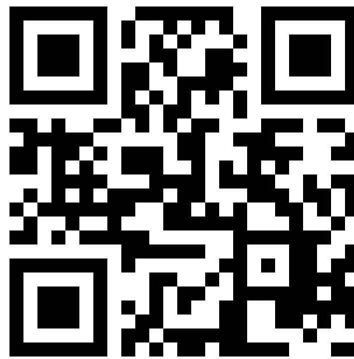
FUTURE VISION BIE

One Stop for All Study Materials & Lab Programs

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page



Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

COMPUTER ALGORITHMS

Ellis Horowitz
University of Southern California

Sartaj Sahni
University of Florida

Sanguthevar Rajasekaran
University of Florida

Computer Science Press
An imprint of W. H. Freeman and Company
New York

https://hemanthrajhemu.github.io

4.5.2 Kruskal's Algorithm 220
4.5.3 An Optimal Randomized Algorithm (*) 225
4.6 OPTIMAL STORAGE ON TAPES 229
4.7 OPTIMAL MERGE PATTERNS 234
4.8 SINGLE-SOURCE SHORTEST PATHS 241
4.9 REFERENCES AND READINGS 249
4.10 ADDITIONAL EXERCISES 250

5 DYNAMIC PROGRAMMING 253
5.1 THE GENERAL METHOD 253
5.2 MULTISTAGE GRAPHS 257
5.3 ALL PAIRS SHORTEST PATHS 265
5.4 SINGLE-SOURCE SHORTEST PATHS: . . . .
GENERAL WEIGHTS 270
5.5 OPTIMAL BINARY SEARCH TREES (*) 275
5.6 STRING EDITING 284
5.7 0/1-KNAPSACK 287
5.8 RELIABILITY DESIGN 295
5.9 THE TRAVELING SALESPERSON PROBLEM 298
5.10 FLOW SHOP SCHEDULING 301
5.11 REFERENCES AND READINGS 307
5.12 ADDITIONAL EXERCISES 308

6 BASIC TRAVERSAL AND SEARCH TECHNIQUES 313
6.1 TECHNIQUES FOR BINARY TREES 313
6.2 TECHNIQUES FOR GRAPHS 318
6.2.1 Breadth First Search and Traversal 320
6.2.2 Depth First Search and Traversal 323
6.3 CONNECTED COMPONENTS AND SPANNING TREES . 325
6.4 BICONNECTED COMPONENTS AND DFS 329
6.5 REFERENCES AND READINGS 338

7 BACKTRACKING 339
7.1 THE GENERAL METHOD 339
7.2 THE 8-QUEENS PROBLEM 353
7.3 SUM OF SUBSETS 357
7.4 GRAPH COLORING 360
7.5 HAMILTONIAN CYCLES 364
7.6 KNAPSACK PROBLEM 368

https://hemanthrajhemu.github.io

Chapter 5

DYNAMIC PROGRAMMING

5.1 THE GENERAL METHOD

Dynamic programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions. In earlier chapters we saw many problems that can be viewed this way. Here are some examples:

Example 5.1 [Knapsack] The solution to the knapsack problem (Section 4.2) can be viewed as 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 5.2 [Optimal merge patterns] This problem was discussed in Section 4.7. 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 5.3 [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. □

https://hemanthrajhemu.github.io

For some of the problems that may be viewed in this way, an optimal sequence of decisions can be found by making the decisions one at a time and never making an erroneous decision. This is true for all problems solvable by the greedy method. For many other problems, it is not possible to make stepwise decisions (based only on local information) in such a manner that the sequence of decisions made is optimal.

Example 5.4 [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 (see Section 4.8). □

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. □

https://hemanthrajhemu.github.io

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

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

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. □

Let S_0 be the initial problem state. Assume that n decisions d_i, $1 \le i \le n$, have to be made. Let $D_1 = \{r_1, r_2, \ldots, r_j\}$ be the set of possible decision values for d_1. Let S_i be the problem state following the choice of decision r_i, $1 \le i \le j$. Let Γ_i be an optimal sequence of decisions with respect to the problem state S_i. Then, when the principle of optimality holds, an optimal sequence of decisions with respect to S_0 is the best of the decision sequences r_i, Γ_i, $1 \le i \le j$.

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. □

https://hemanthrajhemu.github.io

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, $1 \leq j \leq n$, $y_1, \ldots, y_j$, and $y_{j+1}, \ldots, y_n$ must be optimal solutions to the problems KNAP$(1, j, \sum_{1 \leq i \leq j} w_i y_i)$ and KNAP$(j+1, n, m - \sum_{1 \leq i \leq j} w_i y_i)$ 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 $y \geq 0$ 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 $i = 0$.

Example 5.11 [0/1 knapsack] Consider the case in which $n = 3$, $w_1 = 2, w_2 = 3, w_3 = 4$, $p_1 = 1, p_2 = 2, p_3 = 5$, and $m = 6$. We have to compute $g_0(6)$. The value of $g_0(6) = \max \ \{g_1(6),\ g_1(4) + 1\}$.

In turn, $g_1(6) = \max \ \{g_2(6),\ g_2(3)+2\}$. But $g_2(6) = \max \ \{g_3(6),\ g_3(2)+5\} = \max \ \{0, 5\} = 5$. Also, $g_2(3) = \max \ \{g_3(3),\ g_3(3-4)+5\} = \max \ \{0, -\infty\} = 0$. Thus, $g_1(6) = \max \ \{5, 2\} = 5$.

Similarly, $g_1(4) = \max \ \{g_2(4),\ g_2(4-3)+2\}$. But $g_2(4) = \max \ \{g_3(4),\ g_3(4-4)+5\} = \max \ \{0, 5\} = 5$. The value of $g_2(1) = \max \ \{g_3(1),\ g_3(1-4)+5\} = \max \ \{0, -\infty\} = 0$. Thus, $g_1(4) = \max \ \{5, 0\} = 5$.

Therefore, $g_0(6) = \max \ \{5, 5+1\} = 6$. □

Example 5.12 [Shortest path] Let P_j be the set of vertices adjacent to vertex j (that is, $k \in P_j$ iff $\langle k, j \rangle \in E(G)$). For each $k \in P_j$, let Γ_k be a shortest i to k path. The principle of optimality holds and a shortest i to j path is the shortest of the paths $\{\Gamma_k, j | k \in P_j\}$.

To obtain this formulation, we started at vertex j and looked at the last decision made. The last decision was to use one of the edges $\langle k, j \rangle$, $k \in P_j$. In a sense, we are looking backward on the i to j path. □

Example 5.13 [0/1 knapsack] Looking backward on the sequence of decisions $x_1, x_2, \ldots, x_n$, we see that

$$f_j(y) = \max \ \{f_{j-1}(y),\ f_{j-1}(y - w_j) + p_j\} \tag{5.4}$$

where $f_j(y)$ is the value of an optimal solution to KNAP$(1, j, y)$.

https://hemanthrajhemu.github.io

The value of an optimal solution to KNAP$(1, n, m)$ is $f_n(m)$. Equation 5.4 can be solved by beginning with $f_0(y) = 0$ for all y, $y \geq 0$, and $f_0(y) = -\infty$, for all y, $y < 0$. From this, $f_1, f_2, \ldots, f_n$ can be successively obtained. □

The solution method outlined in Examples 5.12 and 5.13 may indicate that one has to look at all possible decision sequences to obtain an optimal decision sequence using dynamic programming. This is not the case. Because of the use of the principle of optimality, decision sequences containing subsequences that are suboptimal are *not* considered. Although the total number of different decision sequences is exponential in the number of decisions (if there are d choices for each of the n decisions to be made then there are d^n possible decision sequences), dynamic programming algorithms often have a polynomial complexity.

Another important feature of the dynamic programming approach is that optimal solutions to subproblems are retained so as to avoid recomputing their values. The use of these tabulated values makes it natural to recast the recursive equations into an iterative algorithm. Most of the dynamic programming algorithms in this chapter are expressed in this way.

The remaining sections of this chapter apply dynamic programming to a variety of problems. These examples should help you understand the method better and also realize the advantage of dynamic programming over explicitly enumerating all decision sequences.

EXERCISES

1. The principle of optimality does not hold for every problem whose solution can be viewed as the result of a sequence of decisions. Find two problems for which the principle does not hold. Explain why the principle does not hold for these problems.

2. For the graph of Figure 5.1, find the shortest path between the nodes 1 and 2. Use the recurrence relations derived in Examples 5.10 and 5.13.

5.2 MULTISTAGE GRAPHS

A multistage graph $G = (V, E)$ is a directed graph in which the vertices are partitioned into $k \geq 2$ disjoint sets V_i, $1 \leq i \leq k$. 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 $i, 1 \leq i < k$. 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

https://hemanthrajhemu.github.io

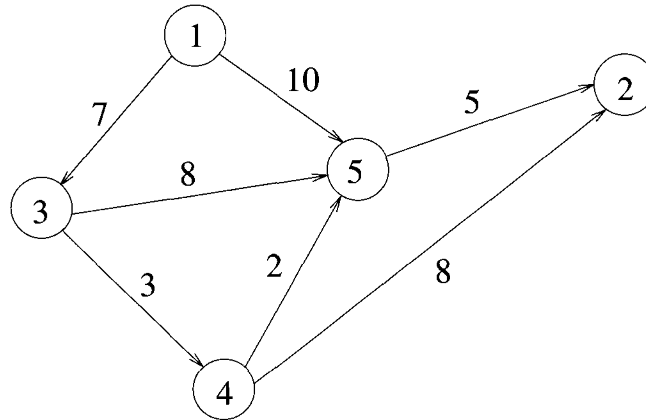


Figure 5.1 Graph for Exercise 2 (Section 5.1)

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.

Many problems can be formulated as multistage graph problems. We give only one example. Consider a resource allocation problem in which n units of resource are to be allocated to r projects. If j, $0 \leq j \leq n$, units of the resource are allocated to project i, then the resulting net profit is $N(i, j)$. The problem is to allocate the resource to the r projects in such a way as to maximize total net profit. This problem can be formulated as an $r+1$ stage graph problem as follows. Stage i, $1 \leq i \leq r$, represents project i. There are $n+1$ vertices $V(i, j)$, $0 \leq j \leq n$, associated with stage i, $2 \leq i \leq r$. Stages 1 and $r+1$ each have one vertex, $V(1, 0) = s$ and $V(r+1, n) = t$, respectively. Vertex $V(i, j)$, $2 \leq i \leq r$, represents the state in which a total of j units of resource have been allocated to projects $1, 2, \ldots, i-1$. The edges in G are of the form $\langle V(i, j), V(i+1, l)\rangle$ for all $j \leq l$ and $1 \leq i < r$. The edge $\langle V(i, j), V(i+1, l)\rangle$, $j \leq l$, is assigned a weight or cost of $N(i, l-j)$ and corresponds to allocating $l-j$ units of resource to project i, $1 \leq i < r$. In addition, G has edges of the type $\langle V(r, j), V(r+1, n)\rangle$. Each such edge is assigned a weight of $\max_{0 \leq p \leq n-j}\{N(r, p)\}$. The resulting graph for a three-project problem with $n = 4$ is shown in Figure 5.3. It should be easy to see that an optimal allocation of resources is defined by a maximum cost s to t path. This is easily converted into a minimum-cost problem by changing the sign of all the edge costs.

https://hemanthrajhemu.github.io

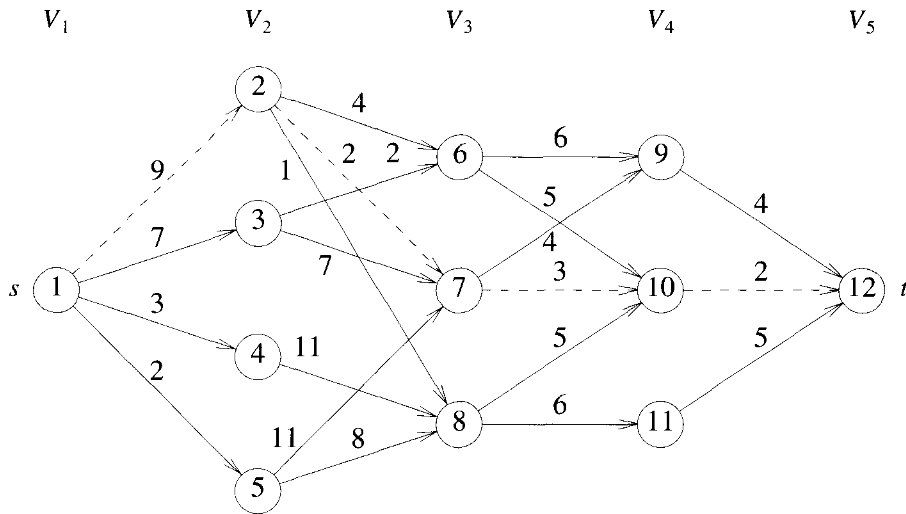


Figure 5.2 Five-stage graph

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

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

https://hemanthrajhemu.github.io

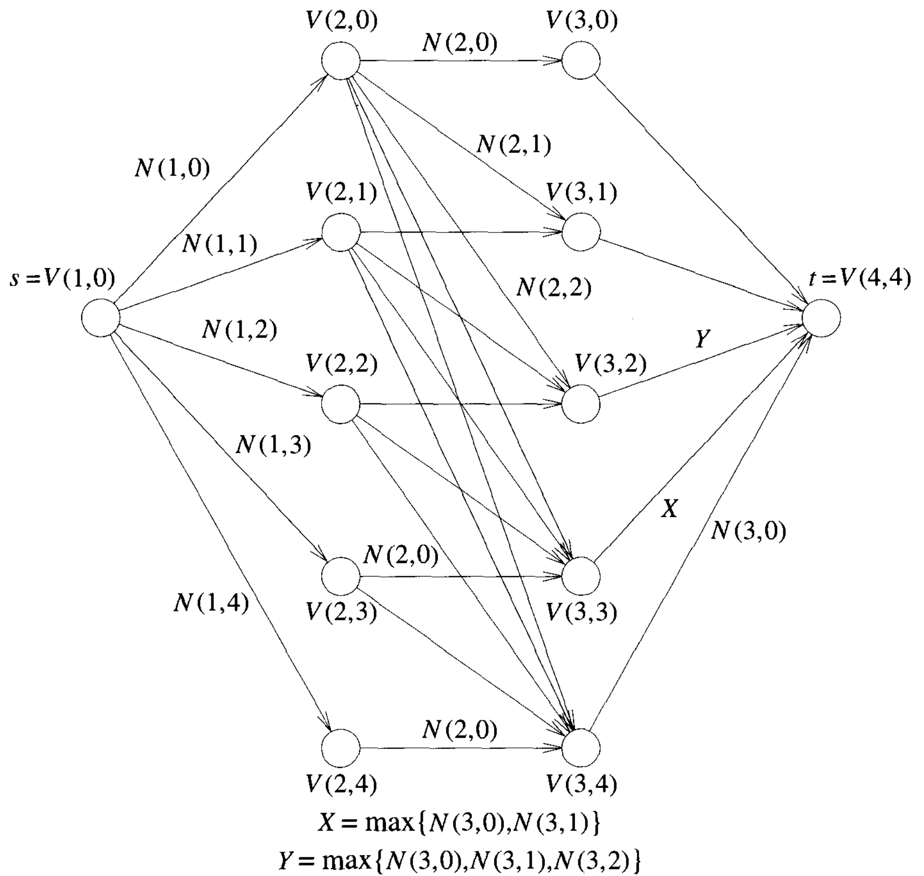


Figure 5.3 Four-stage graph corresponding to a three-project problem

https://hemanthrajhemu.github.io

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

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

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

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 $v_4 = d(3, d(2,\ d(1,1))) = d(3,7) = 10$.

