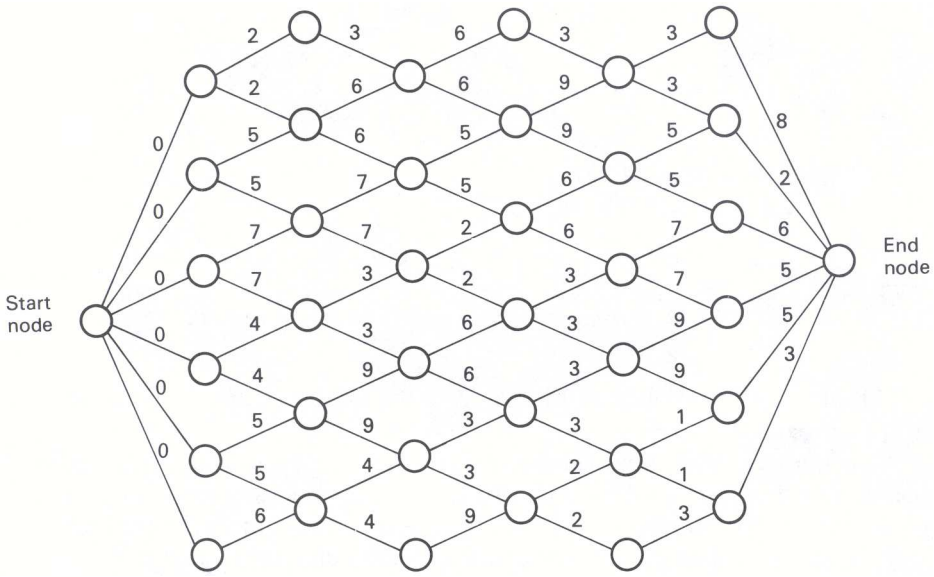

Figure 11.11 Shortest-path network for minimum-delay problem.

such a node. (Otherwise, from any node, we can move along an arc to another node. Starting from any node and continuing to move away from any node encountered, we eventually would revisit a node, determining a cycle, contradicting the acyclic assumption.) By ignoring the numbered nodes and their incident arcs, the procedure is continued until all nodes are numbered.

This procedure is applied, in Fig. 11.12, to the longest-path problem introduced as a critical-path scheduling example in Section 8.1.

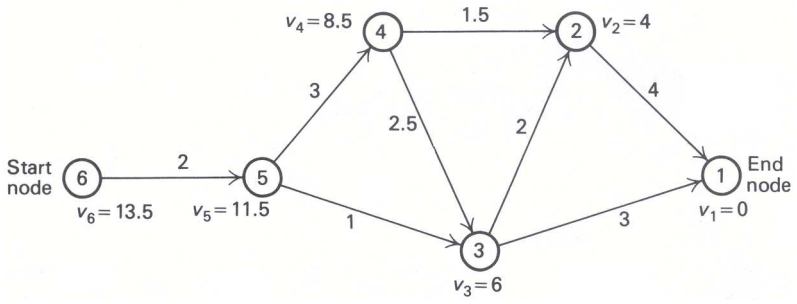

Figure 11.12 Finding the longest path in an acyclic network.

We can apply the dynamic-programming approach by viewing each node as a stage, using either backward induction to consider the nodes in ascending order, or forward induction to consider the nodes in reverse order. For backward induction, v_n will be interpreted as the longest distance from node n to the end node. Setting $v_1 = 0$, dynamic programming determines $v_2, v_3, \ldots, v_N$ in order, by the recursion

$$v_n = \text{Max}[d_{nj} + v_j], \quad j < n,$$

where d_{nj} is the given distance on arc n–j. The results of this procedure are given as node labels in Fig. 11.12 for the critical-path example.

For a shortest-path problem, we use minimization instead of maximization in this recursion. Note that the algorithm finds the longest (shortest) paths from every node to the end node. If we want only the longest path to the start node, we can terminate the procedure once the start node has been labeled. Finally, we could have found the longest distances from the start node to all other nodes by labeling the nodes in the reverse order, beginning with the start node.

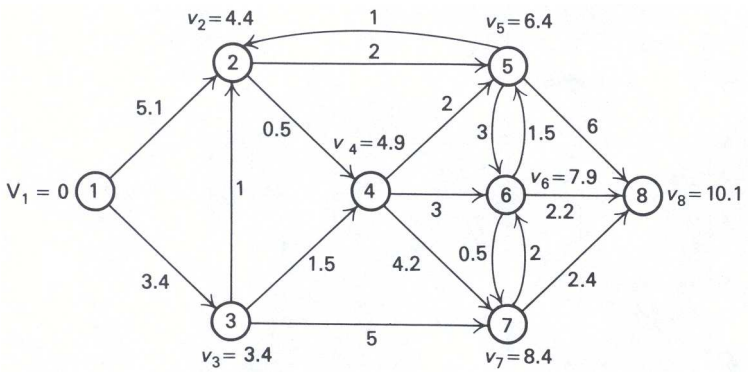

Figure 11.13 Shortest paths in a network without negative cycles.

