Dynamic Programming Methods

Although all dynamic programming (DP) problems have a similar structure, the computational procedures used to find solutions are often quite different. Unlike linear and integer programming where standard conventions are used to describe parameters and coefficients, there is no common data structure that unifies all DPs. Models are problem specific, and for the most part, are represented by recursive formulas rather than algebraic expressions. For this reason it is difficult to create a general computer code that will solve all problems. It is usually necessary to write one's own code to solve a specific application. This is one of the reasons why dynamic programming is much less popular than, say linear programming, where models can be created independently of the software leaving the coding of algorithms to the professionals[1].

Dynamic programming is, however, much more general than linear programming with respect to the class of problems it can handle. As the examples in the previous chapter suggest, there is no difficulty with the integer restrictions on the decision variables, and there is no requirement that any of the functions be linear or even continuous. There is also never any question of local optima; for a correctly formulated problem, a dynamic programming algorithm will always report a global optimum when it terminates.

There is some ambiguity as to the origins of DP but the ideas date back at least to the work of Massé in the mid-1940s. Massé was a French engineer who developed and applied a fairly analytical version of DP to hydropower generation. The standard literature, though, credits Richard Bellman with developing the theoretical foundations of the field and proposing the first algorithms. His early research, published in the 1950s, was aimed at solving problems that arise in the control of continuous systems such as aircraft in flight and electronic circuits. Most traditional expositions follow Bellman's original ideas and describe what is called *backward recursion* on the exhaustive state space. This is perhaps the broadest of the solution methodologies but the least efficient in many instances. We begin with a full description of backward recursion and then move on to forward recursion and reaching. We close the chapter with a limited discussion of stochastic models in which the decision outcomes are viewed as random events governed by known probability distributions.

20.1 Components of Solution Algorithms

The central requirement of a dynamic programming model is that the optimal sequence of decisions from any given state be independent of the sequence that leads up to that state. All computational procedures are based on this requirement known as the *principle of optimality* so the model must be formulated accordingly. If we consider the single machine scheduling problem with due dates discussed in Section 19.6 of the DP Models chapter, we see that the optimal decision at a particular state does not depend on the order of the

[1] The methods of this chapter are implemented in the Teach DP Excel add-in which is quite general with respect to the class of problems it can solve.

scheduled jobs but only on their processing times. To calculate the objective function component $c(d, t)$ given in Table 22 of that chapter, we need to first calculate

$$t = \prod_{j=1}^{n} s_j p(j)) + p(d)$$

for any decision $d \quad \boldsymbol{D}(\mathbf{s})$. This calculation is not sequence-dependent so the principle applies.

Now consider an extension of this problem involving a changeover cost $o(i, j)$ that is incurred when the machine switches from job i to job j. The new objective function component is $c(d, t) + o(i, d)$, where i is the last job in the optimal sequence associated with the current state. This modification invalidates the DP formulation in Table 22 because the current and future decisions depend on the sequence of past decisions. However, if we alter the definition of a state to include an additional component corresponding to the last job in the current sequence, we obtain a valid model similar to the one for the traveling salesman problem in Table 23. The new state variable ranges from 1 to n so the price of the modification is an order n increase in the size of the state space.

The general model structure given in Table 4 in Section 19.2 implies that many optimization problems can be formulated as dynamic programs. In this section, we introduce the basic concepts associated with solution algorithms. Although the algorithms are appropriate for most of the models in Chapter 19, their computational complexity may vary from simple polynomial functions of the number of state variables n to impractically large exponential functions of n.

The Principle of Optimality

For dynamic programming solution methods to work, it is necessary that the states, decisions, and objective function be defined so that the principle of optimality is satisfied. Very often a problem modeled in these terms in an apparently reasonable way will not satisfy this requirement. In such cases, applying the computational procedures will likely produce solutions that are either suboptimal or infeasible. Fortunately, it is possible to model most problems so that the principle holds. As our discussion of the single machine scheduling problem indicates, obtaining a valid model from an invalid one very often involves defining a more complex state space. Theoretically, there is no difficulty in doing this but the analyst must keep in mind that as the size of the state space grows, the computational burden can become prohibitive.

The principle of optimality, first articulated by Bellman, may be stated in a number of ways. We provide several versions in an attempt to further clarify the concept.

1. The optimal path from any given state to a final state depends only on the identity of the given state and not on the path used to reach it.