Before writing an algorithm to solve (5.5) for a general k-stage graph, let us impose an ordering on the vertices in V. This ordering makes it easier to write the algorithm. We require that the n vertices in V are indexed 1 through n. Indices are assigned in order of stages. First, s is assigned index 1, then vertices in V_2 are assigned indices, then vertices from V_3, and so on. Vertex t has index n. Hence, indices assigned to vertices in V_{i+1} are bigger than those assigned to vertices in V_i (see Figure 5.2). As a result of this indexing scheme, $cost$ and d can be computed in the order $n-1, n-2, \ldots, 1$. The first subscript in $cost$, p, and d only identifies the stage number and is omitted in the algorithm. The resulting algorithm, in pseudocode, is FGraph (Algorithm 5.1).

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 $p[\ ]$.

https://hemanthrajhemu.github.io

```
1   Algorithm FGraph(G, k, n, p)
2   // The input is a k-stage graph G = (V, E) with n vertices
3   // indexed in order of stages. E is a set of edges and c[i, j]
4   // is the cost of ⟨i, j⟩. p[1 : k] is a minimum-cost path.
5   {
6       cost[n] := 0.0;
7       for j := n − 1 to 1 step −1 do
8       { // Compute cost[j].
9           Let r be a vertex such that ⟨j, r⟩ is an edge
10          of G and c[j, r] + cost[r] is minimum;
11          cost[j] := c[j, r] + cost[r];
12          d[j] := r;
13      }
14      // Find a minimum-cost path.
15      p[1] := 1; p[k] := n;
16      for j := 2 to k − 1 do p[j] := d[p[j − 1]];
17  }
```

Algorithm 5.1 Multistage graph pseudocode corresponding to the forward 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

$$\begin{aligned}
bcost(3,6) &= \min\ \{bcost(2,2) + c(2,6), bcost(2,3) + c(3,6)\} \\
&= \min\ \{9+4, 7+2\} \\
&= 9 \\
bcost(3,7) &= 11 \\
bcost(3,8) &= 10 \\
bcost(4,9) &= 15
\end{aligned}$$

https://hemanthrajhemu.github.io

$$\begin{aligned} bcost(4,10) &= 14 \\ bcost(4,11) &= 16 \\ bcost(5,12) &= 16 \end{aligned}$$

The corresponding algorithm, in pseudocode, to obtain a minimum-cost $s-t$ path is BGraph (Algorithm 5.2). The first subscript on *bcost*, p, and d are omitted for the same reasons as before. This algorithm has the same complexity as FGraph provided G is now represented by its inverse adjacency lists (i.e., for each vertex v we have a list of vertices w such that $\langle w, v\rangle \in E$).

```
1   Algorithm BGraph(G, k, n, p)
2   // Same function as FGraph
3   {
4       bcost[1] := 0.0;
5       for j := 2 to n do
6       { // Compute bcost[j].
7           Let r be such that ⟨r, j⟩ is an edge of
8           G and bcost[r] + c[r, j] is minimum;
9           bcost[j] := bcost[r] + c[r, j];
10          d[j] := r;
11      }
12      // Find a minimum-cost path.
13      p[1] := 1; p[k] := n;
14      for j := k - 1 to 2 do p[j] := d[p[j + 1]];
15  }
```

Algorithm 5.2 Multistage graph pseudocode corresponding to backward approach

It should be easy to see that both FGraph and BGraph work correctly even on a more generalized version of multistage graphs. In this generalization, the graph is permitted to have edges $\langle u, v\rangle$ such that $u \in V_i, v \in V_j$, and $i < j$.

Note: In the pseudocodes FGraph and BGraph, $bcost(i, j)$ is set to ∞ for any $\langle i, j\rangle \notin E$. When programming these pseudocodes, one could use the maximum allowable floating point number for ∞. If the weight of any such edge is added to some other costs, a floating point overflow might occur. Care should be taken to avoid such overflows.

https://hemanthrajhemu.github.io

EXERCISES

1. Find a minimum-cost path from s to t in the multistage graph of Figure 5.4. Do this first using the forward approach and then using the backward approach.

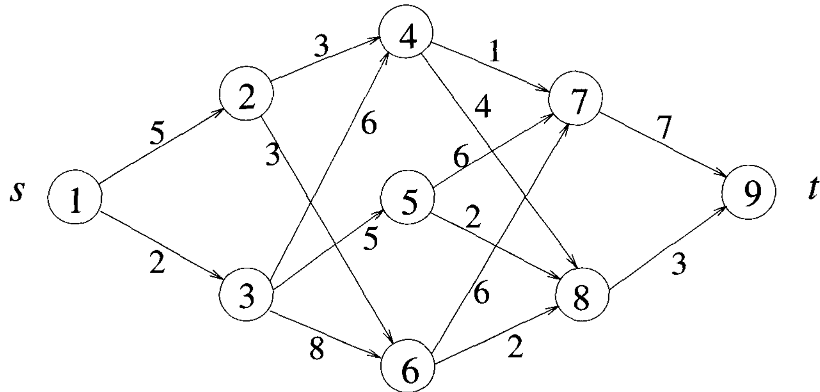


Figure 5.4 Multistage graph for Exercise 1

2. Refine Algorithm 5.1 into a program. Assume that G is represented by its adjacency lists. Test the correctness of your code using suitable graphs.

3. Program Algorithm 5.1. Assume that G is an array $G[1:e, 1:3]$. Each edge $\langle i, j \rangle$, $i < j$, of G is stored in $G[q]$, for some q and $G[q,1] = i$, $G[q,2] = j$, and $G[q,3] =$ cost of edge $\langle i, j \rangle$. Assume that $G[q,1] \leq G[q+1,1]$ for $1 \leq q < e$, where e is the number of edges in the multistage graph. Test the correctness of your function using suitable multistage graphs. What is the time complexity of your function?

4. Program Algorithm 5.2 for the multistage graph problem using the backward approach. Assume that the graph is represented using inverse adjacency lists. Test its correctness. What is its complexity?

5. Do Exercise 4 using the graph representation of Exercise 3. This time, however, assume that $G[q,2] \leq G[q+1,2]$ for $1 \leq q < e$.

6. Extend the discussion of this section to directed acyclic graphs (dags). Suppose the vertices of a dag are numbered so that all edges have the form $\langle i, j \rangle$, $i < j$. What changes, if any, need to be made to Algorithm 5.1 to find the length of the longest path from vertex 1 to vertex n?

https://hemanthrajhemu.github.io

7. [W. Miller] Show that BGraph1 computes shortest paths for directed acyclic graphs represented by adjacency lists (instead of inverse adjacency lists as in BGraph).

```
1    Algorithm BGraph1(G, n)
2    {
3        bcost[1] := 0.0;
4        for j := 2 to n do bcost[j] := ∞;
5        for j := 1 to n − 1 do
6            for each r such that ⟨j, r⟩ is an edge of G do
7                bcost[r] := min(bcost[r], bcost[j] + c[j, r]);
8    }
```

Note: There is a possibility of a floating point overflow in this function. In such cases the program should be suitably modified.

5.3 ALL-PAIRS SHORTEST PATHS

Let $G = (V, E)$ be a directed graph with n vertices. Let *cost* be a cost adjacency matrix for G such that $cost(i,i) = 0$, $1 \leq i \leq n$. Then $cost(i,j)$ is the length (or cost) of edge $\langle i,j \rangle$ if $\langle i,j \rangle \in E(G)$ and $cost(i,j) = \infty$ if $i \neq j$ and $\langle i,j \rangle \notin E(G)$. The *all-pairs shortest-path problem* is to determine a matrix A such that $A(i,j)$ is the length of a shortest path from i to j. The matrix A can be obtained by solving n single-source problems using the algorithm ShortestPaths of Section 4.8. Since each application of this procedure requires $O(n^2)$ time, the matrix A can be obtained in $O(n^3)$ time. We obtain an alternate $O(n^3)$ solution to this problem using the principle of optimality. Our alternate solution requires a weaker restriction on edge costs than required by ShortestPaths. Rather than require $cost(i,j) \geq 0$, for every edge $\langle i,j \rangle$, we only require that G have no cycles with negative length. Note that if we allow G to contain a cycle of negative length, then the shortest path between any two vertices on this cycle has length $-\infty$.

Let us examine a shortest i to j path in G, $i \neq j$. This path originates at vertex i and goes through some intermediate vertices (possibly none) and terminates at vertex j. We can assume that this path contains no cycles for if there is a cycle, then this can be deleted without increasing the path length (no cycle has negative length). If k is an intermediate vertex on this shortest path, then the subpaths from i to k and from k to j must be shortest paths from i to k and k to j, respectively. Otherwise, the i to j path is not of minimum length. So, the principle of optimality holds. This alerts us to the prospect of using dynamic programming. If k is the intermediate vertex with highest index, then the i to k path is a shortest i to k path in G going through no vertex with index greater than $k - 1$. Similarly the k to j path is a shortest k to j path in G going through no vertex of index greater than

https://hemanthrajhemu.github.io

$k-1$. We can regard the construction of a shortest i to j path as first requiring a decision as to which is the highest indexed intermediate vertex k. Once this decision has been made, we need to find two shortest paths, one from i to k and the other from k to j. Neither of these may go through a vertex with index greater than $k-1$. Using $A^k(i,j)$ to represent the length of a shortest path from i to j going through no vertex of index greater than k, we obtain

$$A(i,j) = \min \ \{\min_{1\leq k\leq n} \{A^{k-1}(i,k) + A^{k-1}(k,j)\}, cost(i,j)\} \tag{5.7}$$

Clearly, $A^0(i,j) = cost(i,j)$, $1 \leq i \leq n$, $1 \leq j \leq n$. We can obtain a recurrence for $A^k(i,j)$ using an argument similar to that used before. A shortest path from i to j going through no vertex higher than k either goes through vertex k or it does not. If it does, $A^k(i,j) = A^{k-1}(i,k) + A^{k-1}(k,j)$. If it does not, then no intermediate vertex has index greater than $k-1$. Hence $A^k(i,j) = A^{k-1}(i,j)$. Combining, we get

$$A^k(i,j) = \min \ \{A^{k-1}(i,j),\ A^{k-1}(i,k) + A^{k-1}(k,j)\},\quad k \geq 1 \tag{5.8}$$

The following example shows that (5.8) is not true for graphs with cycles of negative length.

Example 5.14 Figure 5.5 shows a digraph together with its matrix A^0. For this graph $A^2(1,3) \neq \min\{A^1(1,3),\ A^1(1,2) + A^1(2,3)\} = 2$. Instead we see that $A^2(1,3) = -\infty$. The length of the path

$$1, 2, 1, 2, 1, 2, \ldots, 1, 2, 3$$

can be made arbitrarily small. This is so because of the presence of the cycle 1 2 1 which has a length of -1. □

Recurrence (5.8) can be solved for A^n by first computing A^1, then A^2, then A^3, and so on. Since there is no vertex in G with index greater than n, $A(i,j) = A^n(i,j)$. Function AllPaths computes $A^n(i,j)$. The computation is done inplace so the superscript on A is not needed. The reason this computation can be carried out in-place is that $A^k(i,k) = A^{k-1}(i,k)$ and $A^k(k,j) = A^{k-1}(k,j)$. Hence, when A^k is formed, the kth column and row do not change. Consequently, when $A^k(i,j)$ is computed in line 11 of Algorithm 5.3, $A(i,k) = A^{k-1}(i,k) = A^k(i,k)$ and $A(k,j) = A^{k-1}(k,j) = A^k(k,j)$. So, the old values on which the new values are based do not change on this iteration.

https://hemanthrajhemu.github.io

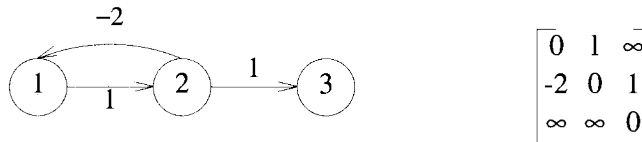


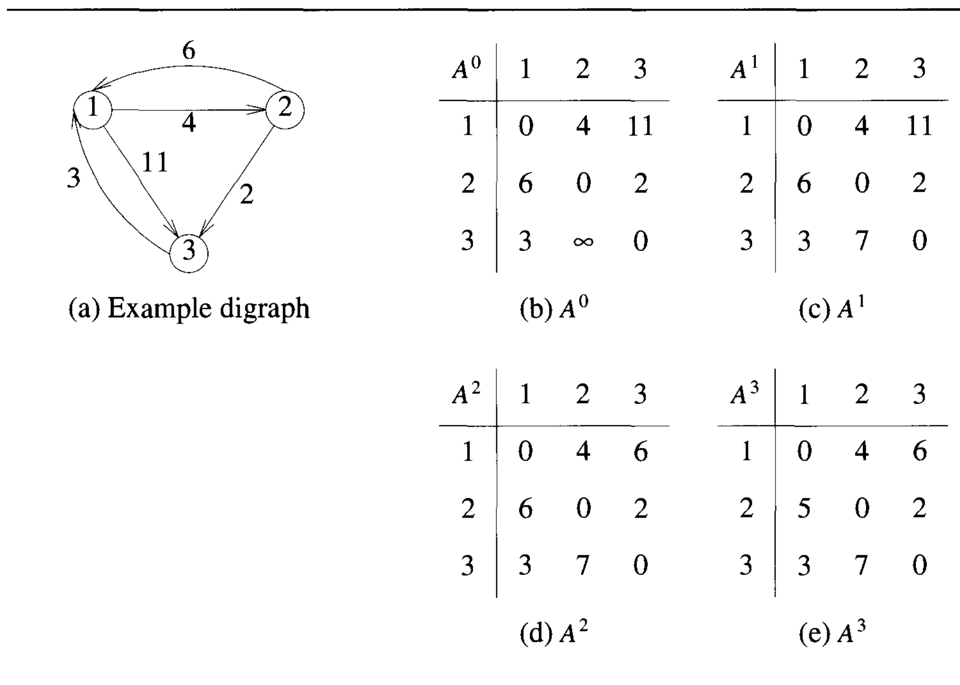
Figure 5.5 Graph with negative cycle

```
0   Algorithm AllPaths(cost, A, n)
1   // cost[1 : n, 1 : n] is the cost adjacency matrix of a graph with
2   // n vertices; A[i, j] is the cost of a shortest path from vertex
3   // i to vertex j. cost[i, i] = 0.0, for 1 ≤ i ≤ n.
4   {
5       for i := 1 to n do
6           for j := 1 to n do
7               A[i, j] := cost[i, j]; // Copy cost into A.
8       for k := 1 to n do
9           for i := 1 to n do
10              for j := 1 to n do
11                  A[i, j] := min(A[i, j], A[i, k] + A[k, j]);
12  }
```

Algorithm 5.3 Function to compute lengths of shortest paths

https://hemanthrajhemu.github.io

Example 5.15 The graph of Figure 5.6(a) has the cost matrix of Figure 5.6(b). The initial A matrix, $A^{(0)}$, plus its values after 3 iterations $A^{(1)}$, $A^{(2)}$, and $A^{(3)}$ are given in Figure 5.6. □



A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

(b) A^0

A^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

(c) A^1

A^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

(d) A^2

A^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

(e) A^3

Figure 5.6 Directed graph and associated matrices

Let $M = \max\ \{cost(i,j) | \langle i,j \rangle \in E(G)\}$. It is easy to see that $A^n(ij) \leq (n-1)M$. From the working of AllPaths, it is clear that if $\langle i,j \rangle \notin E(G)$ and $i \neq j$, then we can initialize $cost(i,j)$ to any number greater than $(n-1)M$ (rather than the maximum allowable floating point number). If, at termination, $A(i,j) > (n-1)M$, then there is no directed path from i to j in G. Even for this choice of ∞, care should be taken to avoid any floating point overflows.

The time needed by AllPaths (Algorithm 5.3) is especially easy to determine because the looping is independent of the data in the matrix A. Line 11 is iterated n^3 times, and so the time for AllPaths is $\Theta(n^3)$. An exercise examines the extensions needed to obtain the i to j paths with these lengths. Some speedup can be obtained by noticing that the innermost **for** loop need be executed only when $A(i,k)$ and $A(k,j)$ are not equal to ∞.

https://hemanthrajhemu.github.io

EXERCISES

1. (a) Does the recurrence (5.8) hold for the graph of Figure 5.7? Why?