A more complicated algorithm must be given for the more general problem of finding the shortest path between two nodes, say nodes 1 and N, in a network without negative cycles. In this case, we can devise a dynamic-programming algorithm based upon a value function defined as follows:

$v_n(j)$ = Shortest distance from node 1 to node j along paths using at most n intermediate nodes.

By definition, then,

$$v_0(j) = d_{1j} \quad \text{for } j = 2, 3, \ldots, N,$$

the length d_{1j} of arc 1–j since no intermediate nodes are used. The dynamic-programming recursion is

$$v_n(j) = \text{Min } \{d_{ij} + v_{n-1}(i)\}, \quad 1 \le j \le N, \tag{16}$$

which uses the principle of optimality: that any path from node 1 to node j, using at most n intermediate nodes, arrives at node j from node i along arc i–j after using the shortest path with at most $(n-1)$ intermediate nodes from node j to node i. We allow $i = j$ in the recursion and take $d_{jj} = 0$, since the optimal path using at most n intermediate nodes may coincide with the optimal path with length $v_{n-1}(j)$ using at most $(n-1)$ intermediate nodes.

The algorithm computes the shortest path from node 1 to every other node in the network. It terminates when $v_n(j) = v_{n-1}(j)$ for every node j, since computations in Eq. (16) will be repeated at every stage from n on. Because no path (without cycles) uses any more than $(N-1)$ intermediate nodes, where N is the total number of nodes, the algorithm terminates after at most $(N-1)$ steps.

As an application of the method, we solve the shortest-path problem introduced in Chapter 8 and given in Fig. 11.13.

Initially the values $v_0(j)$ are given by

$$v_0(1) = 0, \quad v_0(2) = d_{12} = 5.1, \qquad v_0(3) = d_{13} = 3.4,$$

and

$$v_0(j) = \infty \quad \text{for } j = 4, 5, 6, 7, 8,$$

since these nodes are not connected to node 1 by an arc. The remaining steps are specified in Tableaus 7, 8, and 9. The computations are performed conveniently by maintaining a table of distances d_{ij}. If the list $v_0(i)$ is placed to the left of this table, then recursion Eq. (14) states that $v_1(j)$ is given by the smallest of the comparisons:

$$v_0(i) + d_{ij} \quad \text{for } i = 1, 2, \ldots, 8.$$

That is, place the column $v_0(i)$ next to the jth column of the d_{ij} table, add the corresponding elements, and take $v_1(j)$ as the smallest of the values. If $v_1(j)$ is recorded below the jth column, the next iteration to find $v_2(j)$ is initiated by replacing the column $v_0(i)$ with the elements $v_1(j)$ from below the distance table.

As the reader can verify, the next iteration gives $v_4(j) = v_3(j)$ for all j. Consequently, the values $v_3(j)$ recorded in Tableau 9 are the shortest distances from node 1 to each of the nodes $j = 2, 3, \ldots, 8$.

Tableau 7[†]

		Node j							
Node i	$v_0(i)$	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	0	0	5.1	3.4					
2	5.1		0		.5	2			
3	3.4		1	0	1.5			5	
4	$+\infty$				0	2	3	4.2	
5	$+\infty$					0	3		6
6	$+\infty$					2	0	.5	2.2
7	$+\infty$						2	0	2.4
8	$+\infty$								0
$v_1(j) = \min\{d_{ij} + v_0(i)\}$		0	4.4	3.4	4.9	7.1	$+\infty$	8.4	$+\infty$

Tableau 8[†]

		Node j							
Node i	$v_1(i)$	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	0	0	5.1	3.4					
2	4.4		0		.5	2			
3	3.4		1	0	1.5			5	
4	4.9				0	2	3	4.2	
5	7.1					0	3		6
6	$+\infty$					2	0	.5	2.2
7	8.4						2	0	2.4
8	$+\infty$								0
$v_2(j) = \min\{d_{ij} + v_1(i)\}$		0	4.4	3.4	4.9	6.4	7.9	8.4	10.8

Tableau 9[†]

		Node j							
Node i	$v_2(i)$	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	0	0	5.1	3.4					
2	4.4		0		.5	2			
3	3.4		1	0	1.5			5	
4	4.9				0	2	3	4.2	
5	6.4					0	3		6
6	7.9					2	0	.5	2.2
7	8.4						2	0	2.4
8	10.8								0
$v_3(j) = \min\ d_{ij} + v_2(i)$		0	4.4	3.4	4.9	6.4	7.9	8.4	10.1

[†] $d_{ij} = +\infty$, if blank.

