Module-4 : Dynamic Programming

Contents

1. Introduction to Dynamic Programming

Dynamic programming is a te chnique for solving problems with **overlap ping subproblems**. Typically, these subproblems arise from a recurrence relating a given pr oblem's solution to solutions of its smaller subpro blems. Rather than solving overlapping sub problems 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 c an also be used when the solution to a problem can be viewed as the result of **sequence of d ecisions**. *[FROM T2].* Here are some examples.

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

Example 2 [illegible] x_1, x_2, an of length 30, 20,
erging x_1 a noves. Merging
ires anothe mber of record
e the three nstead, we first
; 30 moves) oves), the total
only 90. H attern is faster

tion 4.7. tion 4.7. An o
merged at merged at each
cide which cide which pair
third, and third, and so o

Example 3 [illegible] path] One way to find a shortest path from vertex
ed graph G is to decide which vertex should be the
ne third, which the fourth, and so on, until vertex j
sequence of decisions is one that results in a path of
□

Example 4

Example 5.4 [Shortest path] Supp **Example 5.4**
vertex i to vertex j. Let A_i be the v vertex i to vert
the vertices in A_i should be the seco the vertices in A
to make a decision at this time and to make a decis
to an optimal sequence can be mad to an optimal s
a shortest path from vertex i to all a shortest path
correc ision

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

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

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

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 applic The recursive application of the The recursiv
rence equation of type rence equation of type (5.3). Dyna rence equation
recurrence to obtain a recurrence to obtain a solution to t recurrence to ol
rence (5.3) can be solv rence (5.3) can be solved using the rence (5.3) can
$g_n(y) = -\infty$ for $y < 0$. $g_n(y) = -\infty$ for $y < 0$. From $g_n(y)$, $g_n(y) = -\infty$ for
$i = n - 1$. Then, using $i = n - 1$. Then, using $g_{n-1}(y)$, on $i = n - 1$. The
way, one can determine way, one can determine $g_1(y)$ and fi way, one can de

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.

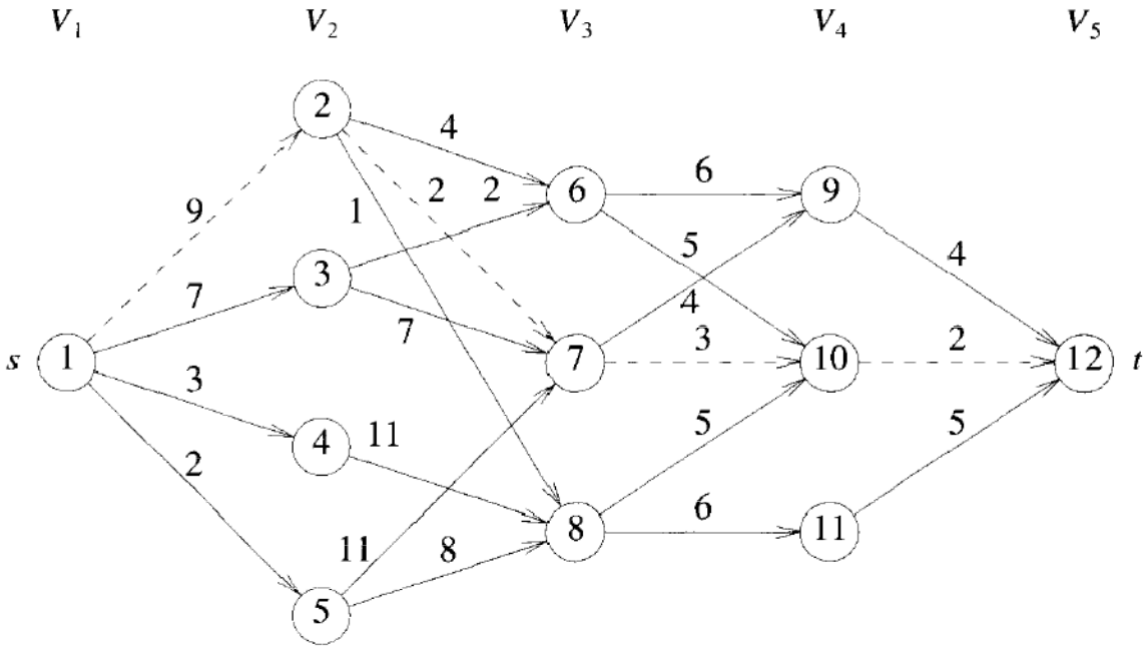


Figure: Five stage graph

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

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

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 ⟨i, j⟩. 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 ⟨j, r⟩ 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