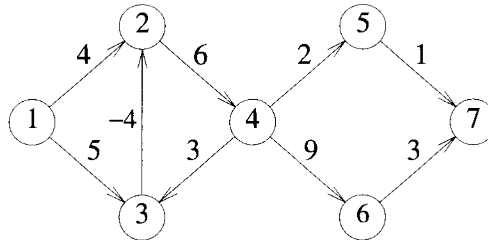


Figure 5.7 Graph for Exercise 1

   (b) Why does Equation 5.8 not hold for graphs with cycles of negative length?

2. Modify the function AllPaths so that a shortest path is output for each pair of vertices (i, j). What are the time and space complexities of the new algorithm?

3. Let A be the adjacency matrix of a directed graph G. Define the transitive closure A^+ of A to be a matrix with the property $A^+(i, j) = 1$ iff G has a directed path, containing at least one edge, from vertex i to vertex j. $A^+(i, j) = 0$ otherwise. The reflexive transitive closure A^* is a matrix with the property $A^*(i, j) = 1$ iff G has a path, containing zero or more edges, from i to j. $A^*(i, j) = 0$ otherwise.

   (a) Obtain A^+ and A^* for the directed graph of Figure 5.8.

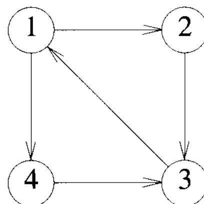


Figure 5.8 Graph for Exercise 3

   (b) Let $A^k(i, j) = 1$ iff there is a path with zero or more edges from i to j going through no vertex of index greater than k. Define A^0 in terms of the adjacency matrix A.

https://hemanthrajhemu.github.io

(c) Obtain a recurrence between A^k and A^{k-1} similar to (5.8). Use the logical operators **or** and **and** rather than **min** and $+$.

(d) Write an algorithm, using the recurrence of part (c), to find A^*. Your algorithm can use only $O(n^2)$ space. What is its time complexity?

(e) Show that $A^+ = A \times A^*$, where matrix multiplication is defined as $A^+(i,j) = \vee_{k=1}^{n}(A(i,k) \wedge A^*(k,j))$. The operation $\vee$ is the logical **or** operation, and $\wedge$ the logical **and** operation. Hence A^+ may be computed from A^*.

5.4 SINGLE-SOURCE SHORTEST PATHS: GENERAL WEIGHTS

We now consider the single-source shortest path problem discussed in Section 4.8 when some or all of the edges of the directed graph G may have negative length. ShortestPaths (Algorithm 4.14) does not necessarily give the correct results on such graphs. To see this, consider the graph of Figure 5.9. Let $v = 1$ be the source vertex. Referring back to Algorithm 4.14, since $n = 3$, the loop of lines 12 to 22 is iterated just once. Also $u = 3$ in lines 15 and 16, and so no changes are made to $dist[\ ]$. The algorithm terminates with $dist[2] = 7$ and $dist[3] = 5$. The shortest path from 1 to 3 is $1, 2, 3$. This path has length 2, which is less than the computed value of $dist[3]$.

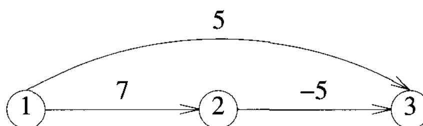


Figure 5.9 Directed graph with a negative-length edge

When negative edge lengths are permitted, we require that the graph have no cycles of negative length. This is necessary to ensure that shortest paths consist of a finite number of edges. For example, in the graph of Figure 5.5, the length of the shortest path from vertex 1 to vertex 3 is $-\infty$. The length of the path

$$1, 2, 1, 2, 1, 2, \cdots, 1, 2, 3$$

can be made arbitrarily small as was shown in Example 5.14.

When there are no cycles of negative length, there is a shortest path between any two vertices of an n-vertex graph that has at most $n-1$ edges

https://hemanthrajhemu.github.io

on it. To see this, note that a path that has more than $n-1$ edges must repeat at least one vertex and hence must contain a cycle. Elimination of the cycles from the path results in another path with the same source and destination. This path is cycle-free and has a length that is no more than that of the original path, as the length of the eliminated cycles was at least zero. We can use this observation on the maximum number of edges on a cycle-free shortest path to obtain an algorithm to determine a shortest path from a source vertex to all remaining vertices in the graph. As in the case of ShortestPaths (Algorithm 4.14), we compute only the length, $dist[u]$, of the shortest path from the source vertex v to u. An exercise examines the extension needed to construct the shortest paths.

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 $dist$:

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

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

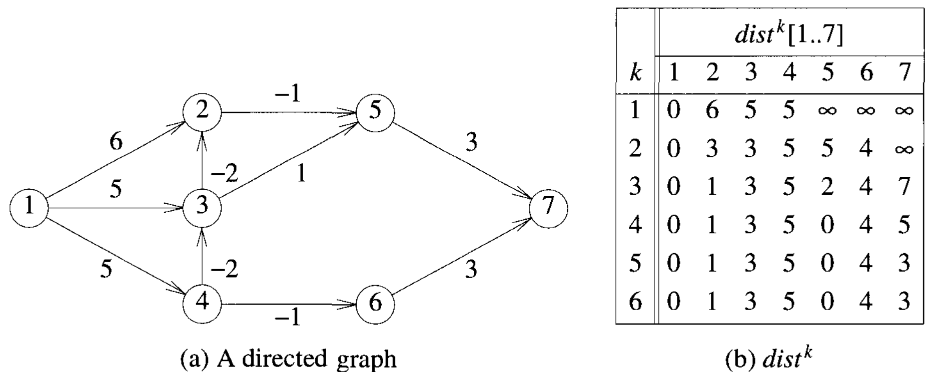
Example 5.16 Figure 5.10 gives a seven-vertex graph, together with the arrays $dist^k$, $k = 1, \ldots, 6$. These arrays were computed using the equation just given. For instance, $dist^k[1] = 0$ for all k since 1 is the source node. Also, $dist^1[2] = 6, dist^1[3] = 5$, and $dist^1[4] = 5$, since there are edges from

https://hemanthrajhemu.github.io

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

k	$dist^k[1..7]$						
	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

An exercise shows that if we use the same memory location $dist[u]$ for $dist^k[u]$, $k = 1, \ldots, n - 1$, then the final value of $dist[u]$ is still $dist^{n-1}[u]$. Using this fact and the recurrence for $dist$ shown above, we arrive at the pseudocode of Algorithm 5.4 to compute the length of the shortest path from vertex v to each other vertex of the graph. This algorithm is referred to as the Bellman and Ford algorithm.

Each iteration of the **for** loop of lines 7 to 12 takes $O(n^2)$ time if adjacency matrices are used and $O(e)$ time if adjacency lists are used. Here e is the number of edges in the graph. The overall complexity is $O(n^3)$ when adjacency matrices are used and $O(ne)$ when adjacency lists are used. The observed complexity of the shortest-path algorithm can be reduced by noting that if none of the $dist$ values change on one iteration of the **for** loop of lines 7 to 12, then none will change on successive iterations. So, this loop can be rewritten to terminate either after $n - 1$ iterations or after the

https://hemanthrajhemu.github.io

```
1   Algorithm BellmanFord(v, cost, dist, n)
2   // Single-source/all-destinations shortest
3   // paths with negative edge costs
4   {
5       for i := 1 to n do // Initialize dist.
6           dist[i] := cost[v, i];
7       for k := 2 to n − 1 do
8           for each u such that u ≠ v and u has
9                   at least one incoming edge do
10              for each ⟨i, u⟩ in the graph do
11                  if dist[u] > dist[i] + cost[i, u] then
12                      dist[u] := dist[i] + cost[i, u];
13  }
```

Algorithm 5.4 Bellman and Ford algorithm to compute shortest paths

first iteration in which no *dist* values are changed, whichever occurs first. Another possibility is to maintain a queue of vertices i whose *dist* values changed on the previous iteration of the **for** loop. These are the only values for i that need to be considered in line 10 during the next iteration. When a queue of these values is maintained, we can rewrite the loop of lines 7 to 12 so that on each iteration, a vertex i is removed from the queue, and the *dist* values of all vertices adjacent from i are updated as in lines 11 and 12. Vertices whose *dist* values decrease as a result of this are added to the end of the queue unless they are already on it. The loop terminates when the queue becomes empty. These two strategies to improve the performance of BellmanFord are considered in the exercises. Other strategies for improving performance are discussed in References and Readings. □

EXERCISES

1. Find the shortest paths from node 1 to every other node in the graph of Figure 5.11 using the Bellman and Ford algorithm.

2. Prove the correctness of BellmanFord (Algorithm 5.4). Note that this algorithm does not faithfully implement the computation of the recurrence for $dist^k$. In fact, for $k < n-1$, the *dist* values following iteration k of the **for** loop of lines 7 to 12 may not be $dist^k$.

3. Transform BellmanFord into a program. Assume that graphs are represented using adjacency lists in which each node has an additional field

https://hemanthrajhemu.github.io

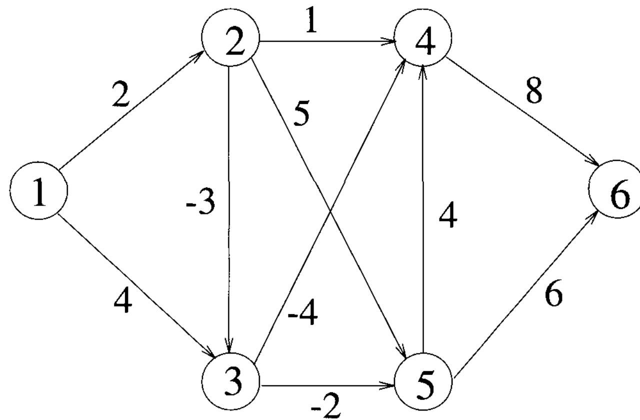


Figure 5.11 Graph for Exercise 1

called *cost* that gives the length of the edge represented by that node. As a result of this, there is no cost adjacency matrix. Generate some test graphs and test the correctness of your program.

4. Rewrite the algorithm BellmanFord so that the loop of lines 7 to 12 terminates either after $n - 1$ iterations or after the first iteration in which no *dist* values are changed, whichever occurs first.

5. Rewrite BellmanFord by replacing the loop of lines 7 to 12 with code that uses a queue of vertices that may potentially result in a reduction of other *dist* vertices. This queue initially contains all vertices that are adjacent from the source vertex v. On each successive iteration of the new loop, a vertex i is removed from the queue (unless the queue is empty), and the *dist* values to vertices adjacent from i are updated as in lines 11 and 12 of Algorithm 5.4. When the *dist* value of a vertex is reduced because of this, it is added to the queue unless it is already on the queue.

   (a) Prove that the new algorithm produces the same results as the original one.
   (b) Show that the complexity of the new algorithm is no more than that of the original one.

6. Compare the run-time performance of the Bellman and Ford algorithms of the preceding two exercises and that of Algorithm 5.4. For this, generate test graphs that will expose the relative performances of the three algorithms.

https://hemanthrajhemu.github.io

7. Modify algorithm BellmanFord so that it obtains the shortest paths, in addition to the lengths of these paths. What is the computing time of your algorithm?

5.5 OPTIMAL BINARY SEARCH TREES (*)

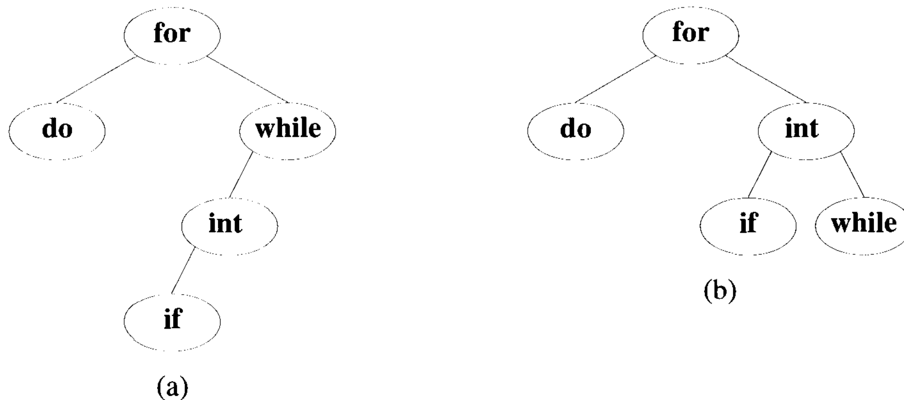


Figure 5.12 Two possible binary search trees

Given a fixed set of identifiers, we wish to create a binary search tree (see Section 2.3) organization. We may expect different binary search trees for the same identifier set to have different performance characteristics. The tree of Figure 5.12(a), in the worst case, requires four comparisons to find an identifier, whereas the tree of Figure 5.12(b) requires only three. On the average the two trees need 12/5 and 11/5 comparisons, respectively. For example, in the case of tree (a), it takes 1, 2, 2, 3, and 4 comparisons, respectively, to find the identifiers **for, do, while, int,** and **if**. Thus the average number of comparisons is $\frac{1+2+2+3+4}{5} = \frac{12}{5}$. This calculation assumes that each identifier is searched for with equal probability and that no unsuccessful searches (i.e., searches for identifiers not in the tree) are made.

In a general situation, we can expect different identifiers to be searched for with different frequencies (or probabilities). In addition, we can expect unsuccessful searches also to be made. Let us assume that the given set of identifiers is $\{a_1, a_2, \ldots, a_n\}$ with $a_1 < a_2 < \cdots < a_n$. Let $p(i)$ be the probability with which we search for a_i. Let $q(i)$ be the probability that the identifier x being searched for is such that $a_i < x < a_{i+1}$, $0 \leq i \leq n$ (assume $a_0 = -\infty$ and $a_{n+1} = +\infty$). Then, $\sum_{0 \leq i \leq n} q(i)$ is the probability of

https://hemanthrajhemu.github.io

an unsuccessful search. Clearly, $\sum_{1 \le i \le n} p(i) + \sum_{0 \le i \le n} q(i) = 1$. Given this data, we wish to construct an optimal binary search tree for $\{a_1, a_2, \ldots, a_n\}$. First, of course, we must be precise about what we mean by an optimal binary search tree.

In obtaining a cost function for binary search trees, it is useful to add a fictitious node in place of every empty subtree in the search tree. Such nodes, called external nodes, are drawn square in Figure 5.13. All other nodes are internal nodes. If a binary search tree represents n identifiers, then there will be exactly n internal nodes and $n+1$ (fictitious) external nodes. Every internal node represents a point where a successful search may terminate. Every external node represents a point where an unsuccessful search may terminate.

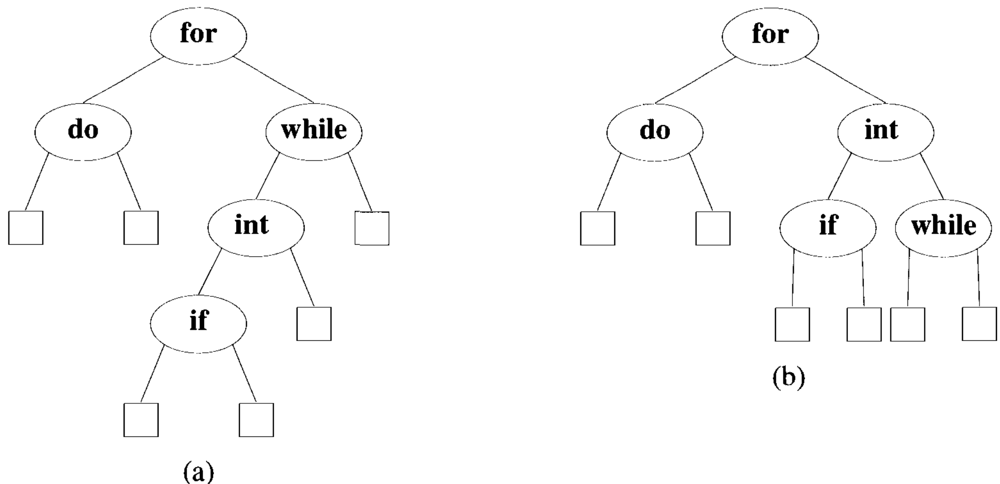


Figure 5.13 Binary search trees of Figure 5.12 with external nodes added

If a successful search terminates at an internal node at level l, then l iterations of the **while** loop of Algorithm 2.5 are needed. Hence, the expected cost contribution from the internal node for a_i is $p(i) * \text{level}(a_i)$.