2. An optimal policy has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

3. An optimal policy must contain only optimal subpolicies (Kaufmann 1967).

4. A policy is optimal if at a stated period, whatever the preceding decisions may have been, the decisions still to be taken constitute an optimal policy when the result of the previous decisions is included.

The principle of optimality seems so obvious that it requires no proof. Indeed the proof is simply that if a solution contains a nonoptimal sequence, it cannot be optimal since it could be improved by incorporating the optimal partial sequence rather than the nonoptimal one. The danger is that in formulating a dynamic programming model there is no clear way of seeing or testing whether or not the principle is satisfied. Much depends on the creativity and insights of the modeler. Alternatively, an extremely complex model might be proposed that does satisfy the principle but is much too elaborate to be of any practical value.

The Acyclic Decision Network

Dynamic programming models with discrete decision variables and a discrete state space have the characteristics of a directed acyclic decision network. This means that a sequence of decisions cannot encounter the same state (node) twice. The network has no cycles and the states are said to be partially ordered. Thus if we consider two states, either there is no relation between them or one can be identified as preceding the other. We write the relation between two states $\mathbf{s}_i$ and $\mathbf{s}_j$ as $\mathbf{s}_i \ll \mathbf{s}_j$ if state $\mathbf{s}_i$ comes before $\mathbf{s}_j$. For an acyclic problem it is impossible that both

$$\mathbf{s}_i \ll \mathbf{s}_j \quad \text{and} \quad \mathbf{s}_j \gg \mathbf{s}_i$$

hold. Not all states need be related in the manner just described so we call this a partial order rather than a complete order.

To illustrate this concept consider the decision network shown in Fig. 1 for a path problem. The numbers on the nodes are used to identify the states.

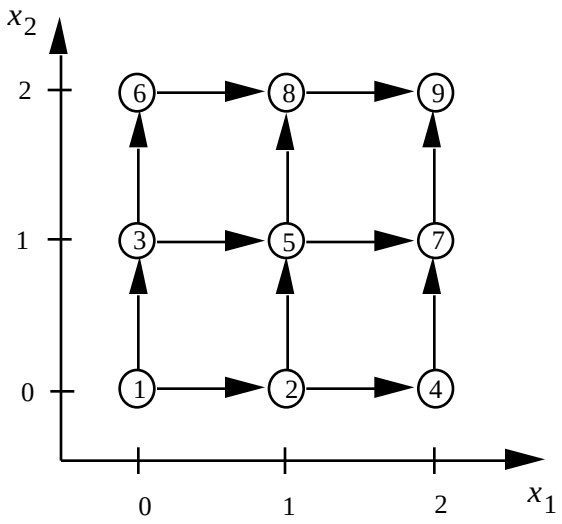


Figure 1. Path problem used to illustrate partial order

The partial order defined by the figure is as follows.

$\mathbf{s}_1 \ll \mathbf{s}_2$ $\mathbf{s}_1 \ll \mathbf{s}_3$ $\mathbf{s}_2 \ll \mathbf{s}_4$ $\mathbf{s}_2 \ll \mathbf{s}_5$ $\mathbf{s}_3 \ll \mathbf{s}_5$

$\mathbf{s}_3 \ll \mathbf{s}_6$ $\mathbf{s}_4 \ll \mathbf{s}_7$ $\mathbf{s}_5 \ll \mathbf{s}_7$ $\mathbf{s}_5 \ll \mathbf{s}_8$ $\mathbf{s}_6 \ll \mathbf{s}_8$

$\mathbf{s}_7 \ll \mathbf{s}_9$ $\mathbf{s}_8 \ll \mathbf{s}_9$

The relationship is transitive in that if $\mathbf{s}_i \ll \mathbf{s}_j$ and $\mathbf{s}_j \ll \mathbf{s}_k$ then $\mathbf{s}_i \ll \mathbf{s}_k$ is also true. Note that several pairs of states in the figure are unrelated such as $\mathbf{s}_2$ and $\mathbf{s}_6$.

A partial order implies that some states must be final states denoted by the set $\boldsymbol{F}$. The corresponding nodes have no originating arcs. State $\mathbf{s}_9$ is the only member of $\boldsymbol{F}$ in Fig. 1. Nodes with no terminating arcs correspond to initial states and are denoted by the set $\boldsymbol{I}$. State $\mathbf{s}_1$ is the sole member of $\boldsymbol{I}$ in the example. The fact that the states have a partial order has important computational implications.