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

$bcost(3, 7) = 11$ $\qquad$ $bcost(4, 10) = 14$

$bcost(3, 8) = 10$ $\qquad$ $bcost(4, 11) = 16$

$bcost(4, 9) = 15$ $\qquad$ $bcost(5, 12) = 16$

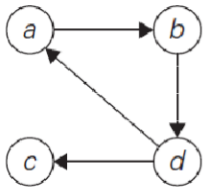
Algorithm 5.2 Multistage graph pseudocode corresponding to backward approach

```
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 ⟨r, j⟩ 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 cl osure** 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 i^{th} row and the j^{th} column is 1 if there exists a nontrivial path (i.e., directed path of a positive length) from the i^{th} vertex to the j^{th} vertex; otherwise, t^{ij} is 0.

Example: An example of a d igraph, 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 d epthfirst search or breadth-first search. Performi ng either traversal starting at the i^{th} vertex gives the information about the vertices reachable from it and hence the columns that contain 1 's in the i^{th} row of the transitive closure. Thus, d oing such a traversal for every vertex as a starting point yields the transitive closure in its ent irety.

Since this method traverses t he same digraph several times, we can use a better algorithm called **Warshall's algorithm**. Warshall's algorithm constructs the transit ive 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 pa ths in the digraph. Specifically, the element in the i^{th} row and j^{th} 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 p ositive length from the i^{th} vertex to the j^{th} 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 v ertices 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. Th e last matrix in the series, $R^{(n)}$, reflects paths that can use all n vertices of the digraph as inte rmediate 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 intermedi ate 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 k^{th} vertex. Then this path from v_i to v_j has intermediate vertices numbered not higher than k−1. i.e. r

   1

2. The second possibility is that path (*) **does contain** the k^{th} ver tex v_k among the intermediate vertices. Then path (*) can be rewritten as;

   v_i, vertices n umbered ≤ k − 1, v_k, vertices numbered ≤ k − 1, v_j .

   i.e r 1 and r

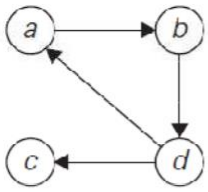
   1

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

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} \quad \text{or} \quad \left(r_{ik}^{(k-1)} \text{ and } r_{kj}^{(k-1)}\right)$$

The Warshall's algorithm wor ks based on the above formula.

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



$R^{(0)}$ =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

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 $R^{(1)}$.

$R^{(1)}$ =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	**1**	1	0

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 $R^{(2)}$.

$R^{(2)}$ =

	a	b	c	d
a	0	1	0	**1**
b	0	0	0	1
c	0	0	0	0
d	1	1	1	**1**

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 $R^{(3)}$.

$R^{(3)}$ =

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

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 $R^{(4)}$.

$R^{(4)}$ =

	a	b	c	d
a	**1**	1	**1**	1
b	**1**	**1**	**1**	1
c	0	0	0	0
d	1	1	1	1

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) ← A
