THE ART AND THEORY OF DYNAMIC PROGRAMMING

Stuart E. Dreyfus

Department of Industrial Engineering and Operations Research
University of California
Berkeley, California

Averill M. Law

Department of Industrial Engineering
University of Wisconsin
Madison, Wisconsin

ACADEMIC PRESS New York San Francisco London 1977
A Subsidiary of Harcourt Brace Jovanovich, Publishers

ACADEMIC PRESS, INC.
111 Fifth Avenue, New York, New York 10003

United Kingdom Edition published by
ACADEMIC PRESS, INC. (LONDON) LTD.
24/28 Oval Road, London NW1

Library of Congress Cataloging in Publication Data

Dreyfus, Stuart E
The art and theory of dynamic programming.

(Mathematics in science and engineering ; vol. 130)
Includes bibliographical references.
1. Dynamic programming. I. Law, Averill M., joint author. II. Title. III. Series.
T57.83.D73 519.7'03 76-19486
ISBN 0-12-221860-4

PRINTED IN THE UNITED STATES OF AMERICA

CONTENTS

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

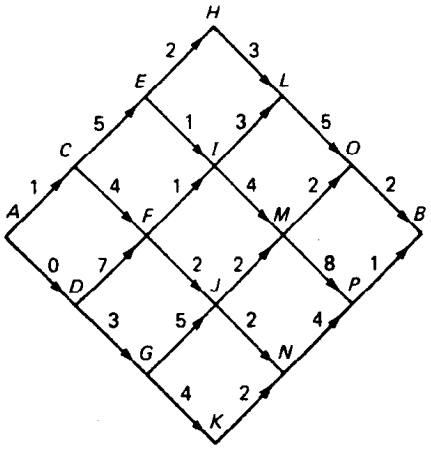

Figure 1.1

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 $\min\begin{bmatrix} x \\ y \end{bmatrix}$ 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 $(x + 1,$

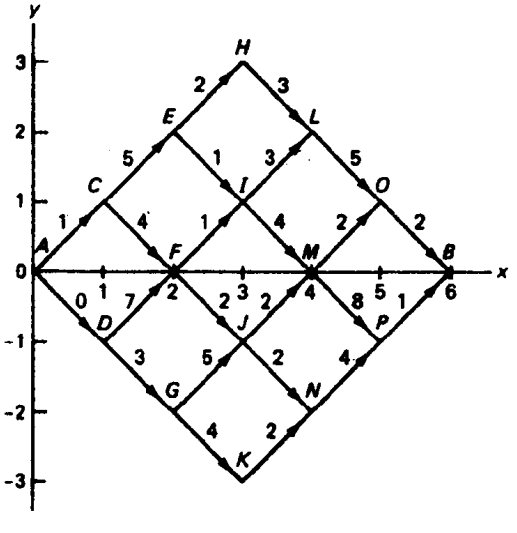

Figure 1.2

$y - 1$), 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)).

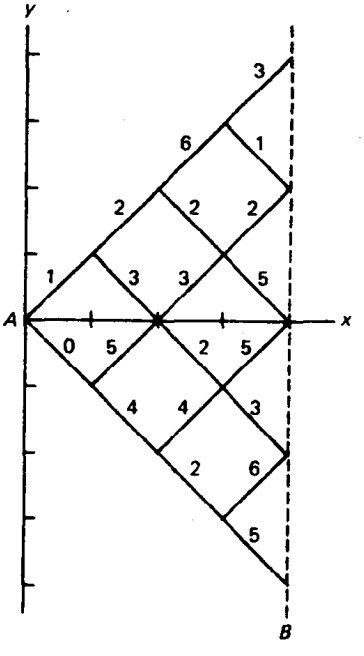

Figure 1.3

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

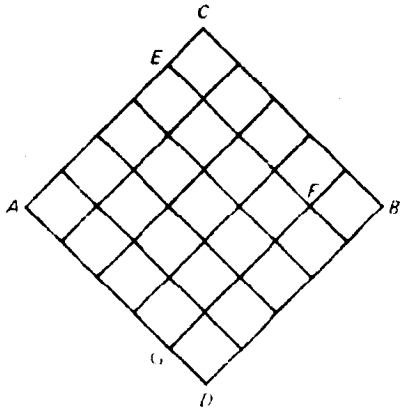

Figure 1.4

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.

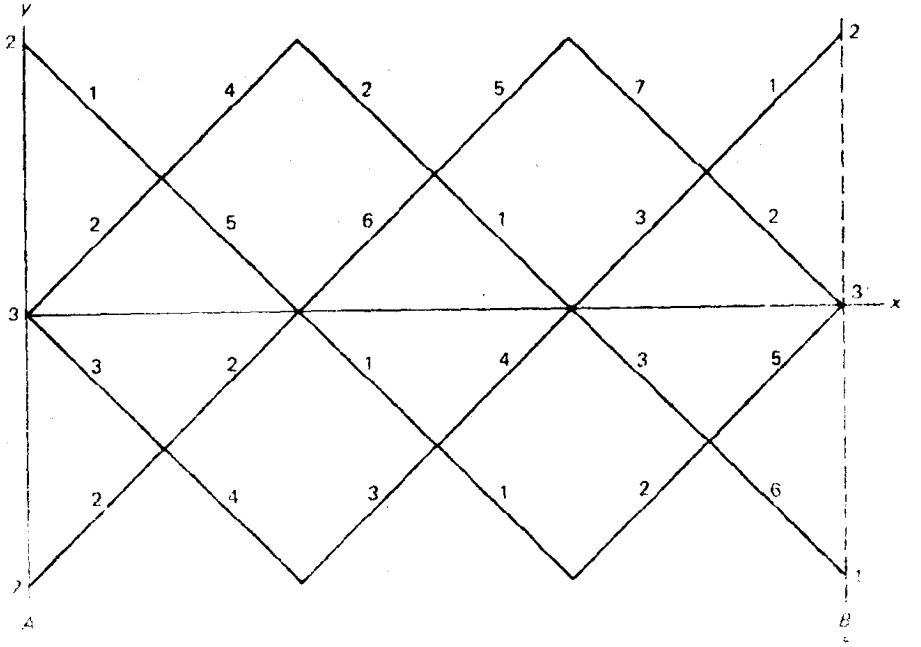

Figure 1.5