Chapter 1

ELEMENTARY PATH PROBLEMS

1. Introduction

Dynamic programming is an optimization procedure that is particularly applicable to problems requiring a sequence of interrelated decisions. Each decision transforms the current situation into a new situation. A sequence of decisions, which in turn yields a sequence of situations, is sought that maximizes (or minimizes) some measure of value. The value of a sequence of decisions is generally equal to the sum of the values of the individual decisions and situations in the sequence.

Through the study of a wide variety of examples we hope the reader will develop the largely intuitive skill for recognizing problems fitting the above very general description. We begin with a problem seeking the best path from one physical location to another. Then we elaborate the problem to show how a "situation" may encompass more than just information about location and must be defined in a way that is appropriate to each particular problem.

2. A Simple Path Problem

Suppose for the moment that you live in a city whose streets are laid out as shown in Figure 1.1, that all streets are one-way, and that the numbers shown on the map represent the effort (usually time but sometimes cost or distance) required to traverse each individual block. You live at A and wish to get to B with minimum total effort. (In Chapter 4, you will learn how to find minimum-effort paths through more realistic cities.)

You could, of course, solve this problem by enumerating all possible paths from A to B; adding up the efforts, block by block, of each; and then choosing the smallest such sum. There are 20 distinct paths from A to

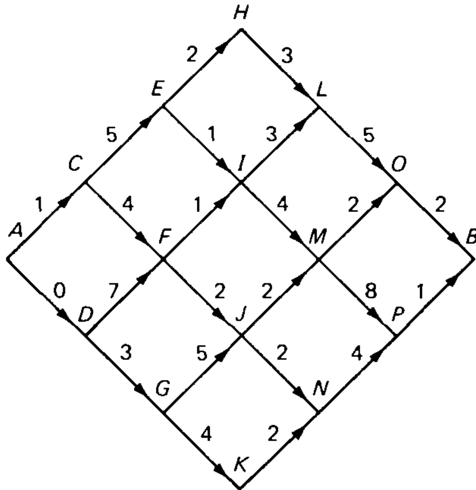


Figure 1.1

B and five additions yield the sum of the six numbers along a particular path, so 100 additions would yield the 20 path sums to be compared. Since one comparison yields the smaller of two numbers, one additional comparison (of that number with a third) yields the smallest of three, etc., 19 comparisons complete this enumerative solution of the problem. As you might suspect, one can solve this problem more efficiently than by brute-force enumeration. This more efficient method is called dynamic programming.

3. The Dynamic-Programming Solution

To develop the dynamic-programming approach, one reasons as follows. I do not know whether to go diagonally upward or diagonally downward from A, but if I somehow knew just two additional numbers—namely, the total effort required to get from C to B by the best (i.e., minimum-effort) path and the total effort required to get from D to B by the best path—I could make the best choice at A. Denoting the minimum effort from C to B by S_C and the minimum effort from D to B by S_D, I would add to S_C the effort required in going from A to C, obtaining the effort required on the best path starting diagonally upward from A. I would then add the effort on AD to S_D to obtain the effort on the best path starting diagonally downward from A, and I would compare these two sums to find the overall minimum effort and the best first decision.

Of course, all this is predicated on knowing the two numbers S_C and S_D which, unfortunately, are not yet known. However, one of the two key ideas of dynamic programming has already made its innocuous appearance. This is the observation that only the efforts along the best

paths from C and from D to B are relevant to the above computation, and the efforts along the nine inferior paths from each of C and D to B need never be computed. This observation is often called the *principle of optimality* and is stated as follows:

The best path from A to B has the property that, whatever the initial decision at A, the remaining path to B, starting from the next point after A, must be the best path from that point to B.

Having defined S_C and S_D as above, we can cite the principle of optimality as the justification for the formula

$$S_A = \min\begin{bmatrix} 1 + S_C \\ 0 + S_D \end{bmatrix},$$

where S_A is the minimum effort to get from A to B and the symbol $\min[^x_y]$ means "the smaller of the quantities x and y." In the future we shall always cite the principle rather than repeat the above verbal reasoning.

Now for the second key idea. While the two numbers S_C and S_D are unknown to us initially, we could compute S_C if we knew the two numbers S_E and S_F (the minimum efforts from E and F to B, respectively) by invoking the principle of optimality to write

$$S_C = \min\begin{bmatrix} 5 + S_E \\ 4 + S_F \end{bmatrix}.$$

Likewise,

$$S_D = \min\begin{bmatrix} 7 + S_F \\ 3 + S_G \end{bmatrix}.$$

S_E, S_F, and S_G are at first not known, but they could be computed if S_H, S_I, S_J, and S_K were available. These numbers, in turn, depend on S_L, S_M, and S_N, which themselves depend on S_O and S_P. Hence we could use formulas of the above type to compute all the S's if we knew S_O and S_P, the minimum efforts from O and P, respectively, to B. But these numbers are trivially known to be 2 and 1, respectively, since O and P are so close to B that only one path exists from each point. Working our way backward from O and P to A, we now carry out the desired computations:

$$S_L = 5 + S_O = 7, \qquad S_M = \min\begin{bmatrix} 2 + S_O \\ 8 + S_P \end{bmatrix} = 4, \qquad S_N = 4 + S_P = 5;$$

$$S_H = 3 + S_L = 10, \qquad S_I = \min\begin{bmatrix} 3 + S_L \\ 4 + S_M \end{bmatrix} = 8, \qquad S_J = \min\begin{bmatrix} 2 + S_M \\ 2 + S_N \end{bmatrix} = 6,$$

$$S_K = 2 + S_N = 7;$$

(equations continue)

$$S_E = \min\begin{bmatrix} 2 + S_H \\ 1 + S_I \end{bmatrix} = 9, \qquad S_F = \min\begin{bmatrix} 1 + S_I \\ 2 + S_J \end{bmatrix} = 8,$$

$$S_G = \min\begin{bmatrix} 5 + S_J \\ 4 + S_K \end{bmatrix} = 11;$$

$$S_C = \min\begin{bmatrix} 5 + S_E \\ 4 + S_F \end{bmatrix} = 12, \qquad S_D = \min\begin{bmatrix} 7 + S_F \\ 3 + S_G \end{bmatrix} = 14;$$

$$S_A = \min\begin{bmatrix} 1 + S_C \\ 0 + S_D \end{bmatrix} = 13.$$

Our second key idea has been to compute lengths of the needed minimum-effort paths by considering starting points further and further away from B, finally working our way back to A. Then the numbers required by idea one, the principle of optimality, are known when they are needed.

In order to establish that the best path has total effort 13 (i.e., that $S_A = 13$), we performed one addition at each of the six points H, L, O, K, N, and P where only one decision was possible and we performed two additions and a comparison at each of the remaining nine points where two initial decisions were possible. This sums to 24 additions and nine comparisons, compared with 100 additions and 19 comparisons for the brute-force enumeration described earlier.