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            R^(k)[i, j] ← 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 $\Theta(n^3)$. W e can make the algorithm to run faster by tr eating matrix rows as bit strings and employ the bitwise or operation available in most modern computer languages.

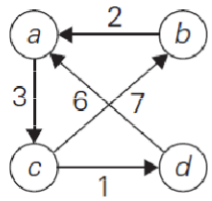
Space efficiency: Although separate matrices for recording intermediate results of the algorithm are used, that can b e avoided.

3. All Pairs Shortest Pa ths 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 s hortest paths - from each vertex to all other vertice s.

Applications: Solution to this problem finds applications in communications, transportation networks, and operations res earch. Among recent applications of the all-pairs shortest-path problem is pre-computing dis tances for motion planning in computer game s.

We store the lengths of shor test paths in an n x n matrix D called the d istance matrix: the element d_{ij} in the i^{th} row and the j^{th} column of this matrix indicates the le ngth of the shortest path from the i^{th} vertex to the j^{th} vertex.



(a) Digraph.

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

(b) Its weight matrix.

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

(c) Its d istance matrix

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

Floyd's algorithm computes t he 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 in the i^{th} ro w and the j^{th} column of matrix $D^{(k)}$ (i, j = 1, 2, . . . , n, k = 0, 1, . . . , n) is equal to the length o f the shortest path among all paths from the i^{th} vertex to the j^{th} 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 m atrix $D^{(k)}$ from its immediate predecessor $D^{(k-1)}$

If

1, then it means that there is a path;

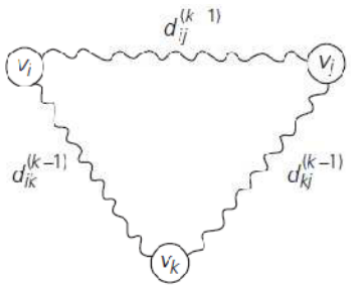
vi, a list of intermediate vertices each numbered not higher than k, vj .

We can partition all such paths into two disjoint subsets: those that do not use the k^{th} vertex v_k as intermediate and those that do.

i. Since the paths of the first subset have their intermediate vertices n umbered not higher than k − 1, the shortest o f them is, by definition of our matrices, of length

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 sym bolically in Figure, which shows the underlying idea of Floyd's algorithm.



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

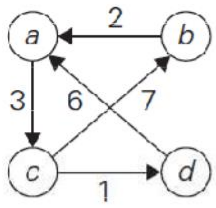
$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \quad \text{for } k \geq 1,\ \ d_{ij}^{(0)} = w_{ij}.$$

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.



$D^{(0)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

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

$D^{(1)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	**5**	∞
c	∞	7	0	1
d	6	∞	**9**	0

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*).

$D^{(2)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	**9**	7	0	1
d	6	∞	9	0

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*).

$D^{(3)} =$

	a	b	c	d
a	0	**10**	3	**4**
b	2	0	5	**6**
c	9	7	0	1
d	6	**16**	9	0

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*).

$D^{(4)} =$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	**7**	7	0	1
d	6	16	9	0

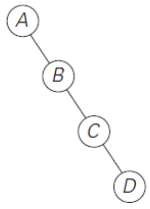
Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e., a, *b*, *c*, and *d* (note a new shortest path from *c* to *a*).

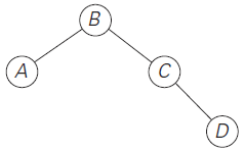
4. Optimal Binary Search Trees

