Dynamic Programming

11

Dynamic programming is an optimization approach that transforms a complex problem into a sequence of simpler problems; its essential characteristic is the multistage nature of the optimization procedure. More so than the optimization techniques described previously, dynamic programming provides a general framework for analyzing many problem types. Within this framework a variety of optimization techniques can be employed to solve particular aspects of a more general formulation. Usually creativity is required before we can recognize that a particular problem can be cast effectively as a dynamic program; and often subtle insights are necessary to restructure the formulation so that it can be solved effectively.

We begin by providing a general insight into the dynamic programming approach by treating a simple example in some detail. We then give a formal characterization of dynamic programming under certainty, followed by an in-depth example dealing with optimal capacity expansion. Other topics covered in the chapter include the discounting of future returns, the relationship between dynamic-programming problems and shortest paths in networks, an example of a continuous-state-space problem, and an introduction to dynamic programming under uncertainty.

11.1 AN ELEMENTARY EXAMPLE

In order to introduce the dynamic-programming approach to solving multistage problems, in this section we analyze a simple example. Figure 11.1 represents a street map connecting homes and downtown parking lots for a group of commuters in a model city. The arcs correspond to streets and the nodes correspond to intersections. The network has been designed in a diamond pattern so that every commuter must traverse five streets in driving from home to downtown. The design characteristics and traffic pattern are such that the total time spent by any commuter between intersections is independent of the route taken. However, substantial delays, are experienced by the commuters in the intersections. The lengths of these delays in minutes, are indicated by the numbers within the nodes. We would like to minimize the total delay any commuter can incur in the intersections while driving from his home to downtown. Figure 11.2 provides a compact tabular representation for the problem that is convenient for discussing its solution by dynamic programming. In this figure, boxes correspond to intersections in the network. In going from home to downtown, any commuter must move from left to right through this diagram, moving at each stage only to an adjacent box in the next column to the right. We will refer to the ‘‘stages to go," meaning the number of intersections left to traverse, not counting the intersection that the commuter is currently in.

The most naive approach to solving the problem would be to enumerate all 150 paths through the diagram, selecting the path that gives the smallest delay. Dynamic programming reduces the number of computations by moving systematically from one side to the other, building the best solution as it goes.

Suppose that we move backward through the diagram from right to left. If we are in any intersection (box) with no further intersections to go, we have no decision to make and simply incur the delay corresponding to that intersection. The last column in Fig. 11.2 summarizes the delays with no (zero) intersections to go.

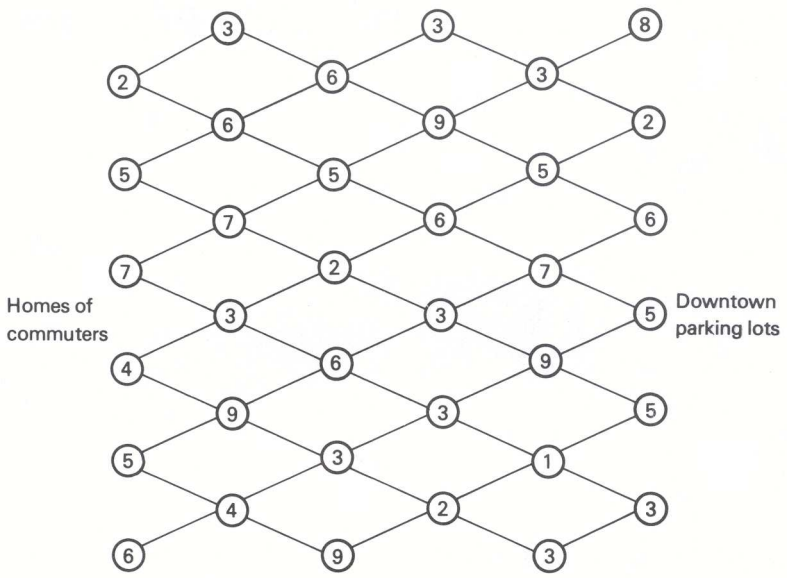

Figure 11.1 Street map with intersection delays.