Unsuccessful searches terminate with $t = 0$ (i.e., at an external node) in algorithm ISearch (Algorithm 2.5). The identifiers not in the binary search tree can be partitioned into $n+1$ equivalence classes $E_i, 0 \le i \le n$. The class E_0 contains all identifiers x such that $x < a_1$. The class E_i contains all identifiers x such that $a_i < x < a_{i+1}$, $1 \le i < n$. The class E_n contains all identifiers x, $x > a_n$. It is easy to see that for all identifiers in the same class E_i, the search terminates at the same external node. For identifiers in different E_i the search terminates at different external nodes. If the failure

https://hemanthrajhemu.github.io

node for E_i is at level l, then only $l-1$ iterations of the **while** loop are made. Hence, the cost contribution of this node is $q(i) * (level(E_i) - 1)$.

The preceding discussion leads to the following formula for the expected cost of a binary search tree:

$$\sum_{1 \le i \le n} p(i) * level(a_i) + \sum_{0 \le i \le n} q(i) * (level(E_i) - 1) \tag{5.9}$$

We define an optimal binary search tree for the identifier set $\{a_1, a_2, \ldots, a_n\}$ to be a binary search tree for which (5.9) is minimum.

Example 5.17 The possible binary search trees for the identifier set (a_1, a_2, a_3) = (**do, if, while**) are given if Figure 5.14. With equal probabilities $p(i) = q(i) = 1/7$ for all i, we have

$$\begin{array}{lcllcl} cost(\text{tree a}) & = & 15/7 & cost(\text{tree b}) & = & 13/7 \\ cost(\text{tree c}) & = & 15/7 & cost(\text{tree d}) & = & 15/7 \\ cost(\text{tree e}) & = & 15/7 & & & \end{array}$$

As expected, tree b is optimal. With $p(1) = .5$, $p(2) = .1$, $p(3) = .05$, $q(0) = .15$, $q(1) = .1$, $q(2) = .05$ and $q(3) = .05$ we have

$$\begin{array}{lcllcl} cost(\text{tree a}) & = & 2.65 & cost(\text{tree b}) & = & 1.9 \\ cost(\text{tree c}) & = & 1.5 & cost(\text{tree d}) & = & 2.05 \\ cost(\text{tree e}) & = & 1.6 & & & \end{array}$$

For instance, $cost$(tree a) can be computed as follows. The contribution from successful searches is $3*0.5+2*0.1+0.05 = 1.75$ and the contribution from unsuccessful searches is $3*0.15+3*0.1+2*0.05+0.05 = 0.90$. All the other costs can also be calculated in a similar manner. Tree c is optimal with this assignment of p's and q's. □

To apply dynamic programming to the problem of obtaining an optimal binary search tree, we need to view the construction of such a tree as the result of a sequence of decisions and then observe that the principle of optimality holds when applied to the problem state resulting from a decision. A possible approach to this would be to make a decision as to which of the a_i's should be assigned to the root node of the tree. If we choose a_k, then it is clear that the internal nodes for $a_1, a_2, \ldots, a_{k-1}$ as well as the external nodes for the classes $E_0, E_1, \ldots, E_{k-1}$ will lie in the left subtree l of the root. The remaining nodes will be in the right subtree r. Define

$$cost(l) = \sum_{1 \le i < k} p(i) * \text{level}(a_i) + \sum_{0 \le i < k} q(i) * (\text{level}(E_i) - 1)$$

https://hemanthrajhemu.github.io

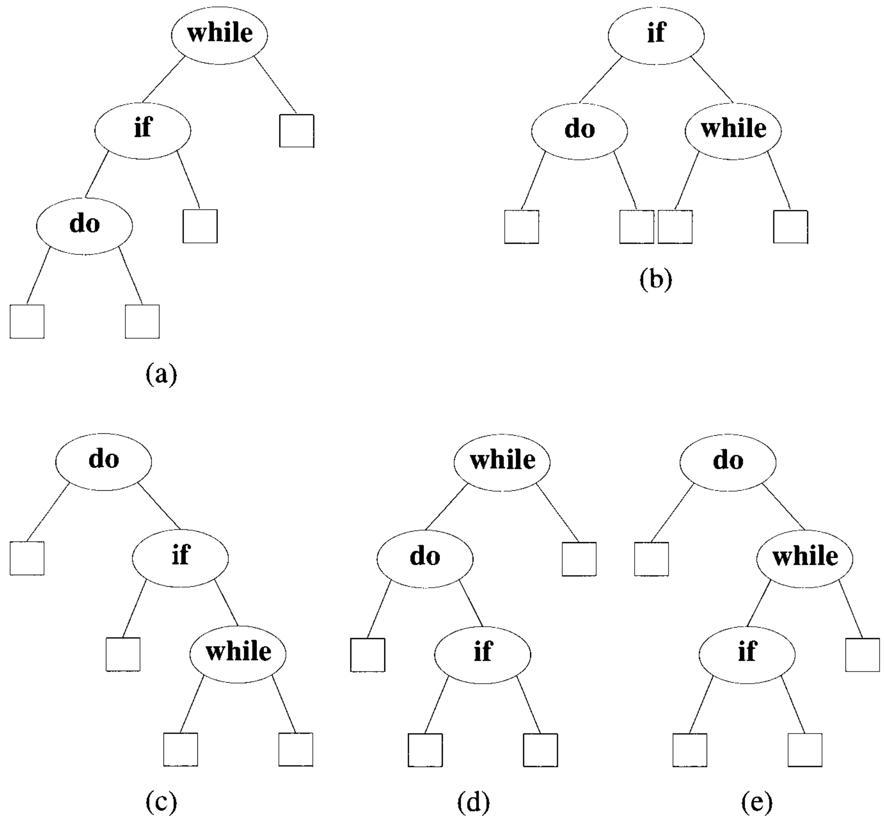


Figure 5.14 Possible binary search trees for the identifier set {**do**, **if**, **while**}

https://hemanthrajhemu.github.io

and

$$cost(r) = \sum_{k<i\leq n} p(i) * \text{level}(a_i) + \sum_{k<i\leq n} q(i) * (\text{level}(E_i) - 1)$$

In both cases the level is measured by regarding the root of the respective subtree to be at level 1.

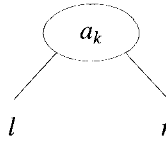


Figure 5.15 An optimal binary search tree with root a_k

Using $w(i,j)$ to represent the sum $q(i) + \sum_{l=i+1}^{j}(q(l) + p(l))$, we obtain the following as the expected cost of the search tree (Figure 5.15):

$$p(k) + cost(l) + cost(r) + w(0, k-1) + w(k, n) \tag{5.10}$$

If the tree is optimal, then (5.10) must be minimum. Hence, $cost(l)$ must be minimum over all binary search trees containing $a_1, a_2, \ldots, a_{k-1}$ and $E_0, E_1, \ldots, E_{k-1}$. Similarly $cost(r)$ must be minimum. If we use $c(i,j)$ to represent the cost of an optimal binary search tree t_{ij} containing $a_{i+1}, \ldots, a_j$ and $E_i, \ldots, E_j$, then for the tree to be optimal, we must have $cost(l) = c(0, k-1)$ and $cost(r) = c(k, n)$. In addition, k must be chosen such that

$$p(k) + c(0, k-1) + c(k, n) + w(0, k-1) + w(k, n)$$

is minimum. Hence, for $c(0, n)$ we obtain

$$c(0, n) = \min_{1\leq k\leq n} \{c(0, k-1) + c(k, n) + p(k) + w(0, k-1) + w(k, n)\} \tag{5.11}$$

We can generalize (5.11) to obtain for any $c(i,j)$

$$c(i,j) \quad = \quad \min_{i<k\leq j} \{c(i, k-1) + c(k, j) + p(k) + w(i, k-1) + w(k, j)\}$$

https://hemanthrajhemu.github.io

$$c(i,j) = \min_{i<k\leq j}\{c(i,k-1)+c(k,j)\}+w(i,j) \tag{5.12}$$

Equation 5.12 can be solved for $c(0,n)$ by first computing all $c(i,j)$ such that $j-i=1$ (note $c(i,i)=0$ and $w(i,i)=q(i)$, $0\leq i\leq n$). Next we can compute all $c(i,j)$ such that $j-i=2$, then all $c(i,j)$ with $j-i=3$, and so on. If during this computation we record the root $r(i,j)$ of each tree t_{ij}, then an optimal binary search tree can be constructed from these $r(i,j)$. Note that $r(i,j)$ is the value of k that minimizes (5.12).

Example 5.18 Let $n=4$ and $(a_1,a_2,a_3,a_4)=$ (**do, if, int, while**). Let $p(1:4)=(3,3,1,1)$ and $q(0:4)=(2,3,1,1,1)$. The p's and q's have been multiplied by 16 for convenience. Initially, we have $w(i,i)=q(i), c(i,i)=0$ and $r(i,i)=0$, $0\leq i\leq 4$. Using Equation 5.12 and the observation $w(i,j)=p(j)+q(j)+w(i,j-1)$, we get

$$\begin{aligned}
w(0,1) &= p(1)+q(1)+w(0,0)=8\\
c(0,1) &= w(0,1)+\min\{c(0,0)+c(1,1)\} = 8\\
r(0,1) &= 1\\
w(1,2) &= p(2)+q(2)+w(1,1) = 7\\
c(1,2) &= w(1,2)+\min\ \{c(1,1)+c(2,2)\} = 7\\
r(0,2) &= 2\\
w(2,3) &= p(3)+q(3)+w(2,2) = 3\\
c(2,3) &= w(2,3)+\min\ \{c(2,2)+c(3,3)\} = 3\\
r(2,3) &= 3\\
w(3,4) &= p(4)+q(4)+w(3,3) = 3\\
c(3,4) &= w(3,4)+\min\ \{c(3,3)+c(4,4)\} = 3\\
r(3,4) &= 4
\end{aligned}$$

Knowing $w(i,i+1)$ and $c(i,i+1)$, $0\leq i<4$, we can again use Equation 5.12 to compute $w(i,i+2)$, $c(i,i+2)$, and $r(i,i+2)$, $0\leq i<3$. This process can be repeated until $w(0,4)$, $c(0,4)$, and $r(0,4)$ are obtained. The table of Figure 5.16 shows the results of this computation. The box in row i and column j shows the values of $w(j,j+i)$, $c(j,j+i)$ and $r(j,j+i)$ respectively. The computation is carried out by row from row 0 to row 4. From the table we see that $c(0,4)=32$ is the minimum cost of a binary search tree for (a_1,a_2,a_3,a_4). The root of tree t_{04} is a_2. Hence, the left subtree is t_{01} and the right subtree t_{24}. Tree t_{01} has root a_1 and subtrees t_{00} and t_{11}. Tree t_{24} has root a_3; its left subtree is t_{22} and its right subtree t_{34}. Thus, with the data in the table it is possible to reconstruct t_{04}. Figure 5.17 shows t_{04}. □

https://hemanthrajhemu.github.io

	0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	
2	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$		
3	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$			
4	$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$				

Figure 5.16 Computation of $c(0,4)$, $w(0,4)$, and $r(0,4)$

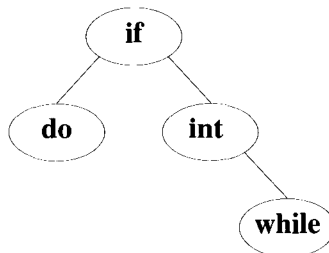


Figure 5.17 Optimal search tree for Example 5.18

https://hemanthrajhemu.github.io

The above example illustrates how Equation 5.12 can be used to determine the c's and r's and also how to reconstruct t_{0n} knowing the r's. Let us examine the complexity of this procedure to evaluate the c's and r's. The evaluation procedure described in the above example requires us to compute $c(i,j)$ for $(j-i) = 1, 2, \ldots, n$ in that order. When $j - i = m$, there are $n - m + 1$ $c(i,j)$'s to compute. The computation of each of these $c(i,j)$'s requires us to find the minimum of m quantities (see Equation 5.12). Hence, each such $c(i,j)$ can be computed in time $O(m)$. The total time for all $c(i,j)$'s with $j - i = m$ is therefore $O(nm - m^2)$. The total time to evaluate all the $c(i,j)$'s and $r(i,j)$'s is therefore

$$\sum_{1 \le m \le n} (nm - m^2) = O(n^3)$$

We can do better than this using a result due to D. E. Knuth which shows that the optimal k in Equation 5.12 can be found by limiting the search to the range $r(i, j-1) \le k \le r(i+1, j)$. In this case the computing time becomes $O(n^2)$ (see the exercises). The function OBST (Algorithm 5.5) uses this result to obtain the values of $w(i,j)$, $r(i,j)$, and $c(i,j)$, $0 \le i \le j \le n$, in $O(n^2)$ time. The tree t_{0n} can be constructed from the values of $r(i,j)$ in $O(n)$ time. The algorithm for this is left as an exercise.

EXERCISES

1. Use function OBST (Algorithm 5.5) to compute $w(i,j)$, $r(i,j)$, and $c(i,j)$, $0 \le i < j \le 4$, for the identifier set (a_1, a_2, a_3, a_4) = (**cout, float, if, while**) with $p(1) = 1/20$, $p(2) = 1/5$, $p(3) = 1/10$, $p(4) = 1/20$, $q(0) = 1/5$, $q(1) = 1/10$, $q(2) = 1/5$, $q(3) = 1/20$, and $q(4) = 1/20$. Using the $r(i,j)$'s, construct the optimal binary search tree.

2. (a) Show that the computing time of function OBST (Algorithm 5.5) is $O(n^2)$.

   (b) Write an algorithm to construct the optimal binary search tree given the roots $r(i,j), 0 \le i < j \le n$. Show that this can be done in time $O(n)$.

3. Since often only the approximate values of the p's and q's are known, it is perhaps just as meaningful to find a binary search tree that is nearly optimal. That is, its cost, Equation 5.9, is almost minimal for the given p's and q's. This exercise explores an $O(n \log n)$ algorithm that results in nearly optimal binary search trees. The search tree heuristic we use is

https://hemanthrajhemu.github.io

```
1   Algorithm OBST(p, q, n)
2   // Given n distinct identifiers a_1 < a_2 < ... < a_n and probabilities
3   // p[i], 1 ≤ i ≤ n, and q[i], 0 ≤ i ≤ n, this algorithm computes
4   // the cost c[i, j] of optimal binary search trees t_ij for identifiers
5   // a_{i+1}, ..., a_j. It also computes r[i, j], the root of t_ij.
6   // w[i, j] is the weight of t_ij.
7   {
8       for i := 0 to n - 1 do
9       {
10          // Initialize.
11          w[i, i] := q[i]; r[i, i] := 0; c[i, i] := 0.0;
12          // Optimal trees with one node
13          w[i, i + 1] := q[i] + q[i + 1] + p[i + 1];
14          r[i, i + 1] := i + 1;
15          c[i, i + 1] := q[i] + q[i + 1] + p[i + 1];
16      }
17      w[n, n] := q[n]; r[n, n] := 0; c[n, n] := 0.0;
18      for m := 2 to n do  // Find optimal trees with m nodes.
19          for i := 0 to n - m do
20          {
21              j := i + m;
22              w[i, j] := w[i, j - 1] + p[j] + q[j];
23              // Solve 5.12 using Knuth's result.
24              k := Find(c, r, i, j);
25                  // A value of l in the range r[i, j - 1] ≤ l
26                  // ≤ r[i + 1, j] that minimizes c[i, l - 1] + c[l, j];
27              c[i, j] := w[i, j] + c[i, k - 1] + c[k, j];
28              r[i, j] := k;
29          }
30      write (c[0, n], w[0, n], r[0, n]);
31  }
```

```
1   Algorithm Find(c, r, i, j)
2   {
3       min := ∞;
4       for m := r[i, j - 1] to r[i + 1, j] do
5           if (c[i, m - 1] + c[m, j]) < min then
6           {
7               min := c[i, m - 1] + c[m, j]; l := m;
8           }
9       return l;
10  }
```

Algorithm 5.5 Finding a minimum-cost binary search tree

https://hemanthrajhemu.github.io

Choose the root k such that $|w(0, k-1) - w(k, n)|$ is as small as possible. Repeat this procedure to find the left and right subtrees of the root.