A binary search tree is one of the most important data structures in compu ter science. One of its principal applications is to implement a dictionary, a set of elements with the operations of searching, insertion, and deletion.

If probabilities of searching for elements of a set are known e.g., from accumulated data about past searches it is natu ral to pose a question about an optimal bin ary search tree for which the average number of comparisons in a search is the smallest possi ble.

As an example, consider four keys A, B, C, and D to be searched for with probabilities 0.1, 0.2, 0.4, and 0.3, respectively. The figure depicts two out of 14 possible binary search t rees containing these keys.





The average number of comp arisons in a successful search in the first of these trees is 0.1 * 1+ 0.2 * 2 + 0.4 * 3+ 0.3 * 4 = 2.9, and for the second one it is 0.1 * 2 + 0.2 * 1+ 0.4 * 2 + 0.3 * 3= 2.1. Neither of thes e two trees is, in fact, optimal.

For our tiny example, we could find the optimal tree by generating all 14 binary search trees with these keys. As a general algorithm, this exhaustive-search approac h is unrealistic: the total number of binary search trees with n keys is equal to the nth **Catalan** number,

$$c(n) = \frac{1}{n+1}\binom{2n}{n} \quad \text{for } n > 0, \quad c(0) = 1,$$

which grows to infinity as fast as $4^n / n^{1.5}$

So let $a_1, \ldots, a_n$ be distinct k eys ordered from the smallest to the largest a nd let $p_1, \ldots, p_n$ be the probabilities of searching for them. Let C(i, j) be the smallest a verage number of comparisons made in a succes sful search in a binary search tree T_i^j made u p of keys $a_i, \ldots, a_j$, where i, j are some integer ind ices, $1 \le i \le j \le n$.

Following the classic dynami c programming approach, we will find values of C(i, j) for all smaller instances of the pro blem, although we are interested just in C(1, n). To derive a recurrence underlying a dyna mic programming algorithm, we will consid er all possible ways to choose a root a_k among the keys $a_i, \ldots, a_j$. For such a binary search tree (Figure 8.8), the root contains key ak, the left subtree T_i^{k-1} contains keys $a_i, \ldots, a_{k-1}$ opti mally arranged, and the right subtree T^j_{k+1}contain s keys $a_{k+1}, \ldots, a_j$ also optimally arranged. (Note how we are taking advantage of the principle of optimality here.)

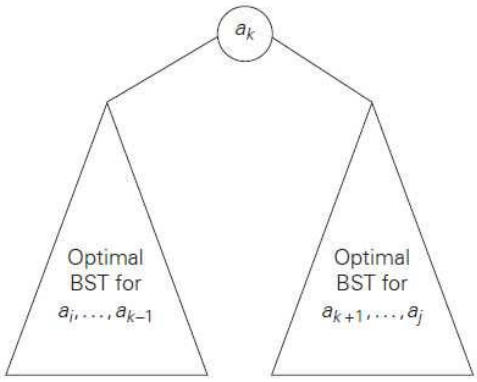


FIGURE 8.8 Binary search tree (BST) with root a_k and two optimal binary search subtrees T_i^{k-1} and T_{k+1}^j.

If we count tree levels starting with 1 to make the comparison numbers eq ual the keys' levels, the following recurrence relat ion is obtained:

$$C(i, j) = \min_{i \le k \le j} \{p_k \cdot 1 + \sum_{s=i}^{k-1} p_s \cdot (\text{level of } a_s \text{ in } T_i^{k-1} + 1) + \sum_{s=k+1}^{j} p_s \cdot (\text{level of } a_s \text{ in } T_{k+1}^j + 1)\}$$

$$= \min_{i \le k \le j} \{ \sum_{s=i}^{k-1} p_s \cdot \text{level of } a_s \text{ in } T_i^{k-1} + \sum_{s=k+1}^{j} p_s \cdot \text{level of } a_s \text{ in } T_{k+1}^{j} + \sum_{s=i}^{j} p_s \}$$

$$= \min_{i \le k \le j} \{C(i, k-1) + C(k+1, j)\} + \sum_{s=i}^{j} p_s.$$

Thus, we have the recurrence

$$C(i, j) = \min_{i \le k \le j} \{C(i, k-1) + C(k+1, j)\} + \sum_{s=i}^{j} p_s \quad \text{for } 1 \le i \le j \le n. \quad \textbf{(8.8)}$$

