of edges and $c[i, j]$
// is the cost of $\langle i, j \rangle$. $p[1 : k]$ is a minimum-cost path.
{
    $cost[n] := 0.0$;
    for $j := n - 1$ to 1 step $-1$ do
    { // Compute $cost[j]$.
        Let $r$ be a vertex such that $\langle j, r \rangle$ is an edge
        of $G$ and $c[j, r] + cost[r]$ is minimum;
        $cost[j] := c[j, r] + cost[r]$;
        $d[j] := r$;
    }
    // Find a minimum-cost path.
    $p[1] := 1$; $p[k] := n$;
    for $j := 2$ to $k - 1$ do $p[j] := d[p[j - 1]]$;
}
```

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

Backward Approach

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

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

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

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

$$\begin{aligned} bcost(3,7) &= 11 & \qquad bcost(4,10) &= 14 \\ bcost(3,8) &= 10 & bcost(4,11) &= 16 \\ bcost(4,9) &= 15 & bcost(5,12) &= 16 \end{aligned}$$

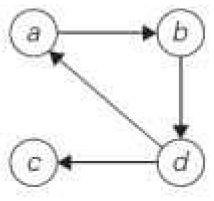
Algorithm 5.2 Multistage graph pseudocode corresponding to backward approach

```
Algorithm BGraph($G, k, n, p$)
// Same function as FGraph
{
    $bcost[1] := 0.0$;
    for $j := 2$ to $n$ do
    { // Compute $bcost[j]$.
        Let $r$ be such that $\langle r, j\rangle$ is an edge of
        $G$ and $bcost[r] + c[r, j]$ is minimum;
        $bcost[j] := bcost[r] + c[r, j]$;
        $d[j] := r$;
    }
    // Find a minimum-cost path.
    $p[1] := 1$; $p[k] := n$;
    for $j := k - 1$ to 2 do $p[j] := d[p[j + 1]]$;
}
```

2. Transitive Closure using Warshall's Algorithm,

Definition: The **transitive closure** of a directed graph with n vertices can be defined as the n × n boolean matrix T = $\{t_{ij}\}$, in which the element in the ith row and the jth column is 1 if there exists a nontrivial path (i.e., directed path of a positive length) from the ith vertex to the jth vertex; otherwise, t^{ij} is 0.

Example: An example of a digraph, its adjacency matrix, and its transitive closure is given below.



(a) Digraph.

$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

(b) Its adjacency matrix.

$$T = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

(c) Its transitive closure.

We can generate the transitive closure of a digraph with the help of depth first search or breadth-first search. Performing either traversal starting at the ith vertex gives the information about the vertices reachable from it and hence the columns that contain 1's in the ith row of the transitive closure. Thus, doing such a traversal for every vertex as a starting point yields the transitive closure in its entirety.

Since this method traverses the same digraph several times, we can use a better algorithm called **Warshall's algorithm**. Warshall's algorithm constructs the transitive closure through a series of n × n boolean matrices:

$$R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots R^{(n)}.$$

Each of these matrices provides certain information about directed paths in the digraph. Specifically, the element$r_{ij}^{(k)}$ in the ith row and jth column of matrix $R^{(k)}$ (i, j = 1, 2, . . . , n, k = 0, 1, . . . , n) is equal to 1 if and only if there exists a directed path of a positive length from the ith vertex to the jth vertex with each intermediate vertex, if any, numbered not higher than k.

Thus, the series starts with $R^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $R^{(0)}$ is nothing other than the adjacency matrix of the digraph. $R^{(1)}$ contains the information about paths that can use the first vertex as intermediate. The last matrix in the series, $R^{(n)}$,reflects paths that can use all n vertices of the digraph as intermediate and hence is nothing other than the digraph's transitive closure.

This means that there exists a path from the ith vertex vi to the jth vertex vj with each intermediate vertex numbered not higher than k:

v_i, a list of intermediate vertices each numbered not higher than k, v_j . --- (*)

Two situations regarding this path are possible.

1. In the first, the list of its intermediate vertices **does not** contain the kth vertex. Then this path from v_i to v_j has intermediate vertices numbered not higher than $k-1$. i.e. $r_{ij}^{(k-1)} = 1$
2. The second possibility is that path (*) **does contain** the kth vertex v_k among the intermediate vertices. Then path (*) can be rewritten as;

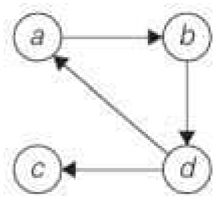
$$v_i, \text{ vertices numbered} \leq k-1, v_k, \text{ vertices numbered} \leq k-1, v_j .$$

$$\text{i.e } r_{ik}^{(k-1)} = 1 \text{ and } r_{kj}^{(k-1)} = 1$$

Thus, we have the following formula for generating the elements of matrix $R^{(k)}$ from the elements of matrix $R^{(k-1)}$

The Warshall's algorithm works based on the above formula.