Of course we are at least as interested in actually finding the best path as we are in knowing its total effort. The path would be easy to obtain had we noted which of the two possible first decisions yielded the minimum in our previous calculations at each point on the figure. If we let x represent any particular starting point, and denote by P_x the node after node x on the optimal path from x to B, then the P table could have been computed as we computed the S table above. For example, $P_M = O$ since $2 + S_O$ was smaller than $8 + S_P$, $P_I = M$ since $4 + S_M$ was smaller than $3 + S_L$, etc. The P table, which can be deduced with no further computations as the S table is developed, is given in Table 1.1. To use this table to find the best path from A to B we note that $P_A = C$, so we move from A to C. Now, since $P_C = F$, we continue on to F; $P_F = J$ means we move next to J;

Table 1.1 *The optimal next point for each initial point*

$P_O = B$,	$P_P = B$;		
$P_L = O$,	$P_M = O$,	$P_N = P$;	
$P_H = L$,	$P_I = M$,	$P_J = M$,	$P_K = N$;
$P_E = I$,	$P_F = J$,	$P_G = J$ or K;	
$P_C = F$,	$P_D = G$;		
$P_A = C$.			

$P_J = M$ sends us on to M where $P_M = O$ tells us O is next and B is last. The best path is therefore A–C–F–J–M–O–B. As a check on the accuracy of our calculations we add the six efforts along this path obtaining $1 + 4 + 2 + 2 + 2 + 2 = 13$ which equals S_A, as it must if we have made no numerical errors.

It may surprise the reader to hear that there are no further key ideas in dynamic programming. Naturally, there are special tricks for special problems, and various uses (both analytical and computational) of the foregoing two ideas, but the remainder of the book and of the subject is concerned only with how and when to use these ideas and not with new principles or profound insights. What is common to all dynamic-programming procedures is exactly what we have applied to our example: first, the recognition that a given "whole problem" can be solved if the values of the best solutions of certain subproblems can be determined (the principle of optimality); and secondly, the realization that if one starts at or near the end of the "whole problem," the subproblems are so simple as to have trivial solutions.

4. Terminology

To clarify our explanations of various elaborations and extensions of the above ideas, let us define some terms and develop some notations. We shall call the rule that assigns values to various subproblems the *optimal value function*. The function S is the optimal value (here minimum-effort) function in our example. The subscript of S—e.g., the A in the symbol S_A—is the *argument* of the function S, and each argument refers to a particular subproblem. By our definition of S, the subscript A indicates that the best path from A to B is desired, while C means that the best path from C to B is sought. The rule that associates the best first decision with each subproblem—the function P in our example—is called the *optimal policy function*. The principle of optimality yields a formula or set of formulas relating various values of S. This formula is called a *recurrence relation*. Finally, the value of the optimal value function S for certain arguments is assumed obvious from the statement of the problem and from the definition of S with no computation required. These obvious values are called the *boundary conditions* on S.

In this jargon, to solve a problem by means of dynamic programming we choose the arguments of the optimal value function and define that function in such a way as to allow the use of the principle of optimality to write a recurrence relation. Starting with the boundary conditions, we then use the recurrence relation to determine concurrently the optimal value

and policy functions. When the optimal value and decision are known for the value of the argument that represents the original whole problem, the solution is completed and the best path can be traced out using the optimal policy function alone.

We now develop a particular notation for the simple path problem at hand which will allow for a more systematic representation of the procedure than did our earlier formulas. We say nothing new. We shall only say it in a different way. Let us place our city map (Figure 1.1) on a coordinate system as shown in Figure 1.2. Now the point A has coordinates (0, 0), B has coordinates (6, 0), I has (3, 1), etc.

We do not show the one-way arrows on the lines, but we assume throughout the remainder of this chapter that admissible paths are always continuous and always move toward the right.

The optimal value function S is now a function of the pair of numbers (x, y) denoting a starting point, rather than a function of a literal argument such as A or C. For those pairs (x, y) denoting a street intersection on our map (henceforth called a *vertex* of our *network*) we define the optimal value function $S(x, y)$ by

$$S(x, y) = \text{the value of the minimum-effort path connecting the vertex } (x, y) \text{ with the terminal vertex } (6, 0). \tag{1.1}$$

The diagonal straight line connecting one vertex of our network and a neighboring one represents a block of our city and will now be called an *arc* of our network. We let the symbol $a_u(x, y)$ denote the effort associated with the arc connecting the vertex (x, y) with the vertex $(x + 1, y + 1)$; the subscript u signifies the arc goes diagonally *up* from (x, y). We let $a_d(x, y)$ denote the effort of the arc going diagonally *down* from (x, y) to $(x + 1,$

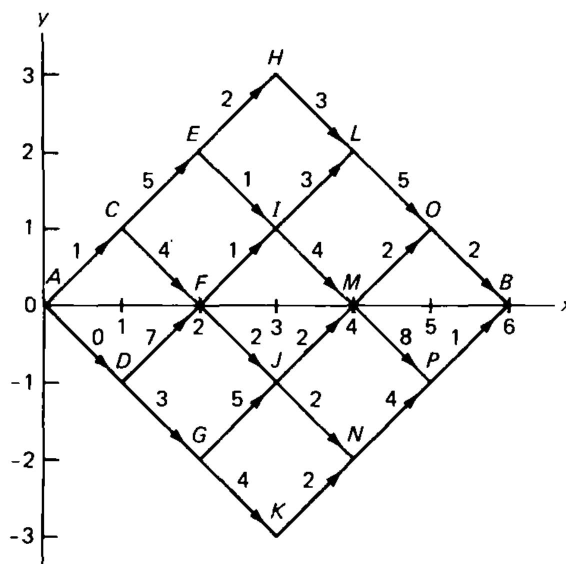


Figure 1.2

$y-1)$, and we say that $a_u(x, y)$ or $a_d(x, y) = \infty$ (a very large number) if there is no such arc in our network (e.g., $a_u(4, 2) = \infty$).

In terms of these symbols, the principle of optimality gives the recurrence relation

$$S(x, y) = \min\begin{bmatrix} a_u(x, y) + S(x+1, y+1) \\ a_d(x, y) + S(x+1, y-1) \end{bmatrix} \tag{1.2}$$

and the obvious boundary condition is

$$S(6, 0) = 0, \tag{1.3}$$

since the effort in going *from* (6, 0) *to* (6, 0) is zero for we are already there. Alternatively, we could write the equally obvious boundary conditions $S(5, 1) = 2$, $S(5, -1) = 1$ as we did earlier, but these are implied by (1.2), (1.3), and our convention that $a_u(5, 1) = \infty$ and $a_d(5, -1) = \infty$. Furthermore, (1.3) is simpler to write. Either boundary condition is correct.

In the exercises assigned during this and subsequent chapters, when we use a phrase such as, "Give the dynamic-programming formulation of this problem," we shall mean:

(1) Define the appropriate optimal value function, including both a specific definition of its arguments and the meaning of the value of the function (e.g., (1.1)).

(2) Write the appropriate recurrence relation (e.g., (1.2)).

(3) Note the appropriate boundary conditions (e.g., (1.3)).