We assume in formula (8.8) that $C(i, i-1) = 0$ for $1 \le i \le n+1$, which can be interpreted as the number of comparisons in the empty tree. Note that this formula implies that

$$C(i, i) = p_i \quad \text{for } 1 \le i \le n,$$

as it should be for a one-node binary search tree containing a_i.

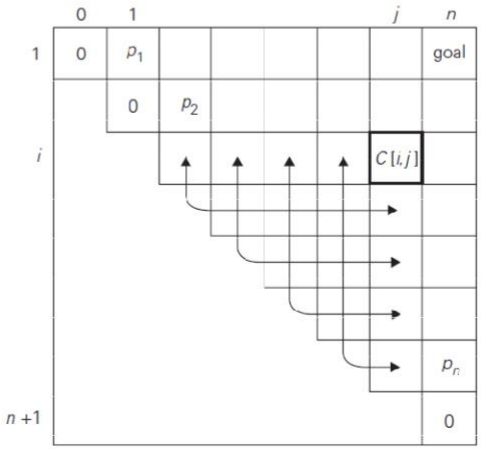


FIGURE 8.9 lable of the cynamic programming algorithm for constructing an optimal binary search tree.

The two-dimensional table in Figure 8.9 shows the values needed for co mputing C(i, j) by formula (8.8): they are in row i and the columns to the left of column j and in column j and the rows below row i. The arrows point to the pairs of entries whose su ms are computed in order to find the smallest one to be recorded as the value of C(i, j). This suggests filling the table along its diagonals, star ting with all zeros on the main diagonal and given probabilities pi, 1≤ i ≤ n, right above it and moving toward the upper right corner.

The algorithm we just sketched computes C(1, n)—the average number of comparisons for successful searches in the op timal binary tree. If we also want to get the optimal tree itself, we need to maintain another two-dimensional table to record the value of k for which the minimum in (8.8) is achieved. The table has the same shape as the table in Figure 8.9 and is filled in the same manner, sta rting with entries R(i, i) = i for 1≤ i ≤ n. When the table is filled, its entries indicate indices of the roots of the optimal subtrees, which m akes it possible to reconstruct an optimal tree for the entire set given.

Example: Let us illustrate the algorithm by applying it to the four-key set we used at the beginning of this section:

key	A	B	C	D
probability	0.1	0.2	0.4	0.3

The initial tables look like thi s:

main table

	0	1	2	3	4
1	0	0.1			
2		0	0.2		
3			0	0.4	
4				0	0.3
5					0

root table

	0	1	2	3	4
1		1			
2			2		
3				3	
4					4
5					

Let us compute C(1, 2):

$$C(1,2) = \min \left\{ \begin{array}{ll} k=1: & C(1,0) + C(2,2) + \sum_{s=1}^{2} p_s = 0 + 0.2 + 0.3 = 0.5 \\ k=2: & C(1,1) + C(3,2) + \sum_{s=1}^{2} p_s = 0.1 + 0 + 0.3 = 0.4 \end{array} \right\}$$
$$= 0.4.$$

Thus, out of two possible binary trees containing the first two keys, A and B, the root of the optimal tree has index 2 (i.e ., it contains B), and the average number of comparisons in a successful search in this tree is 0.4. On finishing the computations we get the following final tables:

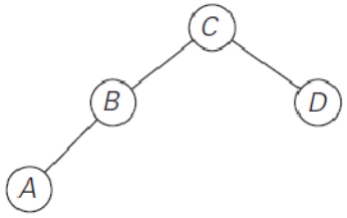
main table

	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2		0	0.2	0.8	1.4
3			0	0.4	1.0
4				0	0.3
5					0

root table

	0	1	2	3	4
1		1	2	3	3
2			2	3	3
3				3	3
4					4
5					

Thus, the average number of key comparisons in the optimal tree is equal to 1.7. Since R(1, 4) = 3, the root of the optimal tree contains the third key, i.e., C. Its left subtree is made up of keys A and B, and its right subtree contains just key D. To find the specifi c structure of these subtrees, we find first their ro ots by consulting the root table again as follows. Since R(1, 2) = 2, the root of the optimal tre e containing A and B is B, with A being its left child (and the root of the one-node tree: R(1, 1) = 1). Since R(4, 4) = 4, the root of this one-node optimal tree is its only key D. Figure g iven below presents the optimal tree in its en tirety.



Here is Pseudocode of the dyn amic programming algorithm.

ALGORITHM *OptimalBST*$(P[1..n])$