A Requirement for the State Definition

The state $\mathbf{s} = (s_1, s_2, \ldots, s_m)$ is uniquely defined by the values assigned to the state variables. Computational procedures require that the state values perform an additional function -- that they define a complete ordering that includes the partial ordering determined by the decision network. This will be accomplished by ordering the states lexicographically.

Definition 1: Let $\mathbf{a} = (a_1, a_2, \ldots, a_m)$ and $\mathbf{b} = (b_1, b_2, \ldots, b_m)$ be two vectors of equal size. Vector **a** is said to be *lexicographically larger* than **b**

if the first component in which the two vectors differ, call it i, has $a_i > b_i$. Similarly, if $a_i < b_i$, then **a** is said to be *lexicographically smaller* than **b**.

We use the symbols $\overset{L}{>}$ and $\overset{L}{<}$ to describe the relations of 'lexicographically larger than' and 'lexicographical smaller than,' respectively. As usual, the symbols > and < are used to indicate the relative magnitude of numbers. Formally,

$$\mathbf{a} \overset{L}{>} \mathbf{b} \text{ if } a_k = b_k \text{ for } k = 1, \ldots, i-1, \text{ and } a_i > b_i \text{ for some } i \leq m.$$

Note that the relation between a_k and b_k for $k > i$ is not material to this ordering.

For computational purposes, we add the additional restriction that the states must be defined so that their lexicographic order includes the order implied by the decision network of the problem. That is, if $\mathbf{s}_i$ and $\mathbf{s}_j$ are two states with $\mathbf{s}_i \ll \mathbf{s}_j$ in the decision network, the lexicographic order of the vector representation of the states must be such that $\mathbf{s}_i \overset{L}{<} \mathbf{s}_j$. This is a restriction on the state definition.

To illustrate these concepts, consider Fig. 2 in which the network of the path problem has been rotated 45^o.

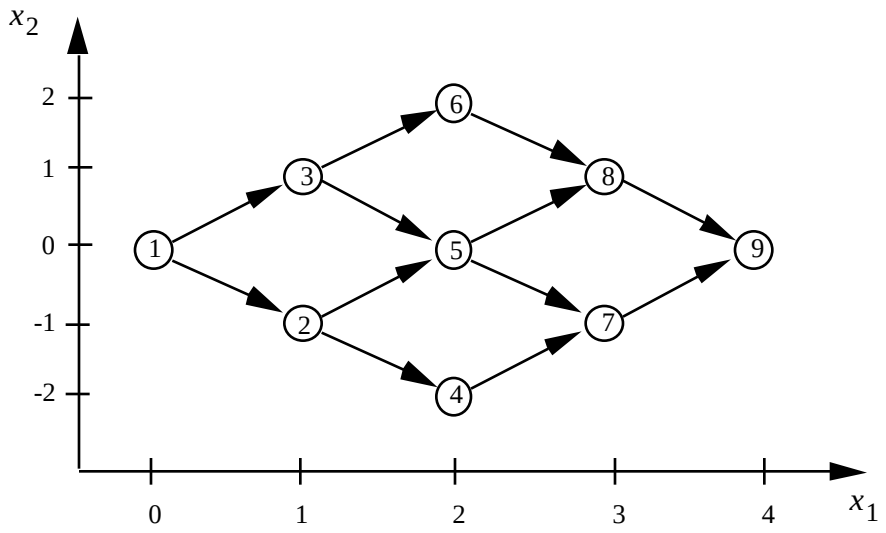


Figure 2. Rotated path problem

Referring to this figure, assume the state definition:

$$\mathbf{s} = (s_1, s_2) \text{ where } s_1 = x_1\text{-coordinate and } s_2 = x_2\text{-coordinate.}$$

The states are listed below in increasing lexicographic order

$$\boldsymbol{S} = \{(0, 0), (1, -1), (1, 1), (2, -2), (2, 0), (2, 2), (3, -1), (3, 1), (4, 0)\}.$$

It is apparent that this ordering includes the ordering represented by the network. The lexicographic order is a complete order and the state order defined by the decision network is a partial order. It is only required that the lexicographic ordering of the state values include the partial order.