As an example, the application of Warshall's algorithm to the digraph is shown below. New 1's are in bold.



$$R^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

1's reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

$$R^{(1)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & \mathbf{1} & 1 & 0 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex *a* (note a new path from *d* to *b*); boxed row and column are used for getting $R^{(2)}$.

$$R^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & \mathbf{1} \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & \mathbf{1} \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., *a* and *b* (note two new paths); boxed row and column are used for getting $R^{(3)}$.

$$R^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 1 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., *a, b,* and *c* (no new paths); boxed row and column are used for getting $R^{(4)}$.

$$R^{(4)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & \mathbf{1} & 1 & \mathbf{1} & 1 \\ b & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., *a, b, c,* and *d* (note five new paths).

```
ALGORITHM  Warshall(A[1..n, 1..n])
    //Implements Warshall's algorithm for computing the transitive closure
    //Input: The adjacency matrix A of a digraph with n vertices
    //Output: The transitive closure of the digraph
    $R^{(0)} \leftarrow A$
    for $k \leftarrow 1$ to $n$ do
        for $i \leftarrow 1$ to $n$ do
            for $j \leftarrow 1$ to $n$ do
                $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ or $(R^{(k-1)}[i, k]$ and $R^{(k-1)}[k, j])$
    return $R^{(n)}$
```

Analysis

Its time efficiency is $\Theta(n^3)$. We can make the algorithm to run faster by treating matrix rows as bit strings and employ the bitwise or operation available in most modern computer languages.

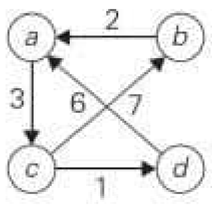
Space efficiency: Although separate matrices for recording intermediate results of the algorithm are used, that can be avoided.

3. All Pairs Shortest Paths using Floyd's Algorithm,

Problem definition: Given a weighted connected graph (undirected or directed), the all-pairs shortest paths problem asks to find the distances—i.e., the lengths of the shortest paths - from each vertex to all other vertices.

Applications: Solution to this problem finds applications in communications, transportation networks, and operations research. Among recent applications of the all-pairs shortest-path problem is pre-computing distances for motion planning in computer games.

We store the lengths of shortest paths in an n x n matrix D called the distance matrix: the element d_{ij} in the ith row and the jth column of this matrix indicates the length of the shortest path from the ith vertex to the jth vertex.



$$W = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \infty & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \infty & 0 \end{array}$$

$$D = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & 7 & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{array}$$

(a) Digraph. (b) Its weight matrix. (c) Its distance matrix

We can generate the distance matrix with an algorithm that is very similar to Warshall's algorithm. It is called **Floyd's algorithm.**

Floyd's algorithm computes the distance matrix of a weighted graph with n vertices through a series of n × n matrices:

$$D^{(0)}, \ldots, D^{(k-1)}, D^{(k)}, \ldots, D^{(n)}.$$

The element $^{(k)}_{ij}$ in the ith row and the jth column of matrix $D^{(k)}$ (i, j = 1, 2, . . . , n, k = 0, 1, . . . , n) is equal to the length of the shortest path among all paths from the ith vertex to the jth vertex with each intermediate vertex, if any, numbered not higher than k.

As in Warshall's algorithm, we can compute all the elements of each matrix $D^{(k)}$ from its immediate predecessor $D^{(k-1)}$

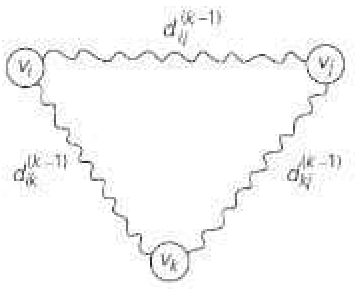
If $^{(k)}_{ij} = 1$, then it means that there is a path;

vi, a list of intermediate vertices each numbered not higher than k, vj .

We can partition all such paths into two disjoint subsets: those that do not use thekth vertex v_k as intermediate and those that do.

i. Since the paths of the first subset have their intermediate vertices numbered not higher than k − 1, the shortest of them is, by definition of our matrices, of length $d_{ij}^{(k-1)}$
ii. In the second subset the paths are of the form
   v_i, vertices numbered ≤ k − 1, v_k, vertices numbered ≤ k − 1, v_j .

The situation is depicted symbolically in Figure, which shows the underlying idea of Floyd's algorithm.



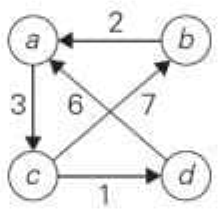
Taking into account the lengths of the shortest paths in both subsets leads to the following recurrence:

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \quad \text{for } k \geq 1,\quad d_{ij}^{(0)} = w_{ij}.$$

```
ALGORITHM  Floyd(W[1..n, 1..n])
    //Implements Floyd's algorithm for the all-pairs shortest-paths problem
    //Input: The weight matrix W of a graph with no negative-length cycle
    //Output: The distance matrix of the shortest paths' lengths
    D ← W  //is not necessary if W can be overwritten
    for k ← 1 to n do
        for i ← 1 to n do
            for j ← 1 to n do
                D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}
    return D
```

Analysis: Its time efficiency is $\Theta(n^3)$, similar to the warshall's algorithm.