```
//Finds an optimal binary search tree by dynamic programming
//Input: An array P[1..n] of search probabilities for a sorted list of n keys
//Output: Average number of comparisons in successful searches in the
//        optimal BST and table R of subtrees' roots in the optimal BST
for i ← 1 to n do
    C[i, i − 1] ← 0
    C[i, i] ← P[i]
    R[i, i] ← i
C[n + 1, n] ← 0
for d ← 1 to n − 1 do    //diagonal count
    for i ← 1 to n − d do
        j ← i + d
        minval ← ∞
        for k ← i to j do
            if C[i, k − 1] + C[k + 1, j] < minval
                minval ← C[i, k − 1] + C[k + 1, j]; kmin ← k
        R[i, j] ← kmin
        sum ← P[i]; for s ← i + 1 to j do sum ← sum + P[s]
        C[i, j] ← minval + sum
return C[1, n], R
```

5. Knapsack problem

We start this section with designing a dynamic programming algorithm for the knapsack problem: given n items of kn own weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ and a knapsack of capacity W, find the most v aluable subset of the items that fit into the k napsack.

To design a dynamic progra mming algorithm, we need to derive a recurrence relation that expresses a solution to an i nstance of the knapsack problem in terms of solutions to its smaller subinstances.

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 knaps ack capacity j, $1 \le j \le W$. Let F(i, j) be the value of an optimal solution to this instance. We c an divide all the subsets of the first i items t hat fit the knapsack of capacity j into two categor ies: those that do not include the i^{th} item and those that do. Note the following:

i. Among the subsets th at do not include the i^{th} item, the value of a n optimal subset is, by definition, $F(i - 1, j)$.

ii. Among the subsets tha t do include the i^{th} item (hence, $j - w_i \ge 0$), a n optimal subset is made up of this item and an optimal subset of the first $i-1$ item s 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 give n items that fit into the knapsack of capacity W, and an optimal subset itself.

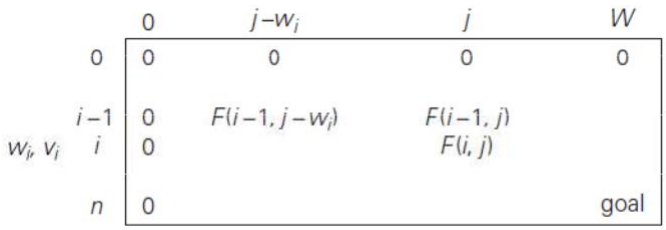


Table for solving the knapsack problem by dynamic programming.

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

item	weight	value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

capacity $W = 5$.

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 backtracing the c omputations 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 n ot be in an optimal subset. Since F(2, 3) > F(1, 3), item 2 is a part of an optimal selection, w hich leaves element F(1, 3 − 1) to specify its rem aining composition. Similarly, since F(1, 2) > F(0, 2), item 1 is the final part of the optimal so lution {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 subprobl ems more than once and hence is very inefficient.

The classic dynamic program ming approach, on the other hand, works b ottom up: it fills a table with solutions to all sm aller subproblems, but each of them is solved only once. An unsatisfying aspect of this ap proach is that solutions to some of these s maller subproblems are often not necessary for getting a solution to the problem given. Since this drawback is not present in the top-down appro ach, it is natural to try to combine the strengths of the top-down and bottom-up approaches. T he goal is to get a method that solves only s ubproblems 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 h ave been used by a bottom-up dynamic programming algorithm. Initially, all the table's entries are initialized with a special "null" symbol t o indicate that they have not yet been calculated. Thereafter, whenever a new value needs to be calculated, the method checks the correspond ing 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 w hose result is then recorded in the table.

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

Algorithm MFKnaps ack(i, j)

//Implements the memory function method for the knapsack problem

//**Input:** A nonnegativ e integer i indicating the number of the first items being considered and a nonnegative integer j indicating the knap sack capacity

//**Output:** The value of an optimal feasible subset of the first i items

//**Note:** Uses as global variables input arrays Weights[1..n], V alue s[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 ← MFKnapsack(i − 1, j)
    else
        value ← max(MFKnapsack(i − 1, j),
                    Values[i] + MFKnapsack(i − 1, j − Weights[i]))
    F[i, j] ← 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 b elow gives the results. Only 11 out of 20 nontrivial values (i.e., not those in row 0 or in colum n 0) have been computed. Just one nontrivi al entry, V (1, 2), is retrieved rather than being re computed. 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 prob lem, because its time efficiency class is the same as that of the bottom-up algorithm

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

Problem definition

Single source shortest path - Given a graph and a source vertex ***s*** in graph, find shortest paths from ***s*** to all vertices in the gi ven graph. The graph may contain negative w eight edges.

Note that we have discussed Dijkstra's algorithm for single source sho rtest path problem. Dijksra's algorithm is a Gre edy 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 cal culates the shortest distances for the shortes t paths which have at-most one edge in the path. Then, it calculates shortest paths with at-most 2 edges, and so on. After the i^{th} iteration of o uter loop, the shortest paths with at most i e dges are calculated. There can be maximum |V| – 1 edges in any simple path, that is why the o uter loop runs |v| – 1 times. The idea is, assumin g 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

source vertex v
ıtains at most ℓ
lier, when there
: shortest paths
he length of an

:an be done us-
ɛe the following

1. If the sho
more thar

> 1, edges has
from v to some
, to j has $k-1$
h that the edge
: interested in a
$u]$ is the correct

esult in the following recurrence for $dist$:

$$\text{ı } \{dist^{k-1}[u],\ \min_i \{dist^{k-1}[i] + cost[i,u]\}\}$$

for $k = 2, 3, \ldots,$

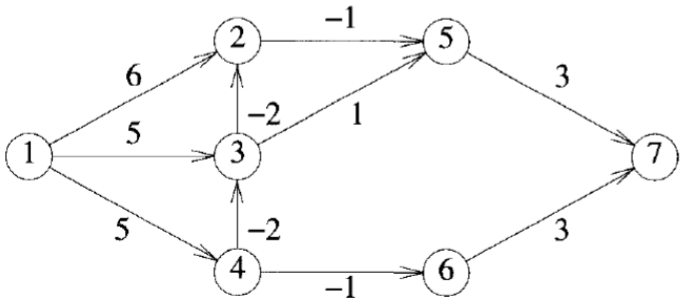
Bellman-Ford algorithm to compute shortest path