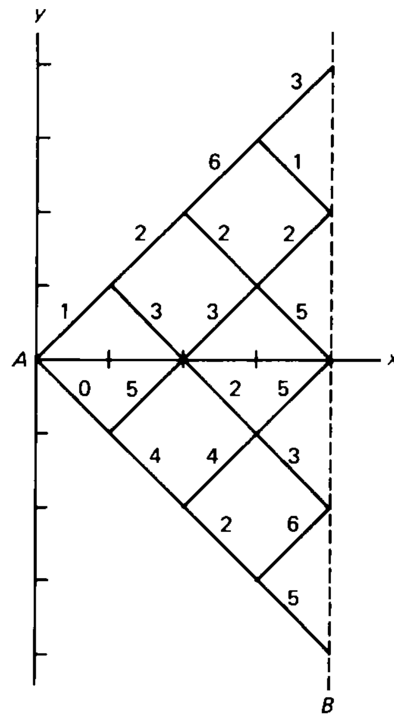


Figure 1.3

As we emphasized in the Preface, the *art* of dynamic-programming formulation can be mastered only through practice, and it is absolutely essential that the reader attempt almost all of the assigned problems. Furthermore, the student should understand the correct solution given in the back of this book before continuing.

Problem 1.1. On the network shown in Figure 1.3, we seek that path connecting A with any point on line B which minimizes the sum of the four arc numbers encountered along the path. (There are 16 admissible paths.) Give the dynamic-programming formulation of this problem.

Problem 1.2. Solve the above problem using dynamic programming, with the additional specification that there is a rebate associated with each terminal point; ending at the point (4, 4) has a cost of -2 (i.e., 2 is subtracted from the path cost), (4, 2) has cost -1, (4, 0) has cost -3, (4, -2) has cost -4, and (4, -4) has cost -3.

5. Computational Efficiency

Before proceeding, let us pause to examine the efficiency of the dynamic-programming approach to the minimum-effort path problems we have considered. Let us first ask approximately how many additions and comparisons are required to solve a problem on a network of the type first considered, an example of which is shown in Figure 1.4 (without specifying the arc costs and without arrows indicating that, as before, all arcs are directed diagonally to the right). First we note that in the problem represented in Figure 1.1 each admissible path contained six arcs, whereas in the one given in Figure 1.4 each path has 10. We call the former a six-stage problem, the one in Figure 1.4 a 10-stage problem, and we shall now analyze an N-stage problem for N an even integer. There are N vertices (those on the lines CB and DB, excluding B, in Figure 1.4) at

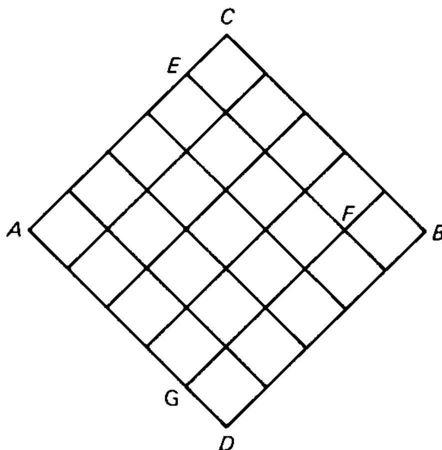


Figure 1.4

which one addition and no comparisons are required in the dynamic-programming solution. There are $(N/2)^2$ remaining vertices (those in the diamond $AEFG$) at which two additions and a comparison are required. Hence a total of $N^2/2 + N$ additions and $N^2/4$ comparisons are needed for the dynamic-programming solution. (Note that for $N = 6$, the first problem in the text, these formulas yield 24 additions and nine comparisons, which checks with the count that we performed earlier.)

When we ask, in future problems, how many additions and how many comparisons are required for the dynamic-programming solution, we expect the reader to do roughly what we did above—imagine that the calculations are really being performed, count the points (or situations) that must be considered, count the additions and comparisons required at each such point (taking account of perhaps varying calculations at differing points), and total the computations.

To get an idea of how much more efficient dynamic programming is than what we called earlier brute-force enumeration, let us consider enumeration for an N-stage problem. There are $\binom{N}{N/2}$ admissible paths. (The symbol $\binom{X}{Y}$ should be read, "The number of different ways of choosing a set of Y items out of a total of X distinguishable items" and $\binom{X}{Y} = X!/[Y!(X-Y)!]$ where $z! = 1 \cdot 2 \cdot 3 \cdots z$.) To derive this formula for the number of paths we note that each path can be represented by a sequence of N symbols, half of which are U's and half of which are D's, where a U in the Kth position in the sequence means the Kth step is diagonally *up* and a D means the Kth step is diagonally *down*. Then $\binom{N}{N/2}$ is the number of different ways of choosing the $N/2$ steps that are U, with, of course, the remainder being D's. Note that the formula gives the correct number, 20, for our original six-stage example. Each path requires $N-1$ additions, and all but the first one evaluated require a comparison in order to find the best path. This totals to $(N-1)\binom{N}{N/2}$ additions and $\binom{N}{N/2} - 1$ comparisons. For $N = 6$ we have already seen that dynamic programming required roughly one-fourth the computation of brute-force enumeration. However, for $N = 20$ we find that the dynamic-programming solution involves 220 additions and 100 comparisons, while enumeration requires more than three million additions and some 184,000 comparisons. We shall find in general that the larger the problem, the more impressive the computational advantage of dynamic programming.

Problem 1.3. How many additions and how many comparisons are required in the dynamic-programming solution and in brute-force enumeration for an N-stage problem involving a network of the type shown in Figure 1.3? Evaluate your formulas for $N = 20$.

Let us note here a further advantage of dynamic programming. Once

a problem has been solved by the computational scheme that we have been using, which works backward from the terminal point or points, one also has solved a variety of other problems. One knows, in our examples, the best paths from each vertex to the end. In our initial example of this chapter, referral to the policy Table 1.1 of Section 3 tells us that the best path from D to B goes first to vertex G, then to J or K, and, if J is chosen, the remaining vertices are M, O, and B.

6. Forward Dynamic Programming

We now explicate a variation on the above dynamic-programming procedure which is equally efficient but which yields solutions to slightly different but related problems. In a sense we reverse all of our original thinking. First we note that we could easily determine the effort of the best path from A to B in Figure 1.1 if we knew the effort of both the best path from A to O and the best path from A to P. Furthermore, we would know these two numbers if we knew the efforts of the best paths to each of L, M, and N from A, etc. This leads us to define a new optimal value function S by

$$S(x, y) = \text{the value of the minimum-effort path connecting the initial vertex } (0, 0) \text{ with the vertex } (x, y). \tag{1.4}$$

Note that this is a quite different function from the S defined and computed previously; however, the reader should not be disturbed by our use of the same symbol S as long as each use is clearly defined. There are far more functions in the world than letters, and surely the reader has let $f(x) = x$ for one problem and $f(x) = x^2$ for the next.

The appropriate recurrence relation for our new optimal value function is

$$S(x, y) = \min \begin{bmatrix} a_u(x-1, y-1) + S(x-1, y-1) \\ a_d(x-1, y+1) + S(x-1, y+1) \end{bmatrix}. \tag{1.5}$$

Here we are using a reversed version of the principle of optimality, which can be stated as follows:

The best path from A to any particular vertex B has the property that whatever the vertex before B, call it C, the path must be the best path from A to C.

The boundary condition is

$$S(0, 0) = 0 \tag{1.6}$$

since the cost of the best path from A to itself is zero.