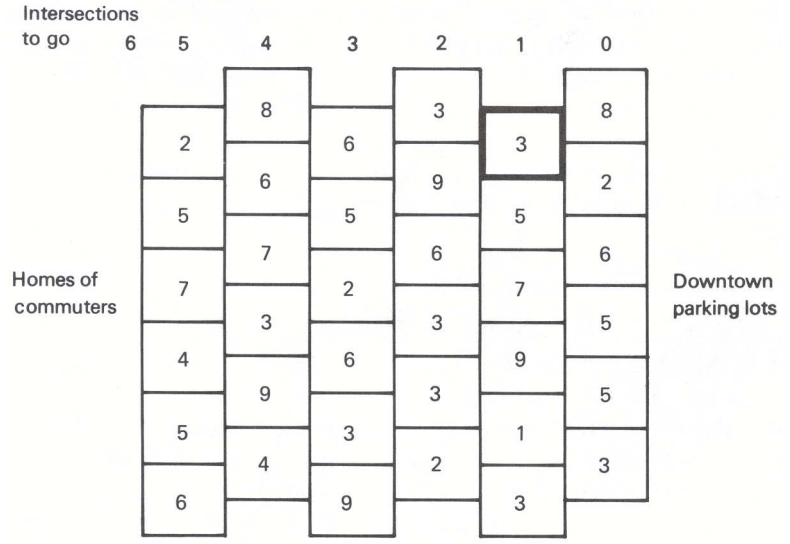

Figure 11.2 Compact representation of the network.

Our first decision (from right to left) occurs with one stage, or intersection, left to go. If for example, we are in the intersection corresponding to the highlighted box in Fig. 11.2, we incur a delay of three minutes in this intersection and a delay of either *eight* or *two* minutes in the last intersection, depending upon whether we move up or down. Therefore, the smallest possible delay, or optimal solution, in this intersection is $3+2=5$ minutes. Similarly, we can consider each intersection (box) in this column in turn and compute the smallest total delay as a result of being in each intersection. The solution is given by the bold-faced numbers in Fig. 11.3. The arrows indicate the optimal decision, up or down, in any intersection with one stage, or one intersection, to go.

Note that the numbers in bold-faced type in Fig. 11.3 completely summarize, for decision-making purposes, the total delays over the last two columns. Although the original numbers in the last two columns have been used to determine the bold-faced numbers, whenever we are making decisions to the left of these columns we need only know the bold-faced numbers. In an intersection, say the topmost with one stage to go, we know that our (optimal) remaining delay, including the delay in this intersection, is five minutes. The bold-faced numbers summarize all delays from this point on. For decision-making to the left of the bold-faced numbers, the last column can be ignored.

With this in mind, let us back up one more column, or stage, and compute the optimal solution in each intersection with two intersections to go. For example, in the bottom-most intersection, which is highlighted in Fig. 11.3, we incur a delay of two minutes in the intersection, plus *four* or *six* additional minutes, depending upon whether we move up or down. To minimize delay, we move *up* and incur a total delay in this intersection and *all remaining intersections* of $2+4=6$ minutes. The remaining computations in this column are summarized in Fig. 11.4, where the bold-faced numbers reflect the optimal total delays in each intersection with two stages, or two intersections, to go.

Once we have computed the optimal delays in each intersection with two stages to go, we can again move back one column and determine the optimal delays and the optimal decisions with three intersections to go. In the same way, we can continue to move back one stage at a time, and compute the optimal delays and decisions with four and five intersections to go, respectively. Figure 11.5 summarizes these calculations.

Figure 11.5(c) shows the optimal solution to the problem. The least possible delay through the network is 18 minutes. To follow the least-cost route, a commuter has to start at the second intersection from the bottom. According to the optimal decisions, or arrows, in the diagram, we see that he should next move down to the bottom-most intersection in column 4. His following decisions should be up, down, up, down, arriving finally at the bottom-most intersection in the last column.

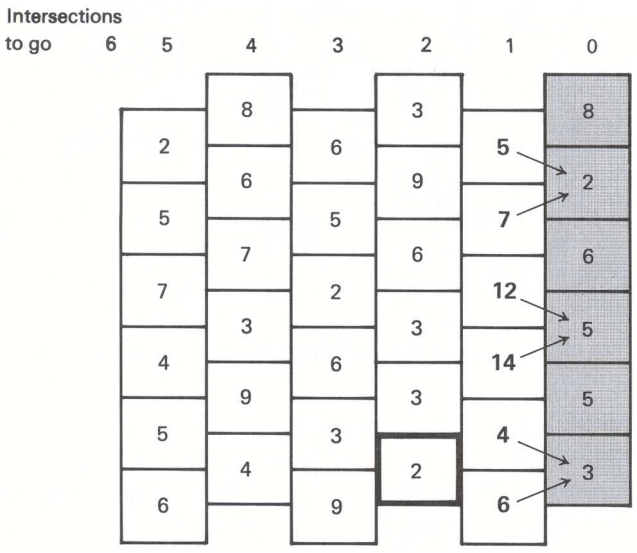

Figure 11.3 Decisions and delays with one intersection to go.