```
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 ≠ v and u has
                at least one incoming edge do
            for each ⟨i, u⟩ 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

7. 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 dis In the following discussion we s In the follov
a tour to be a simple a tour to be a simple path that st a tour to be a
consists of an edge $\langle 1, k$ consists of an edge $\langle 1, k \rangle$ for some k consists of an e
vertex 1. The path fro vertex 1. The path from vertex k t vertex 1. The
$V - \{1, k\}$ exactly once. $V - \{1, k\}$ exactly once. It is easy to $V - \{1, k\}$ exact
path from k to 1 must path from k to 1 must be a shortes path from k to
in $V - \{1, k\}$. Hence, t in $V - \{1, k\}$. Hence, the principle in $V - \{1, k\}$.
length of a shortest pat length of a shortest path starting a length of a sho
S, and terminating at 1 S, and terminating at vertex 1. Th S, and termina
an optimal salesperson an optimal salesperson tour. From t an optimal sale

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

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

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

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

$g(i, \phi) = c_{i1}$, $1 \le i \le n$. $g(i, \phi) = c_{i1}$, $1 \le i \le n$. Hence, we $g(i, \phi) = c_{i1}$, 1
of size 1. Then we can of size 1. Then we can obtain $g(i, S$ of size 1. Then
$|S| < n - 1$, the values $|S| < n - 1$, the values of i and S f $|S| < n - 1$, th
$i \neq 1$, $1 \notin S$, and $i \notin S$. $i \neq 1$, $1 \notin S$, and $i \notin S$. $i \neq 1$, $1 \notin S$, an

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) (b)

Figure 5.21 Directed graph and edge length matrix c

$c_{41} = 8$. Using (5.21), we obtain

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

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

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

Finally, from (5.20) we obtain

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

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

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

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.

8. Reliability design

ic programming
ion. The prob-
es connected in
hat is, r_i is the
eliability of the
ry reliable (the
ay not be very
ıen $\Pi r_i = .904$.
of the same de-
use of switching
ıny given group
device at each

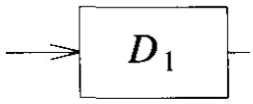


Figure 5.19 *r***Figure 5.19** n devices D_i, $1 \leq i \leq$**Figure**

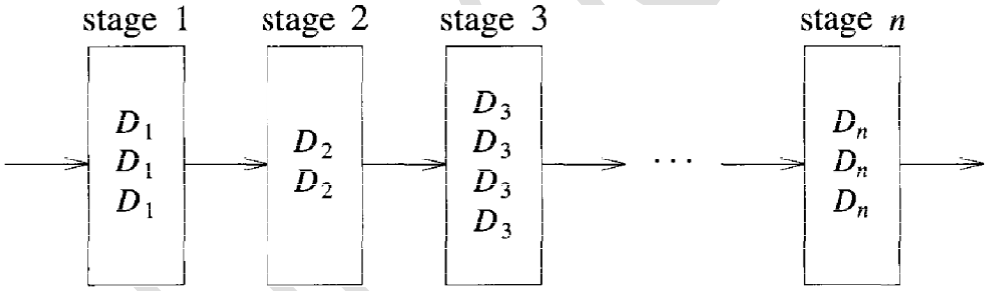


Figure 5.20 Mult**Figure 5.20** Multiple devices con**Figure 5.2**

If stage i contains n If stage i contains m_i copies of (If stage i co:
m_i have a malfunction i m_i have a malfunction is $(1 - r_i)^{m_i}$. m_i have a malfı
ability becomes
little less than
ot fully reliable.
dependent (e.g.,
iability of stage
ble that $\phi_i(m_i)$
f the system of

stages is $\Pi_{1 \leq i \leq n} \phi_i(m_i)$.