Problem 1.4. Solve the problem in Figure 1.1 by using (1.4)–(1.6). How many additions and how many comparisons does the solution entail? How does this compare to the numbers for the original dynamic-programming solution in the text?

We shall call the procedure using the new reversed viewpoint the *forward* dynamic-programming procedure since the computation is performed by moving forward from A to B rather than moving *backward* from B to A as in the original (backward) procedure.

The two procedures differ in the auxiliary information they produce. The forward procedure used in Problem 1.4 yields the optimal path *from A* to every vertex, but tells nothing about the optimal paths from most vertices to B. The latter information is furnished by the backward procedure.

Problem 1.5. Do Problem 1.1 by the forward procedure. Compute and compare the number of additions and comparisons for the backward and the forward solutions.

Problem 1.6. Find the minimum-cost path starting from line A and going to line B in the network shown in Figure 1.5. The numbers shown along lines A and B are additional costs associated with various initial and terminal points.

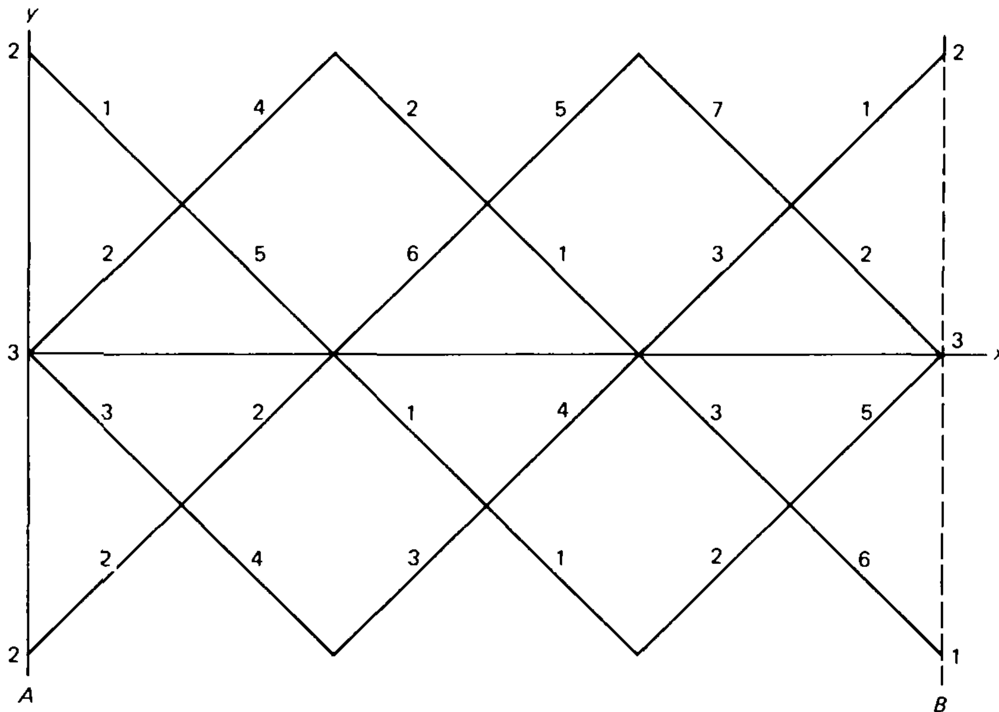
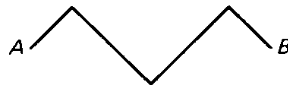


Figure 1.5

7. A More Complicated Example

We now complicate the situation to show that for some problems the argument of the optimal value function may have to contain more information than just the current vertex. A major part of the art of dynamic programming is the appropriate choice of the subproblems that must be solved in order eventually to solve the given problem.

Let us stipulate that if, after arriving at any vertex on our way from A to B in Figure 1.1, we turn rather than continue in a straight line, an additional cost of 3 is assessed. No penalty is assessed if we continue straight on. The path that solved the original problem before the turn-penalty charge was introduced looked like



and hence had three turns. Therefore, that path costs 22 (13 plus 3 for each of three turns) and may no longer be optimal since there may have been paths originally costing less than 19 and containing one turn.

Suppose, just as we did at the start of this chapter, that we are at the initial vertex A and we already know the cost (including the turn penalties) of the best path from C to B and also from D to B. Can we then compute the optimal decision at A? If we move diagonally upward from A to C, the arc cost is 1, but what is the remaining cost? It is clearly the (assumed known) cost of going from C to B if it is optimal, in going from C to B, to go diagonally upward at C (i.e., continue straight at C rather than turn). But what if the optimal decision at C, in order to minimize the cost including turn penalties of going from C to B, is to go diagonally down at C? Then if we first go from A to C and then follow this path the total cost is the cost of the arc AC plus a cost of 3 for the turn plus the (assumed known) optimal cost from C to B. But, in this case, there is another possibility that we must consider. Maybe, if we were to go diagonally upward to C, we should continue diagonally upward at C and avoid the turn penalty cost of 3. If the minimum-cost path from C to B that starts diagonally upward at C costs only 1 or 2 more than the optimal path from C to B (which we assumed went diagonally down at C), using this path would be preferable to turning at C and going diagonally down, even though the optimal path from C to B goes diagonally down at C. So what we need to know at C is both the minimal cost from C to B, including any turn penalties incurred after C, if we leave C in the diagonally upward direction and also the minimal cost should we choose to leave C in the diagonally downward direction. That is, we need two numbers associated with the vertex C. Let us now apply these insights to solve the problem. We define the optimal value function S, which in this case is a function of

three variables (two describe the vertex and one, which can take on only two possible values, tells us whether we are to leave the vertex by going diagonally up or by going diagonally down), by

$S(x, y, z)$ = the minimum attainable sum of arc numbers plus turn penalties if we start at the vertex (x, y), go to B, and move initially in the direction indicated by z, where z equals 0 denotes diagonally upward and z equals 1 denotes diagonally downward. (1.7)

Now, supposing we are at (x, y) and move upward ($z = 0$), we arrive at $(x + 1, y + 1)$ moving in an upward direction. If we continue to move upward, we incur an additional cost $S(x + 1, y + 1, 0)$, but should we turn and proceed downward instead the cost is 3 (the penalty for the turn) plus $S(x + 1, y + 1, 1)$, the remaining cost once we have started downward. The optimal cost from (x, y) starting upward is the minimum of these two alternatives. Hence, by the principle of optimality, we obtain the recurrence relation

$$S(x, y, 0) = \min \begin{bmatrix} a_{\mathrm{u}}(x, y) + S(x + 1, y + 1, 0) \\ a_{\mathrm{u}}(x, y) + 3 + S(x + 1, y + 1, 1) \end{bmatrix}, \tag{1.8}$$

or, equivalently,

$$S(x, y, 0) = a_{\mathrm{u}}(x, y) + \min \begin{bmatrix} S(x + 1, y + 1, 0) \\ 3 + S(x + 1, y + 1, 1) \end{bmatrix}. \tag{1.9}$$

Repeating the reasoning above under the assumption that we choose to start from (x, y) in a downward direction, we obtain

$$S(x, y, 1) = a_{\mathrm{d}}(x, y) + \min \begin{bmatrix} 3 + S(x + 1, y - 1, 0) \\ S(x + 1, y - 1, 1) \end{bmatrix}. \tag{1.10}$$

It is essential to note that we *do not* compare $S(x, y, 0)$ and $S(x, y, 1)$ but, rather, we compute, record, and later use *both* of them. (The clever reader might have observed that if one of the above S values exceeds the other by more than 3, the larger can never be part of an overall minimum-sum path and can be dropped. However, such cleverness, when it exploits the particular data of the problem—the fact that a turn costs 3 in this example—should be avoided since it, at the very least, obscures the general principle and, in this particular example, the computation needed to make the comparison may well exceed the savings accrued from occasionally dropping the larger from consideration.)