(a) Using this heuristic, obtain the resulting binary search tree for the data of Exercise 1. What is its cost?

(b) Write an algorithm implementing the above heuristic. Your algorithm should have time complexity $O(n \log n)$.

5.6 STRING EDITING

We are given two strings $X = x_1, x_2, \ldots, x_n$ and $Y = y_1, y_2, \ldots, y_m$, where x_i, $1 \leq i \leq n$, and y_j, $1 \leq j \leq m$, are members of a finite set of symbols known as the *alphabet*. We want to transform X into Y using a sequence of *edit operations* on X. The permissible edit operations are insert, delete, and change (a symbol of X into another), and there is a cost associated with performing each. The cost of a sequence of operations is the sum of the costs of the individual operations in the sequence. The problem of string editing is to identify a minimum-cost sequence of edit operations that will transform X into Y.

Let $D(x_i)$ be the cost of deleting the symbol x_i from X, $I(y_j)$ be the cost of inserting the symbol y_j into X, and $C(x_i, y_j)$ be the cost of changing the symbol x_i of X into y_j.

Example 5.19 Consider the sequences $X = x_1, x_2, x_3, x_4, x_5 = a, a, b, a, b$ and $Y = y_1, y_2, y_3, y_4 = b, a, b, b$. Let the cost associated with each insertion and deletion be 1 (for any symbol). Also let the cost of changing any symbol to any other symbol be 2. One possible way of transforming X into Y is delete each $x_i, 1 \leq i \leq 5$, and insert each $y_j, 1 \leq j \leq 4$. The total cost of this edit sequence is 9. Another possible edit sequence is delete x_1 and x_2 and insert y_4 at the end of string X. The total cost is only 3. □

A solution to the string editing problem consists of a sequence of decisions, one for each edit operation. Let $\mathcal{E}$ be a minimum-cost edit sequence for transforming X into Y. The first operation, O, in $\mathcal{E}$ is delete, insert, or change. If $\mathcal{E}' = \mathcal{E} - \{O\}$ and X' is the result of applying O on X, then $\mathcal{E}'$ should be a minimum-cost edit sequence that transforms X' into Y. Thus the principle of optimality holds for this problem. A dynamic programming solution for this problem can be obtained as follows. Define $cost(i, j)$ to be the minimum cost of any edit sequence for transforming $x_1, x_2, \ldots, x_i$ into $y_1, y_2, \ldots, y_j$ (for $0 \leq i \leq n$ and $0 \leq j \leq m$). Compute $cost(i, j)$ for each i and j. Then $cost(n, m)$ is the cost of an optimal edit sequence.

For $i = j = 0$, $cost(i, j) = 0$, since the two sequences are identical (and empty). Also, if $j = 0$ and $i > 0$, we can transform X into Y by a sequence of

https://hemanthrajhemu.github.io

deletes. Thus, $cost(i,0) = cost(i-1,0)+D(x_i)$. Similarly, if $i = 0$ and $j > 0$, we get $cost(0,j) = cost(0,j-1) + I(y_j)$. If $i \neq 0$ and $j \neq 0$, $x_1, x_2, \ldots, x_i$ can be transformed into $y_1, y_2, \ldots, y_j$ in one of three ways:

1. Transform $x_1, x_2, \ldots, x_{i-1}$ into $y_1, y_2, \ldots, y_j$ using a minimum-cost edit sequence and then delete x_i. The corresponding cost is $cost(i-1,j) + D(x_i)$.

2. Transform $x_1, x_2, \ldots, x_{i-1}$ into $y_1, y_2, \ldots, y_{j-1}$ using a minimum-cost edit sequence and then change the symbol x_i to y_j. The associated cost is $cost(i-1,j-1) + C(x_i, y_j)$.

3. Transform $x_1, x_2, \ldots, x_i$ into $y_1, y_2, \ldots, y_{j-1}$ using a minimum-cost edit sequence and then insert y_j. This corresponds to a cost of $cost(i, j-1) + I(y_j)$.

The minimum cost of any edit sequence that transforms $x_1, x_2, \ldots, x_i$ into $y_1, y_2, \ldots, y_j$ (for $i > 0$ and $j > 0$) is the minimum of the above three costs, according to the principle of optimality. Therefore, we arrive at the following recurrence equation for $cost(i,j)$:

$$cost(i,j) = \begin{cases} 0 & i = j = 0 \\ cost(i-1,0) + D(x_i) & j = 0,\ i > 0 \\ cost(0,j-1) + I(y_j) & i = 0,\ j > 0 \\ cost'(i,j) & i > 0,\ j > 0 \end{cases} \tag{5.13}$$

$$\text{where } cost'(i,j) = \min\{\ cost(i-1,j) + D(x_i),\ cost(i-1,j-1) + C(x_i,y_j),\ cost(i,j-1) + I(y_j)\ \}$$

We have to compute $cost(i,j)$ for all possibles values of i and j ($0 \leq i \leq n$ and $0 \leq j \leq m$). There are $(n+1)(m+1)$ such values. These values can be computed in the form of a table, M, where each row of M corresponds to a particular value of i and each column of M corresponds to a specific value of j. $M(i,j)$ stores the value $cost(i,j)$. The zeroth row can be computed first since it corresponds to performing a series of insertions. Likewise the zeroth column can also be computed. After this, one could compute the entries of M in row-major order, starting from the first row. Rows should be processed in the order $1, 2, \ldots, n$. Entries in any row are computed in increasing order of column number.

The entries of M can also be computed in column-major order, starting from the first column. Looking at Equation 5.13, we see that each entry of M takes only $O(1)$ time to compute. Therefore the whole algorithm takes $O(mn)$ time. The value $cost(n,m)$ is the final answer we are interested in. Having computed all the entries of M, a minimum edit sequence can be

https://hemanthrajhemu.github.io

obtained by a simple backward trace from $cost(n, m)$. This backward trace is enabled by recording which of the three options for $i > 0, j > 0$ yielded the minimum cost for each i and j.

Example 5.20 Consider the string editing problem of Example 5.19. $X = a, a, b, a, b$ and $Y = b, a, b, b$. Each insertion and deletion has a unit cost and a change costs 2 units. For the cases $i = 0, j > 1$, and $j = 0, i > 1$, $cost(i, j)$ can be computed first (Figure 5.18). Let us compute the rest of the entries in row-major order. The next entry to be computed is $cost(1, 1)$.

$$\begin{array}{rcl} cost(1,1) & = & \min\ \{cost(0,1) + D(x_1), cost(0,0) + C(x_1, y_1), cost(1,0) + I(y_1)\} \\ & = & \min\ \{2,2,2\} = 2 \end{array}$$

Next is computed $cost(1, 2)$.

$$\begin{array}{rcl} cost(1,2) & = & \min\ \{cost(0,2) + D(x_1), cost(0,1) + C(x_1, y_2), cost(1,1) + I(y_2)\} \\ & = & \min\ \{3,1,3\} = 1 \end{array}$$

The rest of the entries are computed similarly. Figure 5.18 displays the whole table. The value $cost(5, 4) = 3$. One possible minimum-cost edit sequence is delete x_1, delete x_2, and insert y_4. Another possible minimum cost edit sequence is change x_1 to y_2 and delete x_4. □

$i \downarrow$ \ $j \rightarrow$	0	1	2	3	4
0	0	1	2	3	4
1	1	2	1	2	3
2	2	3	2	3	4
3	3	2	3	2	3
4	4	3	2	3	4
5	5	4	3	2	3

Figure 5.18 Cost table for Example 5.20

https://hemanthrajhemu.github.io

EXERCISES

1. Let $X = a, a, b, a, a, b, a, b, a, a$ and $Y = b, a, b, a, a, b, a, b$. Find a minimum-cost edit sequence that transforms X into Y.

2. Present a pseudocode algorithm that implements the string editing algorithm discussed in this section. Program it and test its correctness using suitable data.

3. Modify the above program not only to compute $cost(n, m)$ but also to output a minimum-cost edit sequence. What is the time complexity of your program?

4. Given a sequence X of symbols, a subsequence of X is defined to be any contiguous portion of X. For example, if $X = x_1, x_2, x_3, x_4, x_5$, x_2, x_3 and x_1, x_2, x_3 are subsequences of X. Given two sequences X and Y, present an algorithm that will identify the longest subsequence that is common to both X and Y. This problem is known as *the longest common subsequence problem.* What is the time complexity of your algorithm?

5.7 0/1 KNAPSACK

The terminology and notation used in this section is the same as that in Section 5.1. A solution to the knapsack problem can be obtained by making a sequence of decisions on the variables $x_1, x_2, \ldots, x_n$. A decision on variable x_i involves determining which of the values 0 or 1 is to be assigned to it. Let us assume that decisions on the x_i are made in the order $x_n, x_{n-1}, \ldots, x_1$. Following a decision on x_n, we may be in one of two possible states: the capacity remaining in the knapsack is m and no profit has accrued or the capacity remaining is $m - w_n$ and a profit of p_n has accrued. It is clear that the remaining decisions $x_{n-1}, \ldots, x_1$ must be optimal with respect to the problem state resulting from the decision on x_n. Otherwise, $x_n, \ldots, x_1$ will not be optimal. Hence, the principle of optimality holds.

Let $f_j(y)$ be the value of an optimal solution to KNAP$(1, j, y)$. Since the principle of optimality holds, we obtain

$$f_n(m) = \max \ \{f_{n-1}(m), f_{n-1}(m - w_n) + p_n\} \tag{5.14}$$

For arbitrary $f_i(y)$, $i > 0$, Equation 5.14 generalizes to

$$f_i(y) = \max \ \{f_{i-1}(y), f_{i-1}(y - w_i) + p_i\} \tag{5.15}$$

Equation 5.15 can be solved for $f_n(m)$ by beginning with the knowledge $f_0(y) = 0$ for all y and $f_i(y) = -\infty, y < 0$. Then $f_1, f_2, \ldots, f_n$ can be successively computed using (5.15).

https://hemanthrajhemu.github.io

When the w_i's are integer, we need to compute $f_i(y)$ for integer y, $0 \leq y \leq m$. Since $f_i(y) = -\infty$ for $y < 0$, these function values need not be computed explicitly. Since each f_i can be computed from f_{i-1} in $\Theta(m)$ time, it takes $\Theta(mn)$ time to compute f_n. When the w_i's are real numbers, $f_i(y)$ is needed for real numbers y such that $0 \leq y \leq m$. So, f_i cannot be explicitly computed for all y in this range. Even when the w_i's are integer, the explicit $\Theta(mn)$ computation of f_n may not be the most efficient computation. So, we explore an alternative method for both cases.

Notice that $f_i(y)$ is an ascending step function; i.e., there are a finite number of y's, $0 = y_1 < y_2 < \cdots < y_k$, such that $f_i(y_1) < f_i(y_2) < \cdots < f_i(y_k)$; $f_i(y) = -\infty$, $y < y_1$; $f_i(y) = f(y_k)$, $y \geq y_k$; and $f_i(y) = f_i(y_j)$, $y_j \leq y < y_{j+1}$. So, we need to compute only $f_i(y_j)$, $1 \leq j \leq k$. We use the ordered set $S^i = \{(f(y_j), y_j) | 1 \leq j \leq k\}$ to represent $f_i(y)$. Each member of S^i is a pair (P, W), where $P = f_i(y_j)$ and $W = y_j$. Notice that $S^0 = \{(0, 0)\}$. We can compute S^{i+1} from S^i by first computing

$$S_1^i = \{(P, W) | (P - p_i, W - w_i) \in S^i\} \tag{5.16}$$

Now, S^{i+1} can be computed by merging the pairs in S^i and S_1^i together. Note that if S^{i+1} contains two pairs (P_j, W_j) and (P_k, W_k) with the property that $P_j \leq P_k$ and $W_j \geq W_k$, then the pair (P_j, W_j) can be discarded because of (5.15). Discarding or purging rules such as this one are also known as *dominance rules*. Dominated tuples get purged. In the above, (P_k, W_k) dominates (P_j, W_j).

Interestingly, the strategy we have come up with can also be derived by attempting to solve the knapsack problem via a systematic examination of the up to 2^n possibilities for $x_1, x_2, \ldots, x_n$. Let S^i represent the possible states resulting from the 2^i decision sequences for $x_1, \ldots, x_i$. A state refers to a pair (P_j, W_j), W_j being the total weight of objects included in the knapsack and P_j being the corresponding profit. To obtain S^{i+1}, we note that the possibilities for x_{i+1} are $x_{i+1} = 0$ or $x_{i+1} = 1$. When $x_{i+1} = 0$, the resulting states are the same as for S^i. When $x_{i+1} = 1$, the resulting states are obtained by adding (p_{i+1}, w_{i+1}) to each state in S^i. Call the set of these additional states S_1^i. The S_1^i is the same as in Equation 5.16. Now, S^{i+1} can be computed by merging the states in S^i and S_1^i together.

Example 5.21 Consider the knapsack instance $n = 3, (w_1, w_2, w_3) = (2, 3, 4)$, $(p_1, p_2, p_3) = (1, 2, 5)$, and $m = 6$. For these data we have

$$\begin{aligned}
S^0 &= \{(0,0)\}; S_1^0 = \{(1,2)\} \\
S^1 &= \{(0,0), (1,2)\}; S_1^1 = \{(2,3), (3,5)\} \\
S^2 &= \{(0,0), (1,2), (2,3), (3,5)\}; S_1^2 = \{(5,4), (6,6), (7,7), (8,9)\} \\
S^3 &= \{(0,0), (1,2), (2,3), (5,4), (6,6), (7,7), (8,9)\}
\end{aligned}$$

https://hemanthrajhemu.github.io

Note that the pair (3, 5) has been eliminated from S^3 as a result of the purging rule stated above. □

When generating the S^i's, we can also purge all pairs (P, W) with $W > m$ as these pairs determine the value of $f_n(x)$ only for $x > m$. Since the knapsack capacity is m, we are not interested in the behavior of f_n for $x > m$. When all pairs (P_j, W_j) with $W_j > m$ are purged from the S^i's, $f_n(m)$ is given by the P value of the last pair in S^n (note that the S^i's are ordered sets). Note also that by computing S^n, we can find the solutions to all the knapsack problems KNAP$(1, n, x)$, $0 \le x \le m$, and not just KNAP$(1, n, m)$. Since, we want only a solution to KNAP$(1, n, m)$, we can dispense with the computation of S^n. The last pair in S^n is either the last one in S^{n-1} or it is $(P_j + p_n, W_j + w_n)$, where $(P_j, W_j) \in S^{n-1}$ such that $W_j + w_n \le m$ and W_j is maximum.

If $(P1, W1)$ is the last tuple in S^n, a set of 0/1 values for the x_i's such that $\sum p_i x_i = P1$ and $\sum w_i x_i = W1$ can be determined by carrying out a search through the S^is. We can set $x_n = 0$ if $(P1, W1) \in S^{n-1}$. If $(P1, W1) \notin S^{n-1}$, then $(P1 - p_n, W1 - w_n) \in S^{n-1}$ and we can set $x_n = 1$. This leaves us to determine how either $(P1, W1)$ or $(P1 - p_n, W1 - w_n)$ was obtained in S^{n-1}. This can be done recursively.