Our problem is to u Our problem is to use device du Our problem
maximization is to be maximization is to be carried out maximization i
cost of each unit of de cost of each unit of device i and l cost of each un
the system being desig the system being designed. We wi the system bei
problem: problem: problem:

$$\text{maximize } \Pi_{1\le i\le n}\ \phi_i(m_i)$$

$$\text{subject to } \sum_{1\le i\le n} c_i m_i \le c$$

$$m_i \ge 1 \text{ and integer, } 1 \le i \le n$$

A dynamic programming solution can be obtained in a manner similar to that used for the knapsack problem. Since, we can assume each $c_i > 0$, each m_i must be in the range $1 \le m_i \le u_i$, where

$$u_i = \left\lfloor (c + c_i - \sum_{1}^{n} c_j)/c_i \right\rfloor$$

The upper bound u_i fol The upper bound u_i follows from th The upper bou
solution $m_1, m_2, \ldots, m_r$ solution $m_1, m_2, \ldots, m_n$ is the result solution m_1, m_2
for each m_i. Let $f_i(x)$ re for each m_i. Let $f_i(x)$ represent the for each m_i. Le
to the constraints $\sum_{1\le j}$ to the constraints $\sum_{1\le j\le i} c_j m_j \le x$ to the constrain
value of an optimal solu value of an optimal solution is $f_n(c)$. value of an opti
choose m_n from $\{1, 2, 3$ choose m_n from $\{1, 2, 3, \ldots, u_n\}$. O choose m_n from
remaining decisions mu remaining decisions must be such as remaining decis
an optimal way. The pr an optimal way. The principal of op an optimal way

$$f_n(c) = \max_{1\le m_n\le u_n} \{\phi_n(m_n) f_{n-1}(c - c_n m_n)\} \tag{5.18}$$

For any $f_i(x)$, $i \ge 1$, this equation generalizes to

$$f_i(x) = \max_{1\le m_i\le u_i} \{\phi_i(m_i) f_{i-1}(x - c_i m_i)\} \tag{5.19}$$

Clearly, $f_0(x) = 1$ fo Clearly, $f_0(x) = 1$ for all x, $0 \le a$ Clearly, $f_0(x$
an approach similar to an approach similar to that used fo an approach sir
of tuples of the form $(f$ of tuples of the form (f, x), where f of tuples of the
each different x that res each different x that results from a s each different x
The dominance rule $(f_1$ The dominance rule (f_1, x_1) domina The dominance
for this problem too. H for this problem too. Hence, domin for this problem

Example 5.25 D_1, D_2, and D_3 the system is to .9, .8 and .5 res in parallel, then $c_1 = 30$, $c_2 = 1$ and $u_3 = 3$.

We use S^i t may result fron $f(x) = f_i(x)$. B by trying out a together. Using $m_i = j$, we obt $S_1^2 = \{(.72, 45), (.792, 75)\}$; $S_2^2 = \{(.864, 60)\}$. Note that the tuple (.9504, 90) which comes from (.99, 60) has been eliminated from S_2^2 as this leaves only \$10. This is not enough to allow $m_3 = 1$. The set $S_3^2 = \{(.8928, 75)\}$. Combining, we get $S^2 = \{(.72, 45), (.864, 60), (.8928, 75)\}$ as the tuple (.792, 75) is dominated by (.864, 60). The set $S_1^3 = \{(.36, 65), (.432, 80), (.4464, 95)\}$, $S_2^3 = \{(.54, 85), (.648, 100)\}$, and $S_3^3 = \{(.63, 105)\}$. Combining, we get $S^3 = \{(.36, 65), (.432, 80), (.54, 85), (.648, 100)\}$.

The best design has a reliability of .648 and a cost of 100. Tracing back through the S^i's, we determine that $m_1 = 1, m_2 = 2$, and $m_3 = 2$. □

As in the case of the knapsack problem, a complete dynamic programming algorithm for the reliability problem will use heuristics to reduce the size of the S^i's. There is no need to retain any tuple (f, x) in S^i with x value greater that $c - \sum_{i \le j \le n} c_j$ as such a tuple will not leave adequate funds to complete the system. In addition, we can devise a simple heuristic to determine the best reliability obtainable by completing a tuple (f, x) in S^i. If this is less than a heuristically determined lower bound on the optimal system reliability, then (f, x) can be eliminated from S^i.