Formulas (1.9) and (1.10) together constitute the recurrence relation for this problem. We shall now write (1.9) and (1.10) together as just one formula, but first we want to *discourage* the reader from doing so in the

future and promise not to do so ourselves. The elegance of the result does not, in our opinion, justify the complexity of the formula. By first letting z equal 0 and comparing with (1.9) and then letting z equal 1 and comparing with (1.10), the reader can verify that

$$S(x, y, z) = (1 - z)a_u(x, y) + za_d(x, y) + \min\begin{bmatrix} S(x+1, y+1-2z, z) \\ 3 + S(x+1, y+1-2z, 1-z) \end{bmatrix}. \tag{1.11}$$

The boundary condition for our particular problem is, literally,

$$S(5, 1, 0) = \infty, \qquad S(5, 1, 1) = a_d(5, 1) = 2;$$
$$S(5, -1, 0) = a_u(5, -1) = 1, \qquad S(5, -1, 1) = \infty; \tag{1.12}$$

but if we write

$$S(6, 0, 0) = 0, \qquad S(6, 0, 1) = 0 \tag{1.13}$$

(and use our convention that nonexistent arcs cost ∞), then (1.9) and (1.10) imply the results (1.12). Conditions (1.13) assert that once we are at B, the end, there is no remaining cost no matter what we do there.

8. Solution of the Example

Using (1.7), (1.9) and (1.10), and (1.13), we now actually numerically solve the problem shown in Figure 1.1 with the additional stipulation that each direction change costs 3. $P(x, y, z) = U$ means that *up* is the optimal *second* decision if we start at (x, y) and move first in the direction indicated by z and a similar definition holds for $P(x, y, z) = D$.

Defining S as ∞ at points that are not vertices of our problem network,

$$S(5, 1, 0) = \infty + \min\begin{bmatrix} \infty \\ 3 + \infty \end{bmatrix} = \infty;$$
$$S(5, 1, 1) = 2 + \min\begin{bmatrix} 3 \\ 0 \end{bmatrix} = 2;$$
$$S(5, -1, 0) = 1 + \min\begin{bmatrix} 0 \\ 3 \end{bmatrix} = 1;$$
$$S(5, -1, 1) = \infty + \min\begin{bmatrix} 3 + \infty \\ \infty \end{bmatrix} = \infty.$$

Now, using the above results,

$$S(4, 2, 0) = \infty + \min\begin{bmatrix} \infty \\ 3 + \infty \end{bmatrix} = \infty;$$
$$S(4, 2, 1) = 5 + \min\begin{bmatrix} 3 + S(5, 1, 0) \\ S(5, 1, 1) \end{bmatrix} = 7, \qquad P(4, 2, 1) = D;$$

$$S(4,0,0) = 2 + \min\begin{bmatrix} S(5,1,0) \\ 3 + S(5,1,1) \end{bmatrix} = 7, \qquad P(4,0,0) = D;$$

$$S(4,0,1) = 8 + \min\begin{bmatrix} 3 + S(5,-1,0) \\ S(5,-1,1) \end{bmatrix} = 12, \qquad P(4,0,1) = U;$$

$$S(4,-2,0) = 4 + \min\begin{bmatrix} S(5,-1,0) \\ 3 + S(5,-1,1) \end{bmatrix} = 5, \qquad P(4,-2,0) = U;$$

$$S(4,-2,1) = \infty.$$

Proceeding to various situations that are three steps from the end (i.e., $x = 3$),

$$S(3,3,0) = \infty;$$

$$S(3,3,1) = 3 + \min\begin{bmatrix} 3 + \infty \\ 7 \end{bmatrix} = 10, \qquad P(3,3,1) = D;$$

$$S(3,1,0) = 3 + \min\begin{bmatrix} \infty \\ 3 + 7 \end{bmatrix} = 13, \qquad P(3,1,0) = D;$$

$$S(3,1,1) = 4 + \min\begin{bmatrix} 3 + 7 \\ 12 \end{bmatrix} = 14, \qquad P(3,1,1) = U;$$

$$S(3,-1,0) = 2 + \min\begin{bmatrix} 7 \\ 3 + 12 \end{bmatrix} = 9, \qquad P(3,-1,0) = U;$$

$$S(3,-1,1) = 2 + \min\begin{bmatrix} 3 + 5 \\ \infty \end{bmatrix} = 10, \qquad P(3,-1,1) = U;$$

$$S(3,-3,0) = 2 + \min\begin{bmatrix} 5 \\ \infty \end{bmatrix} = 7, \qquad P(3,-3,0) = U;$$

$$S(3,-3,1) = \infty.$$

Using these eight numbers to compute all four-step solutions (i.e., $x = 2$),

$$S(2,2,0) = 2 + \min\begin{bmatrix} \infty \\ 3 + 10 \end{bmatrix} = 15, \qquad P(2,2,0) = D;$$

$$S(2,2,1) = 1 + \min\begin{bmatrix} 3 + 13 \\ 14 \end{bmatrix} = 15, \qquad P(2,2,1) = D;$$

$$S(2,0,0) = 1 + \min\begin{bmatrix} 13 \\ 3 + 14 \end{bmatrix} = 14, \qquad P(2,0,0) = U;$$

$$S(2,0,1) = 2 + \min\begin{bmatrix} 3 + 9 \\ 10 \end{bmatrix} = 12, \qquad P(2,0,1) = D;$$

$$S(2,-2,0) = 5 + \min\begin{bmatrix} 9 \\ 3 + 10 \end{bmatrix} = 14, \qquad P(2,-2,0) = U;$$

$$S(2,-2,1) = 4 + \min\begin{bmatrix} 3 + 7 \\ \infty \end{bmatrix} = 14, \qquad P(2,-2,1) = U.$$

Next,

$$S(1, 1, 0) = 5 + \min\begin{bmatrix} 15 \\ 3 + 15 \end{bmatrix} = 20, \qquad P(1, 1, 0) = U;$$

$$S(1, 1, 1) = 4 + \min\begin{bmatrix} 3 + 14 \\ 12 \end{bmatrix} = 16, \qquad P(1, 1, 1) = D;$$

$$S(1, -1, 0) = 7 + \min\begin{bmatrix} 14 \\ 3 + 12 \end{bmatrix} = 21, \qquad P(1, -1, 0) = U;$$

$$S(1, -1, 1) = 3 + \min\begin{bmatrix} 3 + 14 \\ 14 \end{bmatrix} = 17, \qquad P(1, -1, 1) = D.$$