Example 5.22 With $m = 6$, the value of $f_3(6)$ is given by the tuple (6, 6) in S^3 (Example 5.21). The tuple (6, 6) $\notin S^2$, and so we must set $x_3 = 1$. The pair (6, 6) came from the pair $(6 - p_3, 6 - w_3) = (1, 2)$. Hence (1, 2) $\in S^2$. Since $(1, 2) \in S^1$, we can set $x_2 = 0$. Since (1, 2) $\notin S^0$, we obtain $x_1 = 1$. Hence an optimal solution is $(x_1, x_2, x_3) = (1, 0, 1)$. □

We can sum up all we have said so far in the form of an informal algorithm DKP (Algorithm 5.6). To evaluate the complexity of the algorithm, we need to specify how the sets S^i and S_1^i are to be represented; provide an algorithm to merge S^i and S_1^i; and specify an algorithm that will trace through $S^{n-1}, \ldots, S^1$ and determine a set of 0/1 values for $x_n, \ldots, x_1$.

We can use an array *pair*[ ] to represent all the pairs (P, W). The P values are stored in *pair*[ ]*.p* and the W values in *pair*[ ]*.w*. Sets $S^0, S^1, \ldots, S^{n-1}$ can be stored adjacent to each other. This requires the use of pointers $b[i]$, $0 \le i \le n$, where $b[i]$ is the location of the first element in S^i, $0 \le i < n$, and $b[n]$ is one more than the location of the last element in S^{n-1}.

Example 5.23 Using the representation above, the sets S^0, S^1, and S^2 of Example 5.21 appear as

https://hemanthrajhemu.github.io

```
1   Algorithm DKP(p, w, n, m)
2   {
3       S^0 := {(0,0)};
4       for i := 1 to n - 1 do
5       {
6           S_1^{i-1} := {(P,W)|(P - p_i, W - w_i) ∈ S^{i-1} and W ≤ m};
7           S^i := MergePurge(S^{i-1}, S_1^{i-1});
8       }
9       (PX, WX) :=last pair in S^{n-1};
10      (PY, WY) := (P' + p_n, W' + w_n) where W' is the largest W in
11          any pair in S^{n-1} such that W + w_n ≤ m;
12      // Trace back for x_n, x_{n-1}, ..., x_1.
13      if (PX > PY) then x_n := 0;
14      else x_n := 1;
15      TraceBackFor(x_{n-1}, ..., x_1);
16  }
```

Algorithm 5.6 Informal knapsack algorithm

	1	2	3	4	5	6	7
pair[ ].*p*	0	0	1	0	1	2	3
pair[ ].*w*	0	0	2	0	2	3	5

↑ $b[0]$ ↑ $b[1]$ ↑ $b[2]$ ↑ $b[3]$ □

The merging and purging of S^{i-1} and S_1^{i-1} can be carried out at the same time that S_1^{i-1} is generated. Since the pairs in S^{i-1} are in increasing order of P and W, the pairs for S^i are generated in this order. If the next pair generated for S_1^{i-1} is (PQ, WQ), then we can merge into S^i all pairs from S^{i-1} with W value $\leq WQ$. The purging rule can be used to decide whether any pairs get purged. Hence, no additional space is needed in which to store S_1^{i-1}.

DKnap (Algorithm 5.7) generates S^i from S^{i-1} in this way. The S^i's are generated in the **for** loop of lines 7 to 42 of Algorithm 5.7. At the start of each iteration $t = b[i-1]$ and h is the index of the last pair in S^{i-1}. The variable k points to the next tuple in S^{i-1} that has to be merged into S^i. In line 10, the function Largest determines the largest q, $t \leq q \leq h$,

https://hemanthrajhemu.github.io

for which $pair[q].w + w[i] \leq m$. This can be done by performing a binary search. The code for this function is left as an exercise. Since u is set such that for all $W_j, h \geq j > u$, $W_j + w_i > m$, the pairs for S_1^{i-1} are $(P(j) + p_i, W(j) + w_i)$, $1 \leq j \leq u$. The **for** loop of lines 11 to 33 generates these pairs. Each time a pair (pp, ww) is generated, all pairs (P, W) in S^{i-1} with $W < ww$ not yet purged or merged into S^i are merged into S^i. Note that none of these may be purged. Lines 21 to 25 handle the case when the next pair in S^{i-1} has a W value equal to ww. In this case the pair with lesser P value gets purged. In case $pp > P(next-1)$, then the pair (pp, ww) gets purged. Otherwise, (pp, ww) is added to S^i. The **while** loop of lines 31 and 32 purges all unmerged pairs in S^{i-1} that can be purged at this time. Finally, following the merging of S_1^{i-1} into S^i, there may be pairs remaining in S^{i-1} to be merged into S^i. This is taken care of in the **while** loop of lines 35 to 39. Note that because of lines 31 and 32, none of these pairs can be purged. Function TraceBack (line 43) implements the **if** statement and trace-back step of the function DKP (Algorithm 5.6). This is left as an exercise.

If $|S^i|$ is the number of pairs in S^i, then the array *pair* should have a minimum dimension of $d = \sum_{0 \leq i \leq n-1} |S^i|$. Since it is not possible to predict the exact space needed, it is necessary to test for $next > d$ each time $next$ is incremented. Since each S^i, $i > 0$, is obtained by merging S^{i-1} and S_1^{i-1} and $|S_1^{i-1}| \leq |S^{i-1}|$, it follows that $|S^i| \leq 2|S^{i-1}|$. In the worst case no pairs will get purged and

$$\sum_{0 \leq i \leq n-1} |S^i| = \sum_{0 \leq i \leq n-1} 2^i = 2^n - 1$$

The time needed to generate S^i from S^{i-1} is $\Theta(|S^{i-1}|)$. Hence, the time needed to compute all the S^i's, $0 \leq i < n$, is $\Theta(\sum |S^{i-1}|)$. Since $|S^i| \leq 2^i$, the time needed to compute all the S^i's is $O(2^n)$. If the p_j's are integers, then each pair (P, W) in S^i has an integer P and $P \leq \sum_{1 \leq j \leq i} p_j$. Similarly, if the w_j's are integers, each W is an integer and $W \leq m$. In any S^i the pairs have distinct W values and also distinct P values. Hence,

$$|S^i| \leq 1 + \sum_{1 \leq j \leq i} p_j$$

when the p_j's are integers and

$$|S^i| \leq 1 + \min \{\sum_{1 \leq j \leq i} w_j, m\}$$

https://hemanthrajhemu.github.io

```
    PW = record {float p; float w; }

1   Algorithm DKnap(p, w, x, n, m)
2   {
3       // pair[ ] is an array of PW's.
4       b[0] := 1; pair[1].p := pair[1].w := 0.0; // S^0
5       t := 1; h := 1; // Start and end of S^0
6       b[1] := next := 2; // Next free spot in pair[ ]
7       for i := 1 to n - 1 do
8       { // Generate S^i.
9           k := t;
10          u := Largest(pair, w, t, h, i, m);
11          for j := t to u do
12          { // Generate S_1^{i-1} and merge.
13              pp := pair[j].p + p[i]; ww := pair[j].w + w[i];
14                  // (pp, ww) is the next element in S_1^{i-1}.
15              while ((k ≤ h) and (pair[k].w ≤ ww)) do
16              {
17                  pair[next].p := pair[k].p;
18                  pair[next].w := pair[k].w;
19                  next := next + 1; k := k + 1;
20              }
21              if ((k ≤ h) and (pair[k].w = ww)) then
22              {
23                  if pp < pair[k].p then pp := pair[k].p;
24                  k := k + 1;
25              }
26              if pp > pair[next - 1].p then
27              {
28                  pair[next].p := pp; pair[next].w := ww;
29                  next := next + 1 ;
30              }
31              while ((k ≤ h) and (pair[k].p ≤ pair[next - 1].p))
32                  do k := k + 1;
33          }
34          // Merge in remaining terms from S^{i-1}.
35          while (k ≤ h) do
36          {
37              pair[next].p := pair[k].p; pair[next].w := pair[k].w;
38              next := next + 1; k := k + 1;
39          }
40          // Initialize for S^{i+1}.
41          t := h + 1; h := next - 1; b[i + 1] := next;
42      }
43      TraceBack(p, w, pair, x, m, n);
44  }
```

https://hemanthrajhemu.github.io

when the w_j's are integers. When both the p_j's and w_j's are integers, the time and space complexity of DKnap (excluding the time for TraceBack) is $O(\min\{2^n, n\sum_{1\le i\le n} p_i, nm\})$. In this bound $\sum_{1\le i\le n} p_i$ can be replaced by $\sum_{1\le i\le n} p_i/\gcd(p_1, \ldots, p_n)$ and m by $\gcd(w_1, w_2, \ldots, w_n, m)$ (see the exercises). The exercises indicate how TraceBack may be implemented so as to have a space complexity $O(1)$ and a time complexity $O(n^2)$.

Although the above analysis may seem to indicate that DKnap requires too much computational resource to be practical for large n, in practice many instances of this problem can be solved in a reasonable amount of time. This happens because usually, all the p's and w's are integers and m is much smaller than 2^n. The purging rule is effective in purging most of the pairs that would otherwise remain in the S^i's.

Algorithm DKnap can be speeded up by the use of heuristics. Let L be an estimate on the value of an optimal solution such that $f_n(m) \ge L$. Let PLEFT$(i) = \sum_{i<j\le n} p_j$. If S^i contains a tuple (P, W) such that $P +$ PLEFT$(i) < L$, then (P, W) can be purged from S^i. To see this, observe that (P, W) can contribute at best the pair $(P + \sum_{i<j\le n} p_j, W + \sum_{i<j\le n} w)$ to S_1^{n-1}. Since $P + \sum_{i<j\le n} p_j = P +$ PLEFT$(i) < L$, it follows that this pair cannot lead to a pair with value at least L and so cannot determine an optimal solution. A simple way to estimate L such that $L \le f_n(m)$ is to consider the last pair (P, W) in S^i. Then, $P \le f_n(m)$. A better estimate is obtained by adding some of the remaining objects to (P, W). Example 5.24 illustrates this. Heuristics for the knapsack problem are discussed in greater detail in the chapter on branch-and-bound. The exercises explore a divide-and-conquer approach to speed up DKnap so that the worst case time is $O(2^{n/2})$.

Example 5.24 Consider the following instance of the knapsack problem: $n = 6, (p_1, p_2, p_3, p_4, p_5, p_6) = (w_1, w_2, w_3, w_4, w_5, w_6) = (100, 50, 20, 10, 7, 3)$, and $m = 165$. Attempting to fill the knapsack using objects in the order 1, 2, 3, 4, 5, and 6, we see that objects 1, 2, 4, and 6 fit in and yield a profit of 163 and a capacity utilization of 163. We can thus begin with $L = 163$ as a value with the property $L \le f_n(m)$. Since $p_i = w_i$, every pair $(P, W) \in S^i$, $0 \le i \le 6$ has $P = W$. Hence, each pair can be replaced by the singleton P or W. PLEFT(0) = 190, PLEFT(1) = 90, PLEFT(2) = 40, PLEFT(3) = 20, PLEFT(4) = 10, PLEFT(5) = 3, and PLEFT(6) = 0. Eliminating from each S^i any singleton P such that $P+$ PLEFT$(i) < L$, we obtain

$$S^0 = \{0\}; \quad S_1^0 = \{100\}$$
$$S^1 = \{100\}; \quad S_1^1 = \{150\}$$
$$S^2 = \{150\}; \quad S_1^2 = \phi$$

https://hemanthrajhemu.github.io

$$S^3 = \{150\}; \quad S_1^3 = \{160\}$$
$$S^4 = \{160\}; \quad S_1^4 = \phi$$
$$S^5 = \{160\}$$

The singleton 0 is deleted from S^1 as 0 + PLEFT(1)< 163. The set S_1^2 does not contain the singleton 150 + 20 = 170 as $m < 170$. S^3 does not contain the 100 or the 120 as each is less than L − PLEFT(3). And so on. The value $f_6(165)$ can be determined from S^5. In this example, the value of L did not change. In general, L will change if a better estimate is obtained as a result of the computation of some S^i. If the heuristic wasn't used, then the computation would have proceeded as

$$\begin{aligned}
S^0 &= \{0\} \\
S^1 &= \{0, 100\} \\
S^2 &= \{0, 50, 100, 150\} \\
S^3 &= \{0, 20, 50, 70, 100, 120, 150\} \\
S^4 &= \{0, 10, 20, 30, 50, 60, 70, 80, 100, 110, 120, 130, 150, 160\} \\
S^5 &= \{0, 7, 10, 17, 20, 27, 30, 37, 50, 57, 60, 67, 70, 77, 80, 87, 100, \\
&\quad 107, 110, 117, 120, 127, 130, 137, 150, 157, 160\}
\end{aligned}$$

The value $f_6(165)$ can now be determined from S^5, using the knowledge (p_6, w_6) = (3, 3). □

EXERCISES

1. Generate the sets $S^i, 0 \leq i \leq 4$ (Equation 5.16), when $(w_1, w_2, w_3, w_4) = (10, 15, 6, 9)$ and $(p_1, p_2, p_3, p_4) = (2, 5, 8, 1)$.

2. Write a function Largest($pair, w, t, h, i, m$) that uses binary search to determine the largest q, $t \leq q \leq h$, such that $pair[q].w + w[i] \leq m$.

3. Write a function TraceBack to determine an optimal solution $x_1, x_2, \ldots, x_n$ to the knapsack problem. Assume that $S^i, 0 \leq i < n$, have already been computed as in function DKnap. Knowing $b(i)$ and $b(i+1)$, you can use a binary search to determine whether $(P', W') \in S^i$. Hence, the time complexity of your algorithm should be no more than $O(n \max_i \{\log |S^i|\}) = O(n^2)$.

4. Give an example of a set of knapsack instances for which $|S^i| = 2^i$, $0 \leq i \leq n$. Your set should include one instance for each n.

https://hemanthrajhemu.github.io

5. (a) Show that if the p_j's are integers, then the size of each S^i, $|S^i|$, in the knapsack problem is no more than $1+\sum_{1\le i\le j} p_j/gcd(p_1, p_2, \ldots, p_n)$, where $gcd(p_1, p_2, \ldots, p_n)$ is the greatest common divisor of the p_i's.

   (b) Show that when the w_j's are integer, then $|S^i| \le 1 + \min\{\sum_{1\le j\le i} w_j, m\}/gcd(w_1, w_2, \ldots, w_n, m)$.

6. (a) Using a divide-and-conquer approach coupled with the set generation approach of the text, show how to obtain an $O(2^{n/2})$ algorithm for the 0/1 knapsack problem.

   (b) Develop an algorithm that uses this approach to solve the 0/1 knapsack problem.

   (c) Compare the run time and storage requirements of this approach with those of Algorithm 5.7. Use suitable test data.

7. Consider the integer knapsack problem obtained by replacing the 0/1 constraint in (5.2) by $x_i \ge 0$ and integer. Generalize $f_i(x)$ to this problem in the obvious way.

   (a) Obtain the dynamic programming recurrence relation corresponding to (5.15).

   (b) Show how to transform this problem into a 0/1 knapsack problem. (*Hint*: Introduce new 0/1 variables for each x_i. If $0 \le x_i < 2^j$, then introduce j variables, one for each bit in the binary representation of x_i.)

5.8 RELIABILITY DESIGN

In this section we look at an example of how to use dynamic programming to solve a problem with a multiplicative optimization function. The problem is to design a system that is composed of several devices connected in series (Figure 5.19). Let r_i be the reliability of device D_i (that is, r_i is the probability that device i will function properly). Then, the reliability of the entire system is Πr_i. Even if the individual devices are very reliable (the r_i's are very close to one), the reliability of the system may not be very good. For example, if $n = 10$ and $r_i = .99$, $1 \le i \le 10$, then $\Pi r_i = .904$. Hence, it is desirable to duplicate devices. Multiple copies of the same device type are connected in parallel (Figure 5.20) through the use of switching circuits. The switching circuits determine which devices in any given group are functioning properly. They then make use of one such device at each stage.