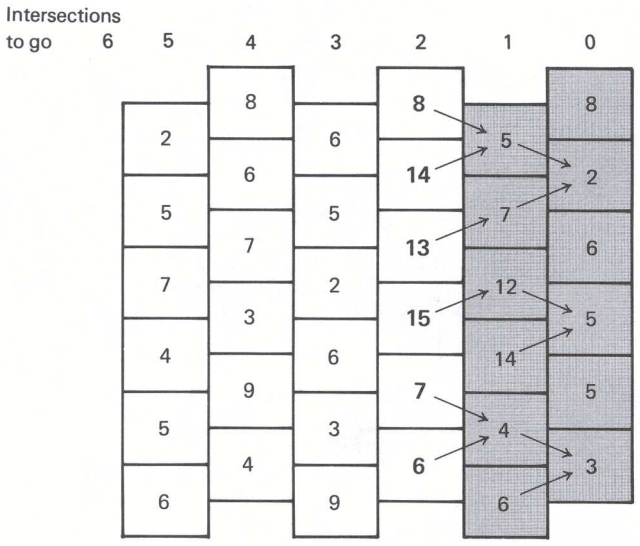

Figure 11.4 Decisions and delays with two intersections to go.

However, the commuters are probably not free to arbitrarily choose the intersection they wish to start from. We can assume that their homes are adjacent to only one of the leftmost intersections, and therefore each commuter's starting point is fixed. This assumption does not cause any difficulty since we have, in fact, determined the routes of minimum delay from the downtown parking lots to *all* the commuter's homes. Note that this assumes that commuters do not care in which downtown lot they park. Instead of solving the minimum-delay problem for only a particular commuter, we have *embedded* the problem of the particular commuter in the more general problem of finding the minimum-delay paths from all homes to the group of downtown parking lots. For example, Fig. 11.5 also indicates that the commuter starting at the topmost intersection incurs a delay of 22 minutes if he follows his optimal policy of down, up, up, down, and then down. He presumably parks in a lot close to the second intersection from the top in the last column. Finally, note that three of the intersections in the last column are not entered by any commuter. The analysis has determined the minimum-delay paths from each of the commuter's homes to the group of downtown parking lots, not to each particular parking lot.

Using dynamic programming, we have solved this minimum-delay problem sequentially by keeping track of how many intersections, or stages, there were to go. In dynamic-programming terminology, each point where decisions are made is usually called a *stage* of the decision-making process. At any stage, we need only know which intersection we are in to be able to make subsequent decisions. Our subsequent decisions do not depend upon how we arrived at the particular intersection. Information that summarizes the knowledge required about the problem in order to make the current decisions, such as the intersection we are in at a particular stage, is called a *state* of the decision-making process.

In terms of these notions, our solution to the minimum-delay problem involved the following intuitive idea, usually referred to as the *principle of optimality*.

> *Any optimal policy has the property that, whatever the current state and decision, the remaining decisions must constitute an optimal policy with regard to the state resulting from the current decision.*

To make this principle more concrete, we can define the *optimal-value function* in the context of the minimum-delay problem.

$v_n(s_n) =$ Optimal value (minimum delay) over the current and subsequent stages (intersections), given that we are in state s_n (in a particular intersection) with n stages (intersections) to go.

The optimal-value function at each stage in the decision-making process is given by the appropriate column