Finally,

$$S(0, 0, 0) = 1 + \min\begin{bmatrix} 20 \\ 3 + 16 \end{bmatrix} = 20, \qquad P(0, 0, 0) = D;$$

$$S(0, 0, 1) = 0 + \min\begin{bmatrix} 3 + 21 \\ 17 \end{bmatrix} = 17, \qquad P(0, 0, 1) = D.$$

From the last two numbers we conclude that the cost of the best path starting from A in an upward direction is 20 and in a downward direction is 17. Hence we choose to start in the downward direction. Since $P(0, 0, 1) = D$, we continue in the downward direction at the vertex $(1, -1)$. Since $P(1, -1, 1)$ is D, we continue downward when we reach $(2, -2)$. $P(2, -2, 1)$ being U, we turn and move diagonally upward at $(3, -3)$. $P(3, -3, 0)$ equaling U means we go upward at $(4, -2)$ and $P(4, -2, 0)$ being U means we go upward at $(5, -1)$. The optimal path is therefore $(0, 0)$, $(1, -1)$, $(2, -2)$, $(3, -3)$, $(4, -2)$, $(5, -1)$, $(6, 0)$; and its cost is 14 for its arc costs plus 3 for its one turn, 17 in all, which conforms with our computed value of $S(0, 0, 1)$.

It is crucial for the reader to understand that, to work the problem correctly, the "current situation" must include current direction information as well as current vertex information. Therefore, the optimal value function depends on an additional argument, which in turn somewhat increases the required computations.

Problem 1.7. The preceding problem can equally well be solved by defining the "current situation" to be a vertex and the direction of the arc by which we arrived at that vertex. Give the dynamic-programming formulation using this approach, but you need not solve the problem numerically.

Problem 1.8. For an N-step problem of the above type on a diamond-shaped network, how many additions and how many comparisons are needed for solution? For simplicity, assume that boundary vertices are the same as all others.

Problem 1.9. Give a forward dynamic-programming procedure for solving the turn-penalty problem solved above in the text. Do not solve numerically.

9. The Consultant Question

Once the key ideas of dynamic programming (the use of an optimal value function, its characterization by a recurrence relation, and its recursive solution yielding successively the solutions to problems of longer and longer duration) are understood, the art of dynamic-programming formulation involves the proper choice of the arguments for the optimal value function. If too few arguments are chosen (such as neglecting the direction information in the turn-penalty problem), no correct recurrence relation can be written. If too many are chosen (such as specifying the direction of the next two arcs, rather than one, in the turn-penalty problem) a correct result can be obtained, but at the expense of an unnecessary amount of computation. A good way to determine the right amount of information to incorporate in the arguments of the optimal value function for a backward procedure is to think as follows. Suppose that someone were making a series of (perhaps nonoptimal) decisions for the problem at hand and then decided that he was not doing too well and needed the help of an expert dynamic-programming consultant, namely you, the reader. After he has described the problem to you, but before he tells you anything about what he has done so far, he says, "Now you take over and do things optimally from now on." The minimal information that you would have to acquire from him about the current situation from which you are to start constitutes the arguments of the optimal value function. Thinking of the situation in this way and asking what information you would need to know in order to take over the decision making will subsequently be called "asking the consultant question."

Problem 1.10. Reconsider the original problem of this chapter with no turn penalty (see Figure 1.1) and define the optimal value function by (1.7). Give a correct dynamic-programming formulation based on this unnecessarily elaborate optimal value function.

10. Stage and State

Thus far we have spoken of the arguments of the optimal value function as describing the current situation. Generally one variable describes how many decisions have thus far been made, and it is one larger on the right-hand side of the recurrence relation than on the left regardless of the particular decision. (In (1.2), we are speaking of the variable x; likewise in (1.9) and (1.10). For some definitions of the arguments of the optimal value function, this variable will *decrease* by one after each decision.) This particular monotonic (i.e., either always increasing or always decreasing) variable is called the *stage* variable. All the remaining

variables needed to describe the current situation, given the stage variable, are called *state* variables. The values of the stage and state variables constitute a description of the situation adequate to allow a dynamic-programming solution.

Below is a series of problems of increasing difficulty that will allow the reader to check his understanding of the simple but elusive ideas covered above. The reader who proceeds without verifying that he can correctly solve the problems, *before* reading the solutions at the back of the book, does so at his own peril. It is our experience that there is a wide gap between understanding someone else's dynamic-programming solution and correctly devising one's own, and that the latter skill is acquired only through practice in independent problem solving. Correct problem formulation requires creative thinking rather than merely passive understanding. While we can lead the student to the understanding, we cannot make him think.

Unless otherwise stated, the following 12 problems seek the minimum-cost continuous path connecting A and B, always moving toward the right, on a general N-stage network (N even) of the type shown in Figure 1.6. Each arc has an arc cost associated with it, with $a_u(x, y)$ and $a_d(x, y)$ representing the arc costs as in the text. The total cost of a path is defined in the statement of the problem. For each problem, define the optimal value function, write the recurrence relation and give boundary conditions.

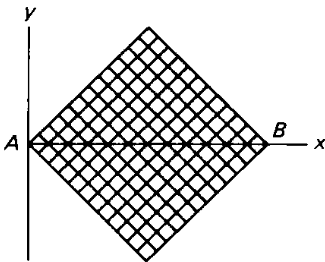


Figure 1.6

Problem 1.11. The cost of a path is the sum of the arc costs plus the total direction-change cost where the kth change of direction costs k, $k = 1, 2, 3, \ldots$ (i.e., a path with three direction changes has a total direction-change cost of $1 + 2 + 3 = 6$).

Problem 1.12. The cost is the sum of the arc costs plus a charge of 3 for each odd-numbered change of direction (i.e., the first, third, fifth, etc. direction change costs 3; other direction changes are free).

Problem 1.13. Only paths with m or fewer direction changes (for given m) are allowable. The cost is the sum of the arc costs. Direction changes cost 0.

Problem 1.14. The cost of each N-stage path from A to B is the sum of the $N - 1$ smallest arc costs along the path (i.e., you do not have to pay the largest arc cost on the path).

Problem 1.15. The cost of a path is the largest arc cost on the path (e.g., the cost of a path with arc costs 1, 7, 6, 2, 5, 3 is 7).

Problem 1.16. The cost of a path is the second largest arc cost on the path.

Problem 1.17. You start at A with Z coupons. If you choose to go diagonally up from (x, y) and simultaneously spend z of your coupons, the original cost $a_u(x, y)$ is reduced by an amount $g_u(x, y, z)$, where g_u increases with increasing z. A similar reduction of $g_d(x, y, z)$ occurs if you go diagonally down. The total cost is the sum of the reduced arc costs.

Problem 1.18. The path may start at any node, not just A. The cost is the sum of the arc costs plus a cost of $p(x, y)$ if the initial node of the path is (x, y).

Problem 1.19. The cost is the usual sum of the arc costs. If you change directions at any vertex, you are not allowed to do so at the next vertex of the path.

Problem 1.20. Only paths with sum of arc costs less than or equal to Z (a given number) are allowed. The cost of a path is the maximum arc cost along the path. All arc costs are nonnegative.