If stage i contains m_i copies of device D_i, then the probability that all m_i have a malfunction is $(1 - r_i)^{m_i}$. Hence the reliability of stage i becomes

https://hemanthrajhemu.github.io

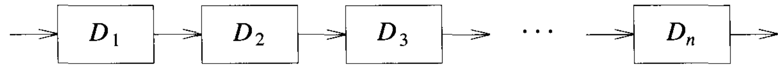


Figure 5.19 n devices D_i, $1 \le i \le n$, connected in series

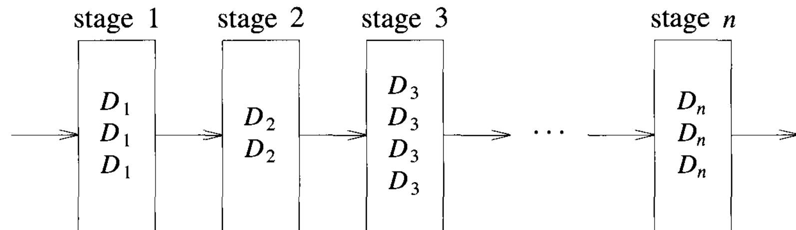


Figure 5.20 Multiple devices connected in parallel in each stage

$1 - (1 - r_i)^{m_i}$. Thus, if $r_i = .99$ and $m_i = 2$, the stage reliability becomes .9999. In any practical situation, the stage reliability is a little less than $1-(1-r_i)^{m_i}$ because the switching circuits themselves are not fully reliable. Also, failures of copies of the same device may not be fully independent (e.g., if failure is due to design defect). Let us assume that the reliability of stage i is given by a function $\phi_i(m_i)$, $1 \le n$. (It is quite conceivable that $\phi_i(m_i)$ may decrease after a certain value of m_i.) The reliability of the system of stages is $\Pi_{1\le i\le n}\phi_i(m_i)$.

Our problem is to use device duplication to maximize reliability. This maximization is to be carried out under a cost constraint. Let c_i be the cost of each unit of device i and let c be the maximum allowable cost of the system being designed. We wish to solve the following maximization problem:

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

$$\text{subject to } \sum_{1\le i\le n} c_i m_i \le c \tag{5.17}$$

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

https://hemanthrajhemu.github.io

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 follows from the observation that $m_j \ge 1$. An optimal solution $m_1, m_2, \ldots, m_n$ is the result of a sequence of decisions, one decision for each m_i. Let $f_i(x)$ represent the maximum value of $\Pi_{1 \le j \le i}\ \phi(m_j)$ subject to the constraints $\sum_{1 \le j \le i} c_j m_j \le x$ and $1 \le m_j \le u_j$, $1 \le j \le i$. Then, the value of an optimal solution is $f_n(c)$. The last decision made requires one to choose m_n from $\{1, 2, 3, \ldots, u_n\}$. Once a value for m_n has been chosen, the remaining decisions must be such as to use the remaining funds $c - c_n m_n$ in an optimal way. The principal of optimality holds and

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

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, $f_0(x) = 1$ for all x, $0 \le x \le c$. Hence, (5.19) can be solved using an approach similar to that used for the knapsack problem. Let S^i consist of tuples of the form (f, x), where $f = f_i(x)$. There is at most one tuple for each different x that results from a sequence of decisions on $m_1, m_2, \ldots, m_n$. The dominance rule (f_1, x_1) dominates (f_2, x_2) iff $f_1 \ge f_2$ and $x_1 \le x_2$ holds for this problem too. Hence, dominated tuples can be discarded from S^i.

Example 5.25 We are to design a three stage system with device types D_1, D_2, and D_3. The costs are \$30, \$15, and \$20 respectively. The cost of the system is to be no more than \$105. The reliability of each device type is .9, .8 and .5 respectively. We assume that if stage i has m_i devices of type i in parallel, then $\phi_i(m_i) = 1 - (1 - r_i)^{m_i}$. In terms of the notation used earlier, $c_1 = 30$, $c_2 = 15$, $c_3 = 20$, $c = 105$, $r_1 = .9$, $r_2 = .8$, $r_3 = .5, u_1 = 2, u_2 = 3$, and $u_3 = 3$.

We use S^i to represent the set of all undominated tuples (f, x) that may result from the various decision sequences for $m_1, m_2, \ldots, m_i$. Hence, $f(x) = f_i(x)$. Beginning with $S^0 = \{(1, 0)\}$, we can obtain each S^i from S^{i-1} by trying out all possible values for m_i and combining the resulting tuples together. Using S_j^i to represent all tuples obtainable from S^{i-1} by choosing $m_i = j$, we obtain $S_1^1 = \{(.9,\ 30)\}$ and $S_2^1 = \{(.9,\ 30), (.99, 60)\}$. The set

https://hemanthrajhemu.github.io

$S_1^2 = \{(.72, 45), (.792, 75)\}$; $S_2^2 = \{(.864, 60)\}$. 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.

EXERCISE

1. (a) Present an algorithm similar to DKnap to solve the recurrence (5.19).
   (b) What are the time and space requirements of your algorithm?
   (c) Test the correctness of your algorithm using suitable test data.

5.9 THE TRAVELING SALESPERSON PROBLEM

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

https://hemanthrajhemu.github.io

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).

Our final example is from a production environment in which several commodities are manufactured on the same set of machines. The manufacture proceeds in cycles. In each production cycle, n different commodities are produced. When the machines are changed from production of commodity i to commodity j, a change over cost c_{ij} is incurred. It is desired to find a sequence in which to manufacture these commodities. This sequence should minimize the sum of change over costs (the remaining production costs are sequence independent). Since the manufacture proceeds cyclically, it is necessary to include the cost of starting the next cycle. This is just the change over cost from the last to the first commodity. Hence, this problem can be regarded as a traveling salesperson problem on an n vertex graph with edge cost c_{ij}'s being the changeover cost from commodity i to commodity j.

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

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

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,

https://hemanthrajhemu.github.io

$g(i, \phi) = c_{i1}$, $1 \le i \le n$. 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).

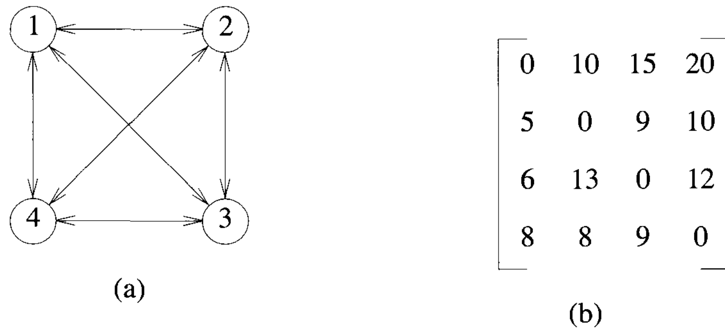


Figure 5.21 Directed graph and edge length matrix c

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

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

https://hemanthrajhemu.github.io

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.

EXERCISE

1. (a) Obtain a data representation for the values $g(i, S)$ of the traveling salesperson problem. Your representation should allow for easy access to the value of $g(i, S)$, given i and S. (i) How much space does your representation need for an n vertex graph? (ii) How much time is needed to retrieve or update the value of $g(i, S)$?
   (b) Using the representation of (a), develop an algorithm corresponding to the dynamic programming solution of the traveling salesperson problem.
   (c) Test the correctness of your algorithm using suitable test data.

5.10 FLOW SHOP SCHEDULING

Often the processing of a job requires the performance of several distinct tasks. Computer programs run in a multiprogramming environment are input and then executed. Following the execution, the job is queued for output

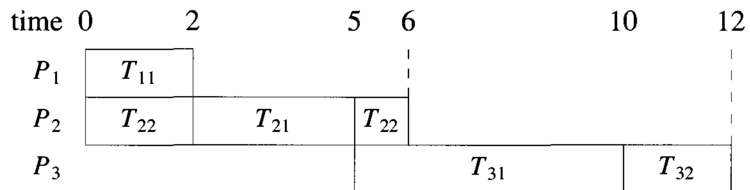
https://hemanthrajhemu.github.io

and the output eventually printed. In a general flow shop we may have n jobs each requiring m tasks $T_{1i}, T_{2i}, \ldots, T_{mi}$, $1 \leq i \leq n$, to be performed. Task T_{ji} is to be performed on processor P_j, $1 \leq j \leq m$. The time required to complete task T_{ji} is t_{ji}. A schedule for the n jobs is an assignment of tasks to time intervals on the processors. Task T_{ji} must be assigned to processor P_j. No processor may have more than one task assigned to it in any time interval. Additionally, for any job i the processing of task T_{ji}, $j > 1$, cannot be started until task $T_{j-1,i}$ has been completed.

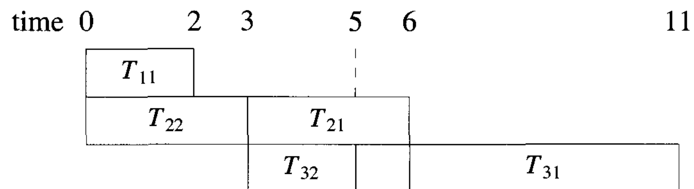
Example 5.27 Two jobs have to be scheduled on three processors. The task times are given by the matrix $\mathcal{J}$

$$\mathcal{J} = \begin{bmatrix} 2 & 0 \\ 3 & 3 \\ 5 & 2 \end{bmatrix}$$

Two possible schedules for the jobs are shown in Figure 5.22. □



(a)



(b)

Figure 5.22 Two possible schedules for Example 5.27

https://hemanthrajhemu.github.io

A *nonpreemptive* schedule is a schedule in which the processing of a task on any processor is not terminated until the task is complete. A schedule for which this need not be true is called *preemptive*. The schedule of Figure 5.22(a) is a preemptive schedule. Figure 5.22(b) shows a nonpreemptive schedule. The *finish time* $f_i(S)$ of job i is the time at which all tasks of job i have been completed in schedule S. In Figure 5.22(a), $f_1(S) = 10$ and $f_2(S) = 12$. In Figure 5.22(b), $f_1(S) = 11$ and $f_2(S) = 5$. The finish time $F(S)$ of a schedule S is given by

$$F(S) = \max_{1 \le i \le n} \{f_i(S)\} \tag{5.22}$$

The *mean flow time* MFT(S) is defined to be

$$\text{MFT}(S) = \frac{1}{n} \sum_{1 \le i \le n} f_i(S) \tag{5.23}$$

An optimal finish time (OFT) schedule for a given set of jobs is a nonpreemptive schedule S for which $F(S)$ is minimum over all nonpreemptive schedules S. A preemptive optimal finish time (POFT) schedule, optimal mean finish time schedule (OMFT), and preemptive optimal mean finish (POMFT) schedule are defined in the obvious way.

Although the general problem of obtaining OFT and POFT schedules for $m > 2$ and of obtaining OMFT schedules is computationally difficult (see Chapter 11), dynamic programming leads to an efficient algorithm to obtain OFT schedules for the case $m = 2$. In this section we consider this special case.

For convenience, we shall use a_i to represent t_{1i}, and b_i to represent t_{2i}. For the two-processor case, one can readily verify that nothing is to be gained by using different processing orders on the two processors (this is not true for $m > 2$). Hence, a schedule is completely specified by providing a permutation of the jobs. Jobs will be executed on each processor in this order. Each task will be started at the earliest possible time. The schedule of Figure 5.23 is completely specified by the permutation (5, 1, 3, 2, 4). We make the simplifying assumption that $a_i \neq 0$, $1 \le i \le n$. Note that if jobs with $a_i = 0$ are allowed, then an optimal schedule can be constructed by first finding an optimal permutation for all jobs with $a_i \neq 0$ and then adding all jobs with $a_i = 0$ (in any order) in front of this permutation (see the exercises).

It is easy to see that an optimal permutation (schedule) has the property that given the first job in the permutation, the remaining permutation is optimal with respect to the state the two processors are in following the completion of the first job. Let $\sigma_1, \sigma_2, \ldots, \sigma_k$ be a permutation prefix defining a schedule for jobs $T_1, T_2, \ldots, T_k$. For this schedule let f_1 and f_2 be the times at which the processing of jobs $T_1, T_2, \ldots, T_k$ is completed on processors P_1

https://hemanthrajhemu.github.io

P_1	a_5	a_1		a_3	a_2	a_4		
P_2		b_5		b_1	b_3	b_2		b_4

Figure 5.23 A schedule

and P_2 respectively. Let $t = f_2 - f_1$. The state of the processors following the sequence of decisions $T_1, T_2, \ldots, T_k$ is completely characterized by t. Let $g(S, t)$ be the length of an optimal schedule for the subset of jobs S under the assumption that processor 2 is not available until time t. The length of an optimal schedule for the job set $\{1, 2, \ldots, n\}$ is $g(\{1, 2, \ldots, n\}, 0)$.

Since the principle of optimality holds, we obtain

$$g(\{1, 2, \cdots, n\}, 0) = \min_{1 \le i \le n} \{a_i + g(\{1, 2, \cdots, n\} - \{i\}, b_i)\} \tag{5.24}$$

Equation 5.24 generalizes to (5.25) for arbitrary S and t. This generalization requires that $g(\phi, t) = \max\{t, 0\}$ and that $a_i \neq 0,\ 1 \le i \le n$.

$$g(S, t) = \min_{i \in S} \ \{a_i + g(S - \{i\}, b_i + \max\{t - a_i, 0\})\} \tag{5.25}$$

The term max $\{t - a_i, 0\}$ comes into (5.25) as task T_{2i} cannot start until $\max\{a_i, t\}$ (P_2 is not available until time t). Hence $f_2 - f_1 = b_i + \max\{a_i, t\} - a_i = b_i + \max\{t - a_i, 0\}$. We can solve for $g(S, t)$ using an approach similar to that used to solve (5.21). However, it turns out that (5.25) can be solved algebraically and a very simple rule to generate an optimal schedule obtained.

Consider any schedule R for a subset of jobs S. Assume that P_2 is not available until time t. Let i and j be the first two jobs in this schedule. Then, from (5.25) we obtain

$$\begin{aligned} g(S, t) &= a_i + g(S - \{i\}, b_i + \max\ \{t - a_i, 0\}) \\ g(S, t) &= a_i + a_j + g(S - \{i, j\}, b_j + \max\ \{b_i + \max\ \{t - a_i, 0\} - a_j, 0\}) \end{aligned} \tag{5.26}$$

https://hemanthrajhemu.github.io

Equation 5.26 can be simplified using the following result:

$$\begin{aligned} t_{ij} &= b_j + \max\ \{b_i + \max\ \{t - a_i, 0\} - a_j, 0\} \\ &= b_j + b_i - a_j + \max\ \{\max\ \{t - a_i, 0\}, a_j - b_i\} \\ &= b_j + b_i - a_j + \max\ \{t - a_i, a_j - b_i, 0\} \\ t_{ij} &= b_j + b_i - a_j - a_i + \max\ \{t, a_i + a_j - b_i, a_i\} \end{aligned} \tag{5.27}$$

If jobs i and j are interchanged in R, then the finish time $g'(S,t)$ is

$$g'(S,t) \quad = \quad a_i + a_j + g(S - \{i,j\}, t_{ji})$$

$$\text{where } t_{ji} \quad = \quad b_j + b_i - a_j - a_i + \max\ \{t, a_i + a_j - b_j, a_j\}$$

Comparing $g(S,t)$ and $g'(S,t)$, we see that if (5.28) below holds, then $g(S,t) \leq g'(S,t)$.

$$\max\ \{t, a_i + a_j - b_i, a_i\} \leq \max\ \{t, a_i + a_j - b_j, a_j\} \tag{5.28}$$

In order for (5.28) to hold for all values of t, we need

$$\max\ \{a_i + a_j - b_i, a_i\} \leq \max\ \{a_i + a_j - b_j, a_j\}$$

$$\text{or} \quad a_i + a_j + \max\ \{-b_i, -a_j\} \leq a_i + a_j + \max\ \{-b_j, -a_i\}$$

$$\text{or} \quad \min\ \{b_i, a_j\} \geq \min\ \{b_j, a_i\} \tag{5.29}$$

From (5.29) we can conclude that there exists an optimal schedule in which for every pair (i,j) of adjacent jobs, $\min\{b_i, a_j\} \geq \min\{b_j, a_i\}$. Exercise 4 shows that all schedules with this property have the same length. Hence, it suffices to generate any schedule for which (5.29) holds for every pair of adjacent jobs. We can obtain a schedule with this property by making the following observations from (5.29). If $\min\{a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n\}$ is a_i, then job i should be the first job in an optimal schedule. If $\min\{a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n\}$ is b_j, then job j should be the last job in an optimal schedule. This enables us to make a decision as to the positioning of one of the n jobs. Equation 5.29 can now be used on the remaining $n-1$ jobs to correctly position another job, and so on. The scheduling rule resulting from (5.29) is therefore:

https://hemanthrajhemu.github.io

1. Sort all the a_i's and b_j's into nondecreasing order.

2. Consider this sequence in this order. If the next number in the sequence is a_j and job j hasn't yet been scheduled, schedule job j at the leftmost available spot. If the next number is b_j and job j hasn't yet been scheduled, schedule job j at the rightmost available spot. If j has already been scheduled, go to the next number in the sequence.