Application of Floyd's algorithm to the digraph is shown below. Updated elements are shown in bold.



$$D^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \infty & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \infty & 0 \end{array}$$

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

$$D^{(1)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \mathbf{5} & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \mathbf{9} & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e., just *a* (note two new shortest paths from *b* to *c* and from *d* to *c*).

$$D^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & 5 & \infty \\ c & \mathbf{9} & 7 & 0 & 1 \\ d & 6 & \infty & 9 & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e., a and *b* (note a new shortest path from *c* to *a*).

$$D^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \mathbf{10} & 3 & \mathbf{4} \\ b & 2 & 0 & 5 & \mathbf{6} \\ c & 9 & 7 & 0 & 1 \\ d & 6 & \mathbf{16} & 9 & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e., a, *b*, and *c* (note four new shortest paths from *a* to *b*, from a to *d*, from *b* to *d*, and from *d* to *b*).

$$D^{(4)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & \mathbf{7} & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e., a, *b*, *c*, and *d* (note a new shortest path from *c* to *a*).

4. Knapsack problem

We start this section with designing a dynamic programming algorithm for the knapsack problem: given n items of known weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack. To design a dynamic programming algorithm, we need to derive a recurrence relation that expresses a solution to an instance of the knapsack problem in terms of solutions to its smaller sub instances. Let us consider an instance defined by the first i items, $1 \le i \le n$, with weights $w_1, \ldots, w_i$, values $v_1, \ldots, v_i$, and knapsack capacity j, $1 \le j \le W$. Let F(i, j) be the value of an optimal solution to this instance. We can divide all the subsets of the first i items that fit the knapsack of capacity j into two categories: those that do not include the i^{th} item and those that do. Note the following:

i. Among the subsets that **do not include the ith item**, the value of an optimal subset is, by definition, i.e $F(i , j) = F(i - 1, j)$.

ii. Among the subsets that **do include the ith item** (hence, $j - w_i \geq 0$), an optimal subset is made up of this item and an optimal subset of the first $i-1$ items that fits into the knapsack of capacity $j - w_i$. The value of such an optimal subset is $v_i + F(i - 1, j - w_i)$.

Thus, the value of an optimal solution among all feasible subsets of the first I items is the maximum of these two values.

$$F(i, j) = \begin{cases} \max\{F(i-1, j), v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases}$$

It is convenient to define the initial conditions as follows:

$$F(0, j) = 0 \text{ for } j \geq 0 \text{ and } F(i, 0) = 0 \text{ for } i \geq 0.$$

Our goal is to find **F(n, W),** the maximal value of a subset of the n given items that fit into the knapsack of capacity W, and an optimal subset itself.

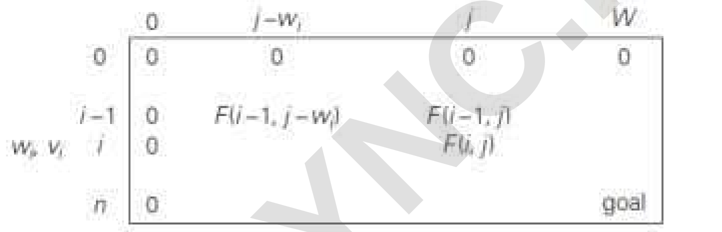


Table for solving the knapsack problem by dynamic programming.

The algorithm for the knapsack problem can be stated as follows

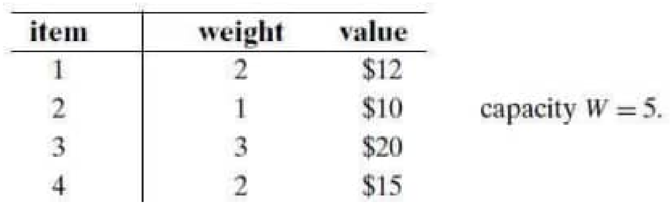
Input: **n** – total items, **W** – capacity of the knapsack
w_i– weight of the ith item, v_i– value of the ith item,

Output: F(i, j) be the value of an optimal solution to this instance considering first i items with capacity j. F(n,W) is the optimal solution

Method:

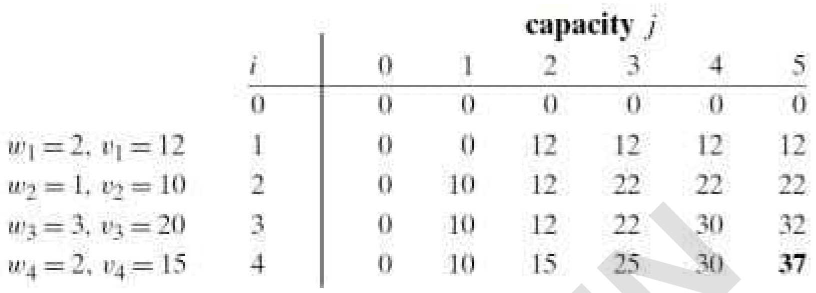
```
for w = 0 to W
    F[0,w] = 0
for i = 1 to n
    F[i,0] = 0
for i = 1 to n
    for w = 0 to W
        if w_i <= w // item i can be part of the solution
            if v_i + F[i-1,w-w_i] > F[i-1,w]
                F[i,w] = v_i + F[i-1,w- w_i]
            else
                F[i,w] = F[i-1,w]
        else F[i,w] = F[i-1,w]  // w_i > w
```

Example-1: Let us consider the instance given by the following data:



item	weight	value	
1	2	\$12	
2	1	\$10	capacity $W = 5$.