As an example of a state definition that does not satisfy the requirement, redefine the state variables of the path problem in Fig. 2 as

$$s_1 = x_2\text{-coordinate and } s_2 = x_1\text{-coordinate.}$$

This state space ordered lexicographically is

$$\boldsymbol{S} = \{(-2, 2), (-1, 1), (-1, 3), (0, 0), (0, 2), (0, 4), (1, 1), (1, 3), (2, 2)\}.$$

It is clear that this order does not include the partial order of the decision network.

It is always possible to define (and arrange) the state variables in a natural way so their values indicate the proper order. For problems involving stages, the requirement is easily fulfilled if the first state variable is the stage number and the stages are numbered in the forward direction.

20.2 State Generation

The simplest dynamic programming algorithms require that the state values, and hence the state space, be generated and stored prior to initiating the computations. More efficient procedures generate states and determine optimal decisions simultaneously. We describe both approaches in later sections, but here we consider the generation process independent of the solution process.

Recall that the m-dimensional vector $\mathbf{s} = (s_1, s_2, \ldots, s_m)$ uniquely identifies a state. To provide procedures for generating and storing states that will work for a wide variety of problem classes and solution techniques, it is necessary to know something about the values that a state variable may assume. Because we are considering only state spaces that are finite, it must be true that the values a particular state variable, say s_i, may assume must be discrete and finite in number. For instance, s_i might take on all integer values from 0 to 10. Alternatively, s_i might assume only even values in the range 0 to 10, or perhaps all values that are multiples of 0.5. Although there will be a finite number of values for a particular state variable, they are not restricted to consecutive integers as is often assumed in elementary discussions.

We now describe two generation procedures. The first will exhaustively generate all states whose values are within specified ranges of evenly spaced intervals. The second will generate only those states that are "reachable" from the initial states in $\boldsymbol{I}$.

Exhaustive Generation

When each state variable s_i is discrete and has a given finite range $R(i)$, the set of all states is simply the set of all possible combinations of the state variable values. Assume $R(i)$ is divided into intervals of equal length and let

$H(i) = \text{Max}\{s_i : i = 1, \ldots, m\}$,

$L(i) = \text{Min}\{s_i : i = 1, \ldots, m\}$,

$d(i)$ = difference between successive values of s_i.

Then $R(i) = H(i) - L(i)$ and there are $R(i)/d(i) + 1$ feasible points within this range. A simple incrementing routine can be used to generate all values of $\mathbf{s}$ over the ranges $R(i)$, $i = 1, \ldots, m$. The total number of states is

$$\prod_{i=1}^{m} \left(\frac{H(i) - L(i)}{d(i)} + 1 \right)$$

If $R(i)$ is not evenly divided, it would be necessary to enumerate each feasible value separately. The same routine, however, could be used to generate the state space.

Forward Generation

In many problems the values of the state variables may not have the regularity required by the exhaustive generation procedure. For example, the range of a state variable may depend on the values of the other state variables. It is also possible that large numbers of states generated may not actually be realizable in a specific problem situation. The set of all states provided by exhaustive generation can be prohibitively large so any reduction that can be achieved by invoking problem-specific logic can lead to significant computational savings.

In this section, we provide an algorithm that only generates states that can be reached from a given set of initial states. A reachable state is one that lies on a path generated by a feasible sequence of decisions starting from a feasible initial state. For the path problem in Fig. 2 only the states shown in the figure are reachable from the initial state (0, 0). States such as (1, 0) are not reachable and so will not be generated.

The algorithm begins with the set of initial states $\boldsymbol{I}$. Referring to the network in Fig. 2, $\boldsymbol{I} = \{(0, 0)\}$. From each initial state, all possible decisions are considered. The combination of decision **d** and state **s** is used in the transition function to define a new reachable state: $\mathbf{s}' = T(\mathbf{s}, \mathbf{d})$. This state is added to the state space if it is feasible and has not been generated previously. When all the decisions for the first initial state have been considered and all states reachable from that state have been generated, the procedure chooses the next lexicographically larger state and generates all feasible states reachable from it.

The computations continue until all states have been considered, and will eventually terminate due to the finiteness of the state space and the use of lexicographical ordering. The process is called forward generation because states are generated using the forward transition function. Note that a state $\mathbf{s}'$ generated by the transition function will always be lexicographically larger than the base state **s** from which it is generated. This assures that no reachable states are missed.