11.6 CONTINUOUS STATE-SPACE PROBLEMS

Until now we have dealt only with problems that have had a finite number of states associated with each stage. Since we also have assumed a finite number of stages, these problems have been identical to finding the shortest path through a network with special structure. Since the development, in Section 11.3, of the fundamental recursive relationship of dynamic programming did not depend on having a finite number of states at each stage, here we introduce an example that has a continuous state space and show that the same procedures still apply.

Suppose that some governmental agency is attempting to perform cost/benefit analysis on its programs in order to determine which programs should receive funding for the next fiscal year. The agency has managed to put together the information in Table 11.2. The benefits of each program have been converted into equivalent tax savings to the public, and the programs have been listed by decreasing benefit-to-cost ratio. The agency has taken the position that there will be no partial funding of programs. Either a program *will* be funded at the indicated level or it will *not* be considered for this budget cycle. Suppose that the agency is fairly sure of receiving a budget of $34 million from the state legislature if it makes a good case that the money is being used effectively. Further, suppose that there is some possibility that the budget will be as high as $42 million. How can the agency make the most effective use of its funds at *either* possible budget level?

Table 11.2 Cost/benefit information by program.

Program	*Expected benefit*	*Expected cost*	*Benefit/Cost*
A	$ 59.2 M	$ 2.8 M	21.1
B	31.4	1.7	18.4
C	15.7	1.0	15.7
D	30.0	3.2	9.4
E	105.1	15.2	6.9
F	11.6	2.4	4.8
G	67.3	16.0	4.2
H	2.3	.7	3.3
I	23.2	9.4	2.5
J	18.4	10.1	1.8
	$364.2 M	$62.5 M	

We should point out that mathematically this problem is an integer program. If b_j is the benefit of the jth program and c_j is the cost of that program, then an integer-programming formulation of the agency's budgeting problem is determined easily.
Letting

$$x_j = \begin{cases} 1 & \text{if program } j \text{ is funded,} \\ 0 & \text{if program } j \text{ is } \textit{not} \text{ funded,} \end{cases}$$

the integer-programming formulation is:

$$\text{Maximize} \sum_{j=1}^{n} b_j x_j,$$

subject to:

$$\sum_{j=1}^{n} c_j x_j \le B,$$

$$x_j = 0 \quad \text{or} \quad 1 \qquad (j = 1, 2, \ldots, n),$$

where B is the total budget allocated. This cost/benefit example is merely a variation of the well-known knapsack problem that was introduced in Chapter 9.We will ignore, for the moment, this integer-programming formulation and proceed to develop a highly efficient solution procedure using dynamic programming.

In order to approach this problem via dynamic programming, we need to define the stages of the system, the state space for each stage, and the optimal-value function.

Let

$v_k(B)$ = Maximum total benefit obtainable, choosing from the first k programs, with budget limitation B.

With this definition of the optimal-value function, we are letting the first k programs included be the number of ‘‘stages to go" and the available budget at each stage be the state space. Since the possible budget might take on any value, we are allowing for a continuous state space for each stage. In what follows the order of the projects is immaterial although the order given in Table 11.2 may have some computational advantages.

Let us apply the dynamic-programming reasoning as before. It is clear that with $k = 0$ programs, the total benefit must be zero regardless of the budget limitation. Therefore

$$v_0(B_0) = 0 \qquad \text{for } B_0 \geq 0.$$

If we now let $k = 1$, it is again clear that the optimal-value function can be determined easily since the budget is either large enough to fund the first project, or *not*. (See Tableau 10.)

Tableau 10

B_1 \ d_1	$x_1 = 0$	$x_1 = 1$	$v_1(B_1)$	$d_1^*(B_1)$
$2.8 \leqq B$	0	59.2	59.2	1
$0 \leqq B < 2.8$	0	—	0	0
	$c_1(B_1, d_1)$			

Now consider which programs to fund when the first two programs are available. The optimal-value function $v_2(B_2)$ and optimal decision function $d_2^*(B_2)$ are developed in Tableau 11.