3	3	\$20	
4	2	\$15	

The dynamic programming table, filled by applying formulas is given below



		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3, v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2, v_4 = 15$	4	0	10	15	25	30	**37**

Thus, the maximal value is F(4, 5) = \$37.

We can find the composition of an optimal subset by back tracing the computations of this entry in the table. Since F(4, 5) > F(3, 5), item 4 has to be included in an optimal solution along with an optimal subset for filling 5 − 2 = 3 remaining units of the knapsack capacity. The value of the latter is F(3, 3). Since F(3, 3) = F(2, 3), item 3 need not be in an optimal subset. Since F(2, 3) > F(1, 3), item 2 is a part of an optimal selection, which leaves element F(1, 3 − 1) to specify its remaining composition. Similarly, since F(1, 2) > F(0, 2), item 1 is the final part of the optimal solution {item 1, item 2, item 4}.

Analysis

The time efficiency and space efficiency of this algorithm are both in Θ(nW). The time needed to find the composition of an optimal solution is in O(n).

Memory Functions

The direct top-down approach to finding a solution to such a recurrence leads to an algorithm that solves common subproblems more than once and hence is very inefficient.

The classic dynamic programming approach, on the other hand, works bottom up: it fills a table with solutions to all smaller subproblems, but each of them is solved only once. An unsatisfying aspect of this approach is that solutions to some of these smaller subproblems are often not necessary for getting a solution to the problem given. Since this drawback is not present in the top-down approach, it is natural to try to combine the strengths of the top-down and bottom-up approaches. The goal is to get a method that solves only subproblems that are necessary and does so only once. Such a method exists; it is based on using **memory functions**.

This method solves a given problem in the top-down manner but, in addition, maintains a table of the kind that would have been used by a bottom-up dynamic programming algorithm.

Initially, all the table's entries are initialized with a special "null" symbol to indicate that they have not yet been calculated. Thereafter, whenever a new value needs to be calculated, the method checks the corresponding entry in the table first: if this entry is not "null," it is simply retrieved from the table; otherwise, it is computed by the recursive call whose result is then recorded in the table.

The following algorithm implements this idea for the knapsack problem. After initializing the table, the recursive function needs to be called with i = n (the number of items) and j = W (the knapsack capacity).

AlgorithmMFKnapsack(i, j)
//Implements the memory function method for the knapsack problem
//**Input:** A nonnegative integer i indicating the number of the first items being considered and a nonnegative integer j indicating the knapsack capacity
//**Output:** The value of an optimal feasible subset of the first i items
//**Note:** Uses as global variables input arrays Weights[1..n], Values[1..n],and table F[0..n, 0..W] whose entries are initialized with −1's except for row 0 and column 0 initialized with 0's

```
if $F[i, j] < 0$
    if $j < Weights[i]$
        $value \leftarrow MFKnapsack(i-1, j)$
    else
        $value \leftarrow \max(MFKnapsack(i-1, j),$
                        $Values[i] + MFKnapsack(i-1, j-Weights[i]))$
    $F[i, j] \leftarrow value$
return $F[i, j]$
```

Example-2 Let us apply the memory function method to the instance considered in Example 1. The table in Figure given below gives the results. Only 11 out of 20nontrivial values (i.e., not those in row 0 or in column 0) have been computed. Just one nontrivial entry, V (1, 2), is retrieved rather than being recomputed. For larger instances, the proportion of such entries can be significantly larger.

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	—	12	22	—	22
$w_3 = 3, v_3 = 20$	3	0	—	—	22	—	32
$w_4 = 2, v_4 = 15$	4	0	—	—	—	—	**37**

Figure: Example of solving an instance of the knapsack problem by the memory function algorithm

In general, we cannot expect more than a constant-factor gain in using the memory function method for the knapsack problem, because its time efficiency class is the same as that of the bottom-up algorithm

5. Bellman-Ford Algorithm (Single source shortest path with –ve weights)

Problem definition: Given a graph and a source vertex ***s*** in graph, find shortest paths from ***s*** to all vertices in the given graph. The graph may contain negative weight edges.