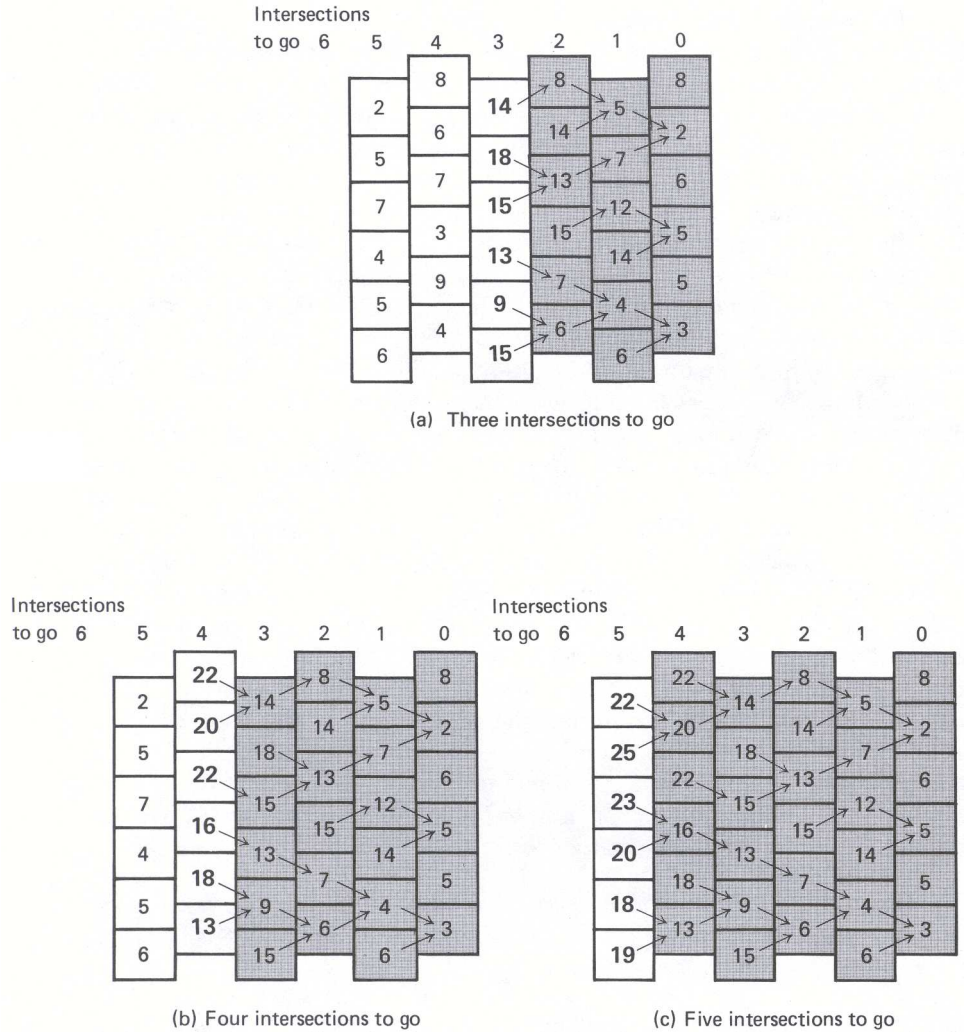

Figure 11.5 Charts of optimal delays and decisions.

of Fig. 11.5(c). We can write down a *recursive* relationship for computing the optimal-value function by recognizing that, at each stage, the decision in a particular state is determined simply by choosing the minimum total delay. If we number the states at each stage as $s_n = 1$ (bottom intersection) up to $s_n = 6$ (top intersection), then

$$v_n(s_n) = \text{Min}\,\{t_n(s_n) + v_{n-1}(s_{n-1})\}\,, \tag{1}$$

subject to:

$$s_{n-1} = \begin{cases} s_n + 1 & \text{if we choose up and } n \text{ even,} \\ s_n - 1 & \text{if we choose down and } n \text{ odd,} \\ s_n & \text{otherwise,} \end{cases}$$

where $t_n(s_n)$ is the delay time in intersection s_n at stage n.

The columns of Fig. 11.5(c) are then determined by starting at the right with

$$v_0(s_0) = t_0(s_0) \qquad (s_0 = 1, 2, \ldots, 6), \tag{2}$$

and successively applying Eq. (1). Corresponding to this optimal-value function is an *optimal-decision function*, which is simply a list giving the optimal decision for each state at every stage. For this example, the optimal decisions are given by the arrows leaving each box in every column of Fig. 11.5(c).

The method of computation illustrated above is called *backward induction*, since it starts at the right and moves back one stage at a time. Its analog, *forward induction*, which is also possible, starts at the left and moves forward one stage at a time. The spirit of the calculations is identical but the interpretation is somewhat different. The optimal-value function for forward induction is defined by:

$u_n(s_n) =$ Optimal value (minimum delay) over the current andcompleted stages (intersections), given that we are in state s_n (in a particular intersection) with n stages (intersections) to go.

The recursive relationship for forward induction on the minimum-delay problem is

$$u_{n-1}(s_{n-1}) = \text{Min}\,\{u_n(s_n) + t_{n-1}(s_{n-1})\}\,, \tag{3}$$

subject to:

$$s_{n-1} = \begin{cases} s_n + 1 & \text{if we choose up and } n \text{ even,} \\ s_n - 1 & \text{if we choose down and } n \text{ odd,} \\ s_n & \text{otherwise,} \end{cases}$$

where the stages are numbered in terms of intersections to go. The computations are carried out by setting

$$u_5(s_5) = t_5(s_5) \qquad (s_5 = 1, 2, \ldots, 6), \tag{4}$$

and successively applying (3). The calculations for forward induction are given in Fig. 11.6. When performing forward induction, the stages are usually numbered in terms of the number of stages *completed* (rather than the number of stages to go). However, in order to make a comparison between the two approaches easier, we have avoided using the ‘‘stages completed" numbering.

The columns of Fig. 11.6(f) give the optimal-value function at each stage for the minimum-delay problem, computed by forward induction. This figure gives the minimum delays from each particular downtown parking lot to the *group* of homes of the commuters. Therefore, this approach will only guarantee finding the minimum delay path from the downtown parking lots to *one* of the commuters' homes. The method, in fact, finds the minimum-delay path to a particular origin only if that origin may be reached from a downtown parking lot by a backward sequence of arrows in Fig. 11.6(f).

If we select the minimum-delay path in Fig. 11.6(f), lasting 18 minutes, and follow the arrows backward, we discover that this path leads to the intersection second from the bottom in the first column. This is the same minimum-delay path determined by backward induction in Fig. 11.5(c).