Tableau 11

B_2 \ d_2	$x_2 = 0$	$x_2 = 1$	$v_2(B_2)$	$d_2^*(B_2)$
$4.5 \leqq B_2$	59.2	59.2 + 31.4	90.6	1
$2.8 \leqq B_2 < 4.5$	59.2	31.4	59.2	0
$1.7 \leqq B_2 < 2.8$	0	31.4	31.4	1
$0 \leqq B_2 < 1.7$	0	—	0	0
	$c_2(B_2, d_2) + v_1(B_1)$			

Here again the dash means that the current state and decision combination will result in a state that is not permissible. Since this tableau is fairly simple, we will go on and develop the optimal-value function $v_3(B_3)$ and optimal decision function $d_3^*(B_3)$ when the first three programs are available (see Tableau 12).

For any budget level, for example, \$4.0 M, we merely consider the two possible decisions: either funding program C ($x_3 = 1$) or not ($x_3 = 0$). If we fund program C, then we obtain a benefit of \$15.7 M while consuming \$1.0 M of our own budget. The remaining \$3.0 M of our budget is then optimally allocated to the remaining programs, producing a benefit of \$59.2 M, which we obtain from the optimal-value function with the first two programs included (Tableau 11). If we do not fund program C, then the entire amount of \$4.0 M is optimally allocated to the remaining two programs (Tableau 11), producing a benefit of \$59.2. Hence, we should clearly fund program C if our budget allocation is \$4.0 M. Optimal decisions taken for other budget levels are determined in a similar manner.

Tableau 12

B_3 \ d_3	$x_3 = 0$	$x_3 = 1$	$v_3(B_3)$	$d_3^*(B_3)$
$5.5 \leqq B_3$	90.6	90.6 + 15.7	106.3	1
$4.5 \leqq B_3 < 5.5$	90.6	59.2 + 15.7	90.6	0
$3.8 \leqq B_3 < 4.5$	59.2	59.2 + 15.7	74.9	1
$2.8 \leqq B_3 < 3.8$	59.2	31.4 + 15.7	59.2	0
$2.7 \leqq B_3 < 2.8$	31.4	31.4 + 15.7	47.1	1
$1.7 \leqq B_3 < 2.7$	31.4	0 + 15.7	31.4	0
$1.0 \leqq B_3 < 1.7$	0	0 + 15.7	15.7	1
$0 \leqq B_3 < 1.0$	0	—	0	0
		$c_3(B_3, d_3) + v_2(B_2)$		

Although it is straightforward to continue the recursive calculation of the optimal-value function for succeeding stages, we will not do so since the number of ranges that need to be reported rapidly becomes rather large. The general recursive relationship that determines the optimal-value function at each stage is given by:

$$v_n(B_n) = \text{Max } [c_n x_n + v_{n-1}(B_n - c_n x_n)],$$

subject to:

$$x_n = 0 \quad \text{or} \quad 1.$$

The calculation is initialized by observing that

$$v_0(B_0) = 0$$

for all possible values of B_0. Note that the state transition function is simply

$$B_{n-1} = t_n(x_n, B_n) = B_n - c_n x_n.$$

We can again illustrate the usual principle of optimality: Given budget B_n at stage n, whatever decision is made with regard to funding the nth program, the remaining budget must be allocated optimally among the first $(n - 1)$ programs. If these calculations were carried to completion, resulting in $v_{10}(B_{10})$ and $d_{10}^*(B_{10})$, then the problem would be solved for all possible budget levels, not just \$3.4 M and \$4.2 M.

Although this example has a continuous state space, a finite number of ranges can be constructed because of the zero–one nature of the decision variables. In fact, all breaks in the range of the state space either are the breaks from the previous stage, or they result from adding the cost of the new program to the breaks in the previous range. This is not a general property of continuous state space problems, and in most cases such ranges cannot be determined. Usually, what is done for continuous state space problems is that they are converted into discrete state problems by defining an appropriate grid on the continuous state space. The optimal-value function is then computed only for the points on the grid. For our cost/benefit example, the total budget must be between zero and \$62.5 M, which provides a range on the state space, although at any stage a tighter upper limit on this range is determined by the sum of the budgets of the first n programs. An appropriate grid would consist of increments of \$0.1 M over the limits of the range at each stage, since this is the accuracy with which the program costs have been estimated. The difference between problems with continuous state spaces and those with discrete state spaces essentially then disappears for computational purposes.

11.7 DYNAMIC PROGRAMMING UNDER UNCERTAINTY

Up to this point we have considered exclusively problems with deterministic behavior. In a deterministic dynamic-programming process, if the system is in state s_n with n stages to go and decision d_n is selected from the set of permissible decisions for this stage and state, then the stage return $f_n(d_n, s_n)$ and the state of