Note that we have discussed Dijkstra's algorithm for single source shortest path problem. Dijksra's algorithm is a Greedy algorithm and time complexity is O(VlogV). But Dijkstra doesn't work for graphs with negative weight edges.

Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.

How it works? - Like other Dynamic Programming Problems, the algorithm calculates shortest paths in ***bottom-up manner***. It first calculates the shortest distances for the shortest paths which have at-most one edge in the path. Then, it calculates shortest paths with at-most 2 edges, and so on.

Iteration i finds all shortest paths that use i edges. There can be maximum |V| – 1 edges in any simple path, that is why the outer loop runs |v| – 1 times. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edges

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

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

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

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

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

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

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

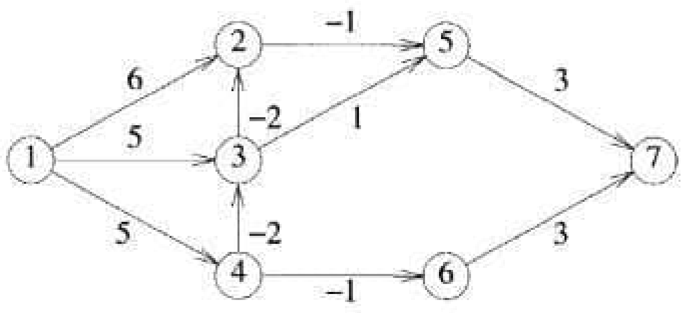
Bellman-Ford algorithm to compute shortest path

```
**Algorithm** BellmanFord($v, cost, dist, n$)
// Single-source/all-destinations shortest
// paths with negative edge costs
{
    for $i := 1$ to $n$ do // Initialize $dist$.
        $dist[i] := cost[v, i]$;
    for $k := 2$ to $n - 1$ do
        for each $u$ such that $u \neq v$ and $u$ has
                at least one incoming edge do
            for each $\langle i, u \rangle$ in the graph do
                if $dist[u] > dist[i] + cost[i, u]$ then
                    $dist[u] := dist[i] + cost[i, u]$;
}
```

Example 5.16 Figure 5.10 gives a seven-vertex graph, together with the arrays $dist^k$, $k = 1, \ldots, 6$. These arrays were computed using the equation just given. For instance, $dist^k[1] = 0$ for all k since 1 is the source node. Also, $dist^1[2] = 6$, $dist^1[3] = 5$, and $dist^1[4] = 5$, since there are edges from 1 to these nodes. The distance $dist^1[\,]$ is ∞ for the nodes 5, 6, and 7 since there are no edges to these from 1.

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

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



(a) A directed graph

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

(b) $dist^k$

Figure 5.10 Shortest paths with negative edge lengths

Another example

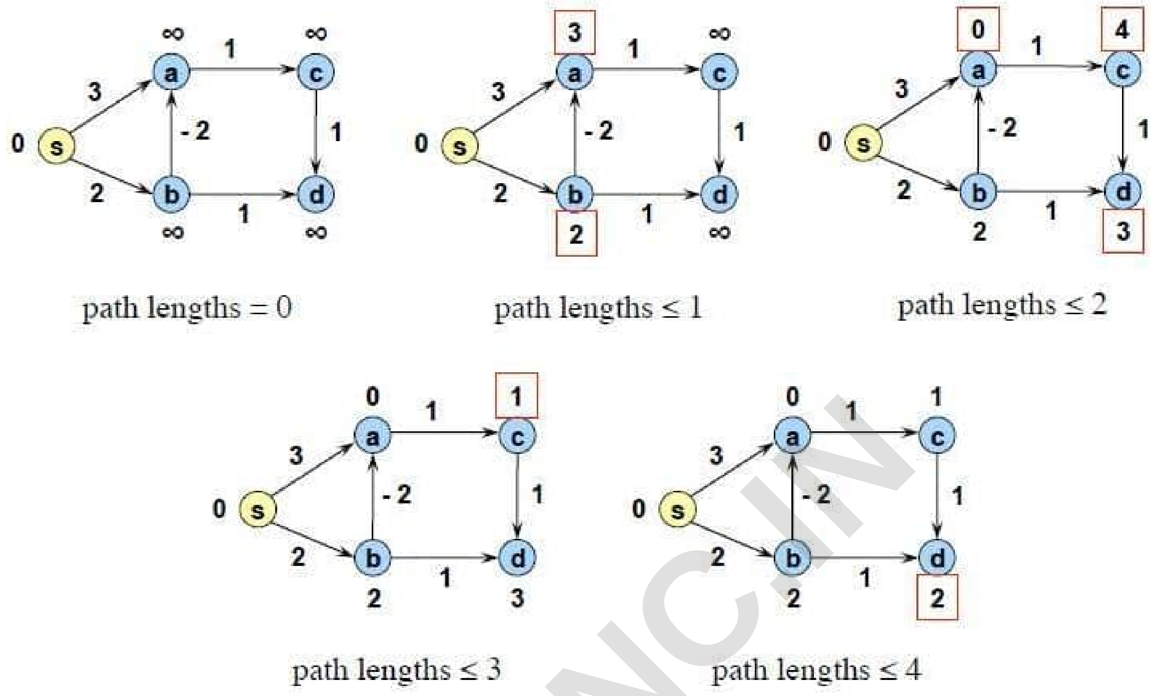


Figure: Steps of the Bellman Ford algorithm. The numbers with red squares indicate what changed on each step.