Problem 1.21. All arc costs are positive, and the cost is the *product* of the arc costs along the path.

Problem 1.22. Some arc costs are positive and some are negative. The cost is the product of the arc costs along the path.

11. The Doubling-Up Procedure

We now use a special kind of simple path problem to develop an idea that, when appropriate, saves considerable computation. The reader will be asked to apply this idea to various problems in subsequent chapters. In what follows we assume that all arcs point diagonally to the right, and we assume, and this is crucial to our method, that while as usual the arc costs depend on their initial and final vertices, they *do not depend* on the stage (i.e., the x coordinate). Such a repeating cost pattern is called stage-invariant and, later, when the stage is often time, it means that costs do not vary with time, only with the nature of the decision. An eight-stage example of such a network with terminal costs (shown in circles) as well as arc costs is shown in Figure 1.7 and subsequently solved.

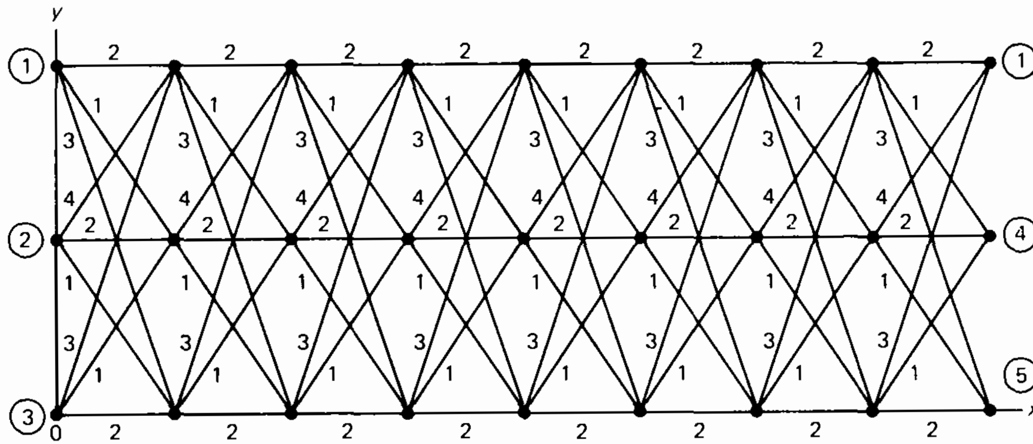


Figure 1.7

Our goal is to devise a procedure for doubling at each iteration the duration of the problem solved. To accomplish this aim we define an optimal value function that depends on three variables as follows:

$S(y_1, y_2, k) =$ the cost (ignoring terminal costs) of the minimum-cost path of length k stages connecting $y = y_1$ and $y = y_2$. (1.14)

Note that the actual x value at the start or finish is irrelevant, only the number of stages matters. We obtain a recurrence relation for this function by seeking the optimal value of y at the midpoint of a path of duration $2k$ stages connecting y_1 and y_2. Given any particular value of y at the end of k stages of a $2k$-stage problem, we clearly want to proceed *from* y_1 *to* that value of y in k stages at minimum cost and *from* that value y *to* y_2 in the next k stages, also at minimum cost. Then if we minimize this sum over all possible values of y at the end of k stages, we clearly have the cost of the best path of length $2k$ stages. The formula is therefore

$$S(y_1, y_2, 2k) = \min_{y=0,1,2} \left[S(y_1, y, k) + S(y, y_2, k) \right], \tag{1.15}$$

and the obvious boundary condition is

$$S(y_1, y_2, 1) = c_{y_1 y_2}, \tag{1.16}$$

where $c_{y_1 y_2}$ is the cost of the single arc directly connecting y_1 to y_2 in one step and, for the example of Figure 1.7, is given by the table

$$c_{00} = 2, \qquad c_{01} = 1, \qquad c_{02} = 3;$$
$$c_{10} = 1, \qquad c_{11} = 2, \qquad c_{12} = 4;$$
$$c_{20} = 3, \qquad c_{21} = 1, \qquad c_{22} = 2.$$

Once $S(y_1, y_2, 8)$ has been computed for all combinations of y_1 and y_2, nine in all, the value of the answer, including terminal costs which we

denote by $t_0(y)$ for $x = 0$ and $t_8(y)$ for $x = 8$, is given by

$$\text{answer} = \min_{\substack{y_1=0,1,2 \\ y_2=0,1,2}} \left[t_0(y_1) + S(y_1, y_2, 8) + t_8(y_2) \right]. \tag{1.17}$$

Letting $P(y_1, y_2, 2k)$ denote the optimal decision, i.e., the minimizing *midpoint* on the best path of duration $2k$ stages connecting y_1 and y_2, we now use (1.15)–(1.17) to solve the problem. By (1.16)

$$\begin{array}{lll} S(0,0,1) = 2, & S(0,1,1) = 1, & S(0,2,1) = 3, \\ S(1,0,1) = 1, & S(1,1,1) = 2, & S(1,2,1) = 4, \\ S(2,0,1) = 3, & S(2,1,1) = 1, & S(2,2,1) = 2. \end{array}$$

Now using (1.15) with $k = 1$ to obtain the optimal two-stage solutions for all pairs of end points,

$$\begin{aligned} S(0,0,2) &= \min[S(0,0,1) + S(0,0,1),\ S(0,1,1) + S(1,0,1),\ S(0,2,1) \\ &\qquad + S(2,0,1)] \\ &= \min[4, 2, 6] = 2, \qquad P(0,0,2) = 1; \\ S(0,1,2) &= \min[S(0,0,1) + S(0,1,1),\ S(0,1,1) + S(1,1,1),\ S(0,2,1) \\ &\qquad + S(2,1,1)] \\ &= \min[3, 3, 4] = 3, \qquad P(0,1,2) = 0 \text{ or } 1; \\ S(0,2,2) &= \min[2+3, 1+4, 3+2] = 5, \qquad P(0,2,2) = 0, 1, \text{ or } 2; \\ S(1,0,2) &= \min[3, 3, 7] = 3, \qquad P(1,0,2) = 0 \text{ or } 1; \\ S(1,1,2) &= \min[2, 4, 5] = 2, \qquad P(1,1,2) = 0; \\ S(1,2,2) &= \min[4, 6, 6] = 4, \qquad P(1,2,2) = 0; \\ S(2,0,2) &= \min[5, 2, 5] = 2, \qquad P(2,0,2) = 1; \\ S(2,1,2) &= \min[4, 3, 3] = 3, \qquad P(2,1,2) = 1 \text{ or } 2; \\ S(2,2,2) &= \min[6, 5, 4] = 4, \qquad P(2,2,2) = 2. \end{aligned}$$