Note that the above rule also correctly positions jobs with $a_i = 0$. Hence, these jobs need not be considered separately.

Example 5.28 Let $n = 4, (a_1, a_2, a_3, a_4) = (3, 4, 8, 10)$, and $(b_1, b_2, b_3, b_4) = (6, 2, 9, 15)$. The sorted sequence of a's and b's is $(b_2, a_1, a_2, b_1, a_3, b_3, a_4, b_4) = (2, 3, 4, 6, 8, 9, 10, 15)$. Let $\sigma_1, \sigma_2, \sigma_3$, and σ_4 be the optimal schedule. Since the smallest number is b_2, we set $\sigma_4 = 2$. The next number is a_1 and we set $\sigma_1 = a_1$. The next smallest number is a_2. Job 2 has already been scheduled. The next number is b_1. Job 1 has already been scheduled. The next is a_3 and we set σ_3. This leaves σ_3 free and job 4 unscheduled. Thus, $\sigma_3 = 4$. □

The scheduling rule above can be implemented to run in time $O(n \log n)$ (see exercises). Solving (5.24) and (5.25) directly for $g(1, 2, \ldots, n, 0)$ for the optimal schedule will take $\Omega(2^n)$ time as there are these many different S's for which $g(S, t)$ will be computed.

EXERCISES

1. N jobs are to be processed. Two machines A and B are available. If job i is processed on machine A, then a_i units of processing time are needed. If it is processed on machine B, then b_i units of processing time are needed. Because of the peculiarities of the jobs and the machines, it is quite possible that $a_i \geq b_i$ for some i while $a_j < b_j$ for some j, $j \neq i$. Obtain a dynamic programming formulation to determine the minimum time needed to process all the jobs. Note that jobs cannot be split between machines. Indicate how you would go about solving the recurrence relation obtained. Do this on an example of your choice. Also indicate how you would determine an optimal assignment of jobs to machines.

2. N jobs have to be scheduled for processing on one machine. Associated with job i is a 3-tuple (p_i, t_i, d_i). The variable t_i is the processing time needed to complete job i. If job i is completed by its deadline d_i, then a profit p_i is earned. If not, then nothing is earned. From Section 4.4 we know that J is a subset of jobs that can all be completed by their

https://hemanthrajhemu.github.io

deadlines iff the jobs in J can be processed in nondecreasing order of deadlines without violating any deadline. Assume $d_i \leq d_{i+1}$, $1 \leq i < n$. Let $f_i(x)$ be the maximum profit that can be earned from a subset J of jobs when $n = i$. Here $f_n(d_n)$ is the value of an optimal selection of jobs J. Let $f_0(x) = 0$. Show that for $x \leq t_i$,

$$f_i(x) = \max\ \{f_{i-1}(x),\ f_{i-1}(x - t_i) + p_i\}$$

3. Let I be any instance of the two-processor flow shop problem.
   (a) Show that the length of every POFT schedule for I is the same as the length of every OFT schedule for I. Hence, the algorithm of Section 5.10 also generates a POFT schedule.
   (b) Show that there exists an OFT schedule for I in which jobs are processed in the same order on both processors.
   (c) Show that there exists an OFT schedule for I defined by some permutation σ of the jobs (see part (b)) such that all jobs with $a_i = 0$ are at the front of this permutation. Further, show that the order in which these jobs appear at the front of the permutation is not important.
4. Let I be any instance of the two-processor flow shop problem. Let $\sigma = \sigma_1\sigma_2 \cdots \sigma_n$ be a permutation defining an OFT schedule for I.
   (a) Use (5.29) to argue that there exists an OFT σ such that $\min\ \{b_i, a_j) \geq \min\ \{b_j, a_i\}$ for every i and j such that $i = \sigma_k$ and $j = \sigma_{k+1}$ (that is, i and j are adjacent).
   (b) For a σ satisfying the conditions of part (a), show that $\min\{b_i, a_j\} \geq \min\{b_j, a_i\}$ for every i and j such that $i = \sigma_k$ and $j = \sigma_r, k < r$.
   (c) Show that all schedules corresponding to σ's satisfying the conditions of part (a) have the same finish time. (*Hint*: use part (b) to transform one of two different schedules satisfying (a) into the other without increasing the finish time.)

5.11 REFERENCES AND READINGS

Two classic references on dynamic programming are:

Introduction to Dynamic Programming, by G. Nemhauser, John Wiley and Sons, 1966.

Applied Dynamic Programming by R. E. Bellman and S. E. Dreyfus, Princeton University Press, 1962.

https://hemanthrajhemu.github.io

See also *Dynamic Programming,* by E. V. Denardo, Prentice-Hall, 1982.

The dynamic programming formulation for the shortest-paths problem was given by R. Floyd.

Bellman and Ford's algorithm for the single-source shortest-path problem (with general edge weights) can be found in *Dynamic Programming* by R. E. Bellman, Princeton University Press, 1957.

The construction of optimal binary search trees using dynamic programming is described in *The Art of Programming: Sorting and Searching*, Vol. 3, by D. E. Knuth, Addison Wesley, 1973.

The string editing algorithm discussed in this chapter is in "The string-to-string correction problem," by R. A. Wagner and M. J. Fischer, *Journal of the ACM* 21, no. 1 (1974): 168–173.

The set generation approach to solving the 0/1 knapsack problem was formulated by G. Nemhauser and Z. Ullman, and E. Horowitz and S. Sahni.

Exercise 6 in Section 5.7 is due to E. Horowitz and S. Sahni.

The dynamic programming formulation for the traveling salesperson problem was given by M. Held and R. Karp.

The dynamic programming solution to the matrix product chain problem (Exercises 1 and 2 in Additional Exercises) is due to S. Godbole.

5.12 ADDITIONAL EXERCISES

1. [Matrix product chains] Let A, B, and C be three matrices such that $C = A \times B$. Let the dimensions of A, B, and C respectively be $m \times n, n \times p$, and $m \times p$. From the definition of matrix multiplication,

$$C(i,j) = \sum_{k=1}^{n} A(i,k)B(k,j)$$

   (a) Write an algorithm to compute C directly using the above formula. Show that the number of multiplications needed by your algorithm is mnp.
   (b) Let $M_1 \times M_2 \times \cdots \times M_r$ be a chain of matrix products. This chain may be evaluated in several different ways. Two possibilities are $(\cdots((M_1 \times M_2) \times M_3) \times M_4) \times \cdots) \times M_r$ and $(M_1 \times (M_2 \times (\cdots \times (M_{r-1} \times M_r)\cdots)$. The cost of any computation of $M_1 \times$

https://hemanthrajhemu.github.io

$M_2 \times \cdots \times M_r$ is the number of multiplications used. Consider the case $r = 4$ and matrices M_1 through M_4 with dimensions $100 \times 1, 1 \times 100, 100 \times 1$, and $1\times$ 100 respectively. What is the cost of each of the five ways to compute $M_1 \times M_2 \times M_3 \times M_4$? Show that the optimal way has a cost of 10,200 and the worst way has a cost of 1,020,000. Assume that all matrix products are computed using the algorithm of part (a).

(c) Let M_{ij} denote the matrix product $M_i \times M_{i+1} \times \cdots \times M_j$. Thus, $M_{ii} = M_i$, $1 \leq i \leq r$. $S = P_1, P_2, \ldots, P_{r-1}$ is a *product sequence* computing M_{1r} iff each product P_k is of the form $M_{ij} \times M_{j+1,q}$, where M_{ij} and $M_{j+1,q}$ have been computed either by an earlier product $P_l, l < k$, or represent an input matrix M_{tt}. Note that $M_{ij} \times M_{j+1,q} = M_{iq}$. Also note that every valid computation of M_{1r} using only pairwise matrix products at each step is defined by a product sequence. Two product sequences $S_1 = P_1, P_2, \ldots, P_{r-1}$ and $S_2 = U_1, U_2, \ldots, U_{r-1}$ are *different* if $P_i \neq U_i$ for some i. Show that the number of different product sequences if $(r-1)!$

(d) Although there are $(r-1)!$ different product sequences, many of these are essentially the same in the sense that the same pairs of matrices are multiplied. For example, the sequences $S_1 = (M_1 \times M_2), (M_3 \times M_4), (M_{12} \times M_{34})$ and $S_2 = (M_3 \times M_4), (M_1 \times M_2), (M_{12} \times M_{34})$ are different under the definition of part (c). However, the same pairs of matrices are multiplied in both S_1 and S_2. Show that if we consider only those product sequences that differ from each other in at least one matrix product, then the number of different sequences is equal to the number of different binary trees having exactly $r-1$ nodes.

(e) Show that the number of different binary trees with n nodes is

$$\frac{1}{n+1}\binom{2n}{n}$$

2. [Matrix product chains] In the preceding exercise it was established that the number of different ways to evaluate a matrix product chain is very large even when r is relatively small (say 10 or 20). In this exercise we shall develop an $O(r^3)$ algorithm to find an optimal product sequence (that is, one of minimum cost). Let $D(i), 0 \leq i \leq r$, represent the dimensions of the matrices; that is, M_i has $D(i-1)$ rows and $D(i)$ columns. Let $C(i,j)$ be the cost of computing M_{ij} using an optimal product sequence for M_{ij}. Observe that $C(i,i) = 0, 1 \leq i \leq r$, and that $C(i, i+1) = D(i-1)D(i)D(i+1), 1 \leq i \leq r$.

https://hemanthrajhemu.github.io

(a) Obtain a recurrence relation for $C(i,j), j > i$. This recurrence relation will be similar to Equation 5.14.

(b) Write an algorithm to solve the recurrence relation of part (a) for $C(1,r)$. Your algorithm should be of complexity $O(r^3)$.

(c) What changes are needed in the algorithm of part (b) to determine an optimal product sequence. Write an algorithm to determine such a sequence. Show that the overall complexity of your algorithm remains $O(r^3)$.

(d) Work through your algorithm (by hand) for the product chain of part (b) of the previous exercise. What are the values of $C(i,j), 1 \le i \le r$ and $j \ge i$? What is an optimal way to compute M_{14}?

3. There are two warehouses W_1 and W_2 from which supplies are to be shipped to destinations $D_i, 1 \le i \le n$. Let d_i be the demand at D_i and let r_i be the inventory at W_i. Assume $r_1 + r_2 = \sum d_i$. Let $c_{ij}(x_{ij})$ be the cost of shipping x_{ij} units from warehouse W_i to destination D_j. The warehouse problem is to find nonnegative integers $x_{ij}, 1 \le i \le 2$ and $1 \le j \le n$, such that $x_{1j} + x_{2j} = d_j$, $1 \le j \le n$, and $\sum_{i,j} c_{ij}(x_{ij})$ is minimized. Let $g_i(x)$ be the cost incurred when W_1 has an inventory of x and supplies are sent to $D_j, 1 \le j \le i$, in an optimal manner (the inventory at W_2 is $\sum_{1 \le j \le i} d_j - x$). The cost of an optimal solution to the warehouse problem is $g_n(r_1)$.

   (a) Use the optimality principle to obtain a recurrence relation for $g_i(x)$.

   (b) Write an algorithm to solve this recurrence and obtain an optimal sequence of values for $x_{ij}, 1 \le i \le 2, 1 \le j \le n$.

4. Given a warehouse with a storage capacity of B units and an initial stock of v units, let y_i be the quantity sold in each month, $i, 1 \le i \le n$. Let P_i be the per-unit selling price in month i, and x_i the quantity purchased in month i. The buying price is c_i per unit. At the end of each month, the stock in hand must be no more than B. That is,

$$v + \sum_{1 \le i \le j} (x_i - y_i) \le B, \quad 1 \le j \le n$$

   The amount sold in each month cannot be more than the stock at the end of the previous month (new stock arrives only at the end of a month). That is,

$$y_i \le v + \sum_{1 \le j < i} (x_j - y_j), \quad 1 \le i \le n$$

https://hemanthrajhemu.github.io

Also, we require x_i and y_i to be nonnegative integers. The total profit derived is

$$P_n = \sum_{j=1}^{n}(p_j y_j - c_j x_j)$$

The problem is to determine x_j and y_j such that P_n is maximized. Let $f_i(v_i)$ represent the maximum profit that can be earned in months $i+1, i+2, \ldots, n$, starting with v_i units of stock at the end of month i. Then $f_0(v)$ is the maximum value of P_n.

(a) Obtain the dynamic programming recurrence for $f_i(v_i)$ in terms of $f_{i+1}(v_i)$.

(b) What is $f_n(v_i)$?

(c) Solve part (a) analytically to obtain the formula

$$f_i(v_i) = a_i x_i + b_i v_i$$

for some constants a_i and b_i.

(d) Show that an optimal P_n is obtained by using the following strategy:

  i. $p_i \geq c_i$
    A. If $b_{i+1} \geq c_i$, then $y_i = v_i$ and $x_i = B$.
    B. If $b_{i+1} \leq c_i$, then $y_i = v_i$ and $x_i = 0$.

  ii. $c_i \geq p_i$
    A. If $b_{i+1} \geq c_i$, then $y_i = 0$ and $x_i = B - v_i$.
    B. If $b_{i+1} \leq p_i$, then $y_i = v_i$ and $x_i = 0$.
    C. If $p_i \leq b_{i+1} \leq c_i$, then $y_i = 0$ and $x_i = 0$.

(e) Use the p_i and c_i in Figure 5.24 and obtain an optimal decision sequence from part (d).

i	1	2	3	4	5	6	7	8
p_i	8	8	2	3	4	3	2	5
c_i	3	6	7	1	4	5	1	3

Figure 5.24 p_i and c_i for Exercise 4

Assume the warehouse capacity to be 100 and the initial stock to be 60.

https://hemanthrajhemu.github.io

(f) From part (d) conclude that an optimal set of values for x_i and y_i will always lead to the following policy: Do no buying or selling for the first k months (k may be zero) and then oscillate between a full and an empty warehouse for the remaining months.

5. Assume that n programs are to be stored on two tapes. Let l_i be the length of tape needed to store the ith program. Assume that $\sum l_i \leq L$, where L is the length of each tape. A program can be stored on either of the two tapes. If S_1 is the set of programs on tape 1, then the worst-case access time for a program is proportional to $\max\{\sum_{i \in S_1} l_i, \sum_{i \notin S_1} l_i\}$. An optimal assignment of programs to tapes minimizes the worst-case access times. Formulate a dynamic programming approach to determine the worst-case access time of an optimal assignment. Write an algorithm to determine this time. What is the complexity of your algorithm?

6. Redo Exercise 5 making the assumption that programs will be stored on tape 2 using a different tape density than that used on tape 1. If l_i is the tape length needed by program i when stored on tape 1, then al_i is the tape length needed on tape 2.

7. Let L be an array of n distinct integers. Give an efficient algorithm to find the length of a longest increasing subsequence of entries in L. For example, if the entries are $11, 17, 5, 8, 6, 4, 7, 12, 3$, a longest increasing subsequence is $5, 6, 7, 12$. What is the run time of your algorithm?

https://hemanthrajhemu.github.io