6. Travelling Sales Person problem (T2:5.9),

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

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

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

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

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

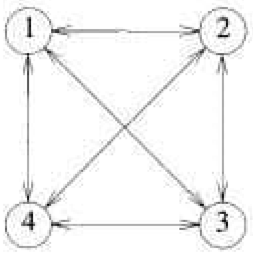
Generalizing (5.20), we obtain (for $i \notin S$)

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

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

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

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



(a)

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

(b)

Figure 5.21 Directed graph and edge length matrix c

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

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

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

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

Finally, from (5.20) we obtain

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

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

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

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

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

***

7. Space-Time Tradeoffs

Introduction

The main idea is to preprocess the problem's input, in whole or in part, and store the additional information obtained to accelerate solving the problem afterward. We call this approach as input enhancement. The algorithms based on this approach are:

- Counting methods for sorting
- Boyer-Moore algorithm for string matching and its simplified version suggested by Horspool

Sorting by Counting

The main idea here is to use the count variable. This variable shall give us the position of the elements in its sorted order. Keeping the count variable as reference we can copy the elements to a new list, so that we have a sorted list with us. The algorithm to do his is called the ***comparison counting sort***.

Array $A[0..5]$		62	31	84	96	19	47
Initially	$Count$ []	0	0	0	0	0	0
After pass $i = 0$	$Count$ []	3	0	1	1	0	0
After pass $i = 1$	$Count$ []		1	2	2	0	1
After pass $i = 2$	$Count$ []			4	3	0	1
After pass $i = 3$	$Count$ []				5	0	1
After pass $i = 4$	$Count$ []					0	2
Final state	$Count$ []	3	1	4	5	0	2
Array $S[0..5]$		19	31	47	62	84	96

FIGURE 7.1 Example of sorting by comparison counting

```
ALGORITHM  ComparisonCountingSort(A[0..n − 1])
    //Sorts an array by comparison counting
    //Input: An array A[0..n − 1] of orderable elements
    //Output: Array S[0..n − 1] of A's elements sorted in nondecreasing order
    for i ← 0 to n − 1 do Count[i] ← 0
    for i ← 0 to n − 2 do
        for j ← i + 1 to n − 1 do
            if A[i] < A[j]
                Count[j] ← Count[j] + 1
            else Count[i] ← Count[i] + 1
    for i ← 0 to n − 1 do S[Count[i]] ← A[i]
    return S
```

What is the time efficiency of this algorithm? It should be quadratic because the algorithm considers all the different pairs of an n-element array. More formally, the number of times its basic operation, the comparison $A[i] < A[j]$, is executed is equal to the sum we have encountered several times already:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) = \frac{n(n-1)}{2}.$$

Since the algorithm makes the same number of key comparisons as selection sort and in addition uses a linear amount of extra space, it can hardly be recommended for practical use.

But the counting idea does work productively in a situation in which elements to be sorted belong to a known small set of values. Assume, for example, that we have to sort a list whose values can be either 1 or 2. Rather than applying a general sorting algorithm, we should be able to take advantage of this additional information about values to be sorted. Indeed, we can scan the list to compute the number of 1's and the number of 2's in it and then, on the second pass, simply make the appropriate number of the first elements equal to 1 and the remaining elements equal to 2. More generally, if element values are integers between some lower bound l and upper bound u, we can compute the frequency of each of those values and store them in array $F[0..u-l]$. Then the first $F[0]$ positions in the sorted list must be filled with l, the next $F[1]$ positions with $l+1$, and so on. All this can be done, of course, only if we can overwrite the given elements.

Let us consider a more realistic situation of sorting a list of items with some other information associated with their keys so that we cannot overwrite the list's elements. Then we can copy elements into a new array $S[0..n-1]$ to hold the sorted list as follows. The elements of A whose values are equal to the lowest possible value l are copied into the first $F[0]$ elements of S, i.e., positions 0 through $F[0]-1$, the elements of value $l+1$ are copied to positions from $F[0]$ to $(F[0]+F[1])-1$, and so on. Since such accumulated sums of frequencies are called a distribution in statistics, the method itself is known as ***distribution counting***.

EXAMPLE Consider sorting the array

13	11	12	13	12	12

whose values are known to come from the set {11, 12, 13} and should not be overwritten in the process of sorting. The frequency and distribution arrays are as follows:

Array values	11	12	13
Frequencies	1	3	2
Distribution values	1	4	6

Note that the distribution values indicate the proper positions for the last occurrences of their elements in the final sorted array. If we index array positions from 0 to $n-1$, the distribution values must be reduced by 1 to get corresponding element positions.