Using (1.15) with $k = 2$ to solve all four-stage problems,

$$\begin{aligned} S(0,0,4) &= \min[S(0,0,2) + S(0,0,2),\ S(0,1,2) + S(1,0,2),\ S(0,2,2) \\ &\qquad + S(2,0,2)] \\ &= \min[2+2, 3+3, 5+2] = 4, \qquad P(0,0,4) = 0; \\ S(0,1,4) &= \min[2+3, 3+2, 5+3] = 5, \qquad P(0,1,4) = 0 \text{ or } 1; \\ S(0,2,4) &= \min[7, 7, 9] = 7, \qquad P(0,2,4) = 0 \text{ or } 1; \\ S(1,0,4) &= \min[5, 5, 6] = 5, \qquad P(1,0,4) = 0 \text{ or } 1; \\ S(1,1,4) &= \min[6, 4, 7] = 4, \qquad P(1,1,4) = 1; \\ S(1,2,4) &= \min[8, 6, 8] = 6, \qquad P(1,2,4) = 1; \end{aligned}$$

(*equations continue*)

$S(2, 0, 4) = \min[4, 6, 6] = 4, \qquad P(2, 0, 4) = 0;$

$S(2, 1, 4) = \min[5, 5, 7] = 5, \qquad P(2, 1, 4) = 0 \text{ or } 1;$

$S(2, 2, 4) = \min[7, 7, 8] = 7, \qquad P(2, 2, 4) = 0 \text{ or } 1.$

Using (1.15) once more to solve all eight-stage problems,

$$S(0, 0, 8) = \min[S(0, 0, 4) + S(0, 0, 4), S(0, 1, 4) + S(1, 0, 4), S(0, 2, 4) + S(2, 0, 4)]$$
$$= \min[8, 10, 11] = 8, \qquad P(0, 0, 8) = 0;$$

$S(0, 1, 8) = \min[9, 9, 12] = 9, \qquad P(0, 1, 8) = 0 \text{ or } 1;$

$S(0, 2, 8) = \min[11, 11, 14] = 11, \qquad P(0, 2, 8) = 0 \text{ or } 1;$

$S(1, 0, 8) = \min[9, 9, 10] = 9, \qquad P(1, 0, 8) = 0 \text{ or } 1;$

$S(1, 1, 8) = \min[10, 8, 11] = 8, \qquad P(1, 1, 8) = 1;$

$S(1, 2, 8) = \min[12, 10, 13] = 10, \qquad P(1, 2, 8) = 1;$

$S(2, 0, 8) = \min[8, 10, 11] = 8, \qquad P(2, 0, 8) = 0;$

$S(2, 1, 8) = \min[9, 9, 12] = 9, \qquad P(2, 1, 8) = 0 \text{ or } 1;$

$S(2, 2, 8) = \min[11, 11, 14] = 11, \qquad P(2, 2, 8) = 0 \text{ or } 1.$

Now, using (1.17) to obtain the value of the answer and the optimal choice of the initial and terminal points,

$$\begin{aligned}
\text{answer} = \min[&t_0(0) + S(0, 0, 8) + t_8(0),\ t_0(0) + S(0, 1, 8) + t_8(1),\\
&t_0(0) + S(0, 2, 8) + t_8(2),\ t_0(1) + S(1, 0, 8) + t_8(0),\\
&t_0(1) + S(1, 1, 8) + t_8(1),\ t_0(1) + S(1, 2, 8) + t_8(2),\\
&t_0(2) + S(2, 0, 8) + t_8(0),\ t_0(2) + S(2, 1, 8) + t_8(1),\\
&t_0(2) + S(2, 2, 8) + t_8(2)]\\
= \min[&3 + 8 + 5, 3 + 9 + 4, 3 + 11 + 1, 2 + 9 + 5, 2 + 8 + 4,\\
&2 + 10 + 1, 1 + 8 + 5, 1 + 9 + 4, 1 + 11 + 1]
\end{aligned}$$

$= 13$ with $y_1 = 1, y_2 = 2$, and $y_1 = 2, y_2 = 2$ both yielding that value.

Using the policy information to reconstruct optimal paths is somewhat tricky. Going from (0, 1) to (8, 2) (i.e., $y_1 = 1, y_2 = 2$) we refer to $P(1, 2, 8)$ and find that (4, 1) is the optimal *midpoint*. Now we conclude from $P(1, 1, 4) = 1$ that (2, 1) is the best midpoint of the first four-stage segment, i.e., (2, 1) lies on the path, and from $P(1, 2, 4) = 1$ that 1 is the best midpoint of the second four-stage segment, i.e., that (6, 1) is on the path. Since the first segment connects (0, 1) to (2, 1), we consult $P(1, 1, 2)$, which is 0, to deduce that (1, 0) is on the path. The second segment connects (2, 1) and (4, 1) and $P(1, 1, 2) = 0$ indicates (3, 0) is on the path.

The third segment connects (4, 1) and (6, 1), so again $P(1, 1, 2)$ tells us that (5, 0) is on the path; and, finally, the last segment joins (6, 1) and (8, 2), and $P(1, 2, 2) = 0$ says (7, 0) is on the path. One optimal solution is therefore (0, 1), (1, 0), (2, 1), (3, 0), (4, 1), (5, 0), (6, 1), (7, 0), (8, 2). A similar deductive process tells us that nine paths connect (0, 2) and (8, 2) at a cost of $11 + 2$ for the terminal costs. Just listing the successive y coordinates, they are

2–1–0–1–0–1–0–0–2, 2–1–0–1–0–1–0–1–2, 2–1–0–1–0–1–0–2–2,
2–1–0–1–0–0–1–0–2, 2–1–0–1–0–1–1–0–2, 2–1–0–0–1–0–1–0–2,
2–1–0–1–1–0–1–0–2, 2–1–1–0–1–0–1–0–2, 2–2–1–0–1–0–1–0–2.

Let us now compare the amount of computation required by this *doubling-up* procedure to that required by the usual procedure for this problem, where we assume that the duration is 2^N stages and there are M states at each stage. In our example $N = 3$ and $M = 3$. For doubling-up, each doubling of k requires M additions for each of M^2 pairs (y_1, y_2). We must double-up N times to solve the 2^N-stage problem (neglecting the terminal costs) so $N \cdot M^3$ additions are needed. For the usual procedure, each stage requires M^2 additions (M decisions at each of M points) so roughly $2^N \cdot M^2$ additions are needed. For $N = 3$ and $M = 3$ the usual one-state-variable procedure is slightly better, but for $N = 4$ (i.e., a duration of 16 stages) doubling-up dominates. No matter what M is, for large enough N, doubling-up will dominate. We remind the reader once more that doubling-up can be used only for time-invariant processes, while the usual procedure still works for completely general stage-dependent costs.

To solve using doubling-up, say a 12-stage problem, one can combine $S(y_1, y_2, 8)$ and $S(y_1, y_2, 4)$ by the formula

$$S(y_1, y_2, 12) = \min_y \left[S(y_1, y, 8) + S(y, y_2, 4) \right]. \tag{1.18}$$

Similarly, a 14-stage problem can then be solved by combining $S(y_1, y_2, 12)$ with $S(y_1, y_2, 2)$, so the procedure can still be used even if the duration is not exactly some power of two. Generally, we have the formula

$$S(y_1, y_2, m + n) = \min_y \left[S(y_1, y, m) + S(y, y_2, n) \right], \tag{1.19}$$

which raises some interesting questions about the minimum number of iterations to get to some given duration N.

Problem 1.23. Using doubling-up and formulas like (1.18), how many iterations are needed for duration 27? Can you find a procedure using (1.19) that requires fewer iterations?