It is more convenient to process the input array right to left. For the example, the last element is 12, and, since its distribution value is 4, we place this 12 in position $4-1=3$ of the array S that will hold the sorted list. Then we decrease the 12's distribution value by 1 and proceed to the next (from the right) element in the given array. The entire processing of this example is depicted in Figure 7.2. ∎

Here is a pseudocode of this algorithm.

ALGORITHM *DistributionCounting*($A[0..n-1], l, u$)

```
//Sorts an array of integers from a limited range by distribution counting
//Input: An array $A[0..n-1]$ of integers between $l$ and $u$ ($l \le u$)
//Output: Array $S[0..n-1]$ of $A$'s elements sorted in nondecreasing order
for $j \leftarrow 0$ to $u-l$ do $D[j] \leftarrow 0$    //initialize frequencies
```

	D[0..2]			S[0..5]					
A [5] = 12	1	**4**	6				12		
A [4] = 12	1	**3**	6			12			
A [3] = 13	1	2	**6**						13
A [2] = 12	1	**2**	5		12				
A [1] = 11	**1**	1	5	11					
A [0] = 13	0	1	**5**					13	

FIGURE 7.2 Example of sorting by distribution counting. The distribution values being decremented are shown in bold.

```
for $i \leftarrow 0$ to $n-1$ do $D[A[i]-l] \leftarrow D[A[i]-l]+1$ //compute frequencies
for $j \leftarrow 1$ to $u-l$ do $D[j] \leftarrow D[j-1]+D[j]$ //reuse for distribution
for $i \leftarrow n-1$ downto 0 do
    $j \leftarrow A[i]-l$
    $S[D[j]-1] \leftarrow A[i]$
    $D[j] \leftarrow D[j]-1$
return $S$
```

Assuming that the range of array values is fixed, this is obviously a linear algorithm because it makes just two consecutive passes through its input array A. This is a better time-efficiency class than that of the most efficient sorting algorithms—mergesort, quicksort, and heapsort—we have encountered. It is important to remember, however, that this efficiency is obtained by exploiting the specific nature of input lists on which sorting by distribution counting works, in addition to trading space for time.

Input Enhancement in String Matching- Harspool's algorithm.

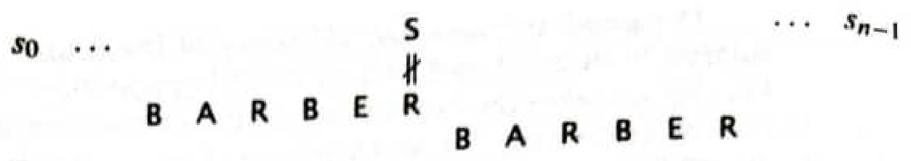
Horspool's Algorithm

Consider, as an example, searching for the pattern BARBER in some text:

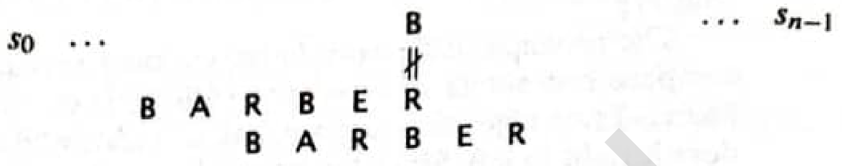
$$\begin{array}{cccccccccccc} s_0 & \dots & & & & & & c & \dots & s_{n-1} \\ & & \text{B} & \text{A} & \text{R} & \text{B} & \text{E} & \text{R} & & \end{array}$$

Starting with the last R of the pattern and moving right to left, we compare the corresponding pairs of characters in the pattern and the text. If all the pattern's characters match successfully, a matching substring is found. (Then the search can be either stopped altogether or continued if another occurrence of the same pattern is desired.) If, however, we encounter a mismatch, we need to shift the pattern to the right. Clearly, we would like to make as large a shift as possible without risking the possibility of missing a matching substring in the text. Horspool's algorithm determines the size of such a shift by looking at the character c of the text that was aligned against the last character of the pattern. In general, the following four possibilities can occur.

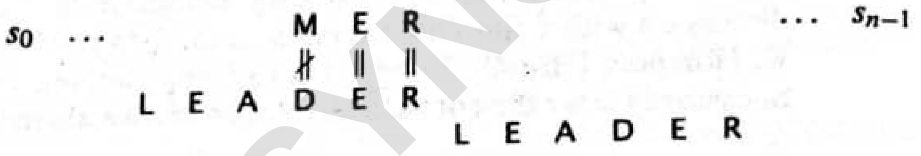
Case 1 If there are no c's in the pattern—e.g., c is letter S in our example—we can safely shift the pattern by its entire length (if we shift less, some character of the pattern would be aligned against the text's character c that is known not to be in the pattern):



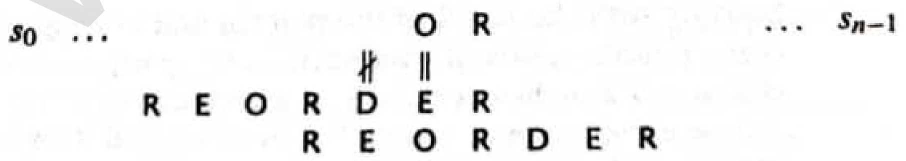
Case 2 If there are occurrences of character c in the pattern but it is not the last one there—e.g., c is letter B in our example—the shift should align the rightmost occurrence of c in the pattern with the c in the text:



Case 3 If c happens to be the last character in the pattern but there are no c's among its other $m - 1$ characters, the shift should be similar to that of Case 1: the pattern should be shifted by the entire pattern's length m, e.g.,



Case 4 Finally, if c happens to be the last character in the pattern and there are other c's among its first $m - 1$ characters, the shift should be similar to that of Case 2: the rightmost occurrence of c among the first $m - 1$ characters in the pattern should be aligned with the text's c, e.g.,



These examples clearly demonstrate that right-to-left character comparisons can lead to farther shifts of the pattern than the shifts by only one position always made by the brute-force algorithm. However, if such an algorithm had to check all the characters of the pattern on every trial, it would lose much of this superiority. Fortunately, the idea of input enhancement makes repetitive comparisons unnecessary. We can precompute shift sizes and store them in a table. The table will be indexed by all possible characters that can be encountered in a text, including, for natural language texts, the space, punctuation symbols, and other special characters. (Note that no other information about the text in which eventual searching will be done is required.) The table's entries will indicate the shift sizes computed by the formula

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern} \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise} \end{cases} \tag{7.1}$$

For example, for the pattern *BARBER*, all the table's entries will be equal to 6, except for the entries for E, B, R, and A, which will be 1, 2, 3, and 4, respectively.

Here is a simple algorithm for computing the shift table entries. Initialize all the entries to the pattern's length m and scan the pattern left to right repeating the following step $m-1$ times: for the jth character of the pattern ($0 \le j \le m-2$), overwrite its entry in the table with $m-1-j$, which is the character's distance to the right end of the pattern. Note that since the algorithm scans the pattern from left to right, the last overwrite will happen for a character's rightmost occurrence—exactly as we would like it to be.

```
ALGORITHM  ShiftTable(P[0..m − 1])
    //Fills the shift table used by Horspool's and Boyer-Moore algorithms
    //Input: Pattern P[0..m − 1] and an alphabet of possible characters
    //Output: Table[0..size − 1] indexed by the alphabet's characters and
    //    filled with shift sizes computed by formula (7.1)
    initialize all the elements of Table with m
    for j ← 0 to m − 2 do Table[P[j]] ← m − 1 − j
    return Table
```

Now, we can summarize the algorithm as follows.

Horspool's algorithm

Step 1 For a given pattern of length m and the alphabet used in both the pattern and text, construct the shift table as described above.

Step 2 Align the pattern against the beginning of the text.

Step 3 Repeat the following until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and text until either all m characters are matched (then stop) or a mismatching pair is encountered. In the latter case, retrieve the entry $t(c)$ from the c's column of the shift table where c is the text's character currently aligned against the last character of the pattern, and shift the pattern by $t(c)$ characters to the right along the text.

Here is a pseudocode of Horspool's algorithm.

```
ALGORITHM  HorspoolMatching(P[0..m − 1], T[0..n − 1])
    //Implements Horspool's algorithm for string matching
    //Input: Pattern P[0..m − 1] and text T[0..n − 1]
    //Output: The index of the left end of the first matching substring
    //        or −1 if there are no matches
    ShiftTable(P[0..m − 1])     //generate Table of shifts
    i ← m − 1                   //position of the pattern's right end
    while i ≤ n − 1 do
        k ← 0                   //number of matched characters
        while k ≤ m − 1 and P[m − 1 − k] = T[i − k] do
            k ← k + 1
        if k = m
            return i − m + 1
        else i ← i+Table[T[i]]
    return −1
```

EXAMPLE As an example of a complete application of Horspool's algorithm, consider searching for the pattern BARBER in a text that comprises English letters and spaces (denoted by underscores). The shift table, as we mentioned, is filled as follows:

character c	A	B	C	D	E	F	...	R	...	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

The actual search in a particular text proceeds as follows:

```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R                 B A R B E R
        B A R B E R             B A R B E R
          B A R B E R                 B A R B E R
```

■

A simple example can demonstrate that the worst-case efficiency of Horspool's algorithm is in $\Theta(nm)$ (Problem 4 in the exercises). But for random texts, it is in $\Theta(n)$, and, though in the same efficiency class, Horspool's algorithm is obviously faster on average than the brute-force algorithm. In fact, as mentioned, it is often at least as efficient as its more sophisticated predecessor discovered by R. Boyer and J. Moore.