Algorithm

Step 1. Start with a set of initial states $\boldsymbol{I}$ and store them in memory. Let the state space $\boldsymbol{S} = \boldsymbol{I}$. Because of our labeling convention, the states in $\boldsymbol{I}$ are lexicographically smaller than any states that can follow them on a decision path. Let **s** be the lexicographically smallest state in $\boldsymbol{S}$.

Step 2. Find the set of decisions $\boldsymbol{D}(\mathbf{s})$ associated with **s**. For each decision $\mathbf{d} \in \boldsymbol{D}(\mathbf{s})$ find the new state $\mathbf{s}'$ using the transition function $\mathbf{s}' = T(\mathbf{s}, \mathbf{d})$. Search the state space to determine if $\mathbf{s}'$ has already been generated. If so, continue with the next decision. If not, add the state to the state space; that is, put $\boldsymbol{S} \leftarrow \boldsymbol{S} \cup \{\mathbf{s}'\}$. After all decisions associated with **s** have been considered go to Step 3.

Step 3. From those elements in $\boldsymbol{S}$, find the next lexicographically larger state than $\mathbf{s}$. If there is none, stop, the state space is complete. Otherwise let this state be $\mathbf{s}$ and go to Step 2.

For the network in Fig. 2, Table 1 highlights the results of the algorithm at each iteration. The asterisk (*) in each column identifies the base state $\mathbf{s}$ as the algorithm enters Step 2. The states following the base state are those generated at the current iteration.

Table 1. Sequence of states obtained by forward generation

Iteration #								
1	2	3	4	5	6	7	8	9
(0,0)*	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)
	(1,–1)*	(1,–1)	(1,–1)	(1,–1)	(1,–1)	(1,–1)	(1,–1)	(1,–1)
	(1,1)	(1,1)*	(1,1)	(1,1)	(1,1)	(1,1)	(1,1)	(1,1)
		(2,–2)	(2,–2)*	(2,–2)	(2,–2)	(2,–2)	(2,–2)	(2,–2)
		(2,0)	(2,0)	(2,0)*	(2,0)	(2,0)	(2,0)	(2,0)
			(2,2)	(2,2)	(2,2)*	(2,2)	(2,2)	(2,2)
				(3,–1)	(3,–1)	(3,–1)*	(3,–1)	(3,–1)
					(3,1)	(3,1)	(3,1)*	(3,1)
							(4,0)	(4,0)*

From the table we see how the state space grows as the algorithm proceeds. The complete state space is obtained at iteration 9. This event is signaled by the fact that there is no state lexicographically greater than (4,0).

The validity of this procedure depends on the fact that if $\mathbf{s}'$ is generated from $\mathbf{s}$ using the transition function $T(\mathbf{s}, \mathbf{d})$, we are assured that $\mathbf{s}' \succ \mathbf{s}$. Also, since all states that have yet to be considered for expansion are lexicographically greater than $\mathbf{s}$, none of them can create a state smaller than $\mathbf{s}$. Therefore, considering the states in the order prescribed by the algorithm will generate all states that are reachable from the initial states in $\boldsymbol{I}$.

A second example illustrates the benefits that can be achieved in state space reduction by using the forward generation procedure instead of the exhaustive procedure. To begin, consider the following resource allocation problem with binary decision variables. No specific values for the objective coefficients are given because they have no bearing on the size of the state space.

Maximize $z = \sum_{j=1}^{10} c_j x_j$

subject to

$$9x_1 + 7x_2 + 6x_3 + 8x_4 + 3x_5 + 3x_6 + 5x_7 + 5x_8 + 7x_{10} \le 20$$
$$9x_1 + 4x_2 + 5x_3 + x_4 + 8x_5 + x_6 + 4x_7 + 8x_8 + 7x_9 \le 20$$
$$x_2 + 8x_3 + 6x_4 + 9x_5 + 6x_6 + 5x_7 + 7x_8 + 7x_9 + 3x_{10} \le 20$$
$$x_j = 0 \text{ or } 1,\ j = 1,\ldots,10$$

For a resource allocation problem with m constraints and n variables, the number of state variables is $m + 1$. The constraints represent limits on resource usage. In the DP model presented in Chapter 19 for this problem, the first state variable s_1 represents the index of the x variables. In general, s_1 ranges from 1 to n+1, with $s_1 = n$+1 signaling a final state. For the example, $1 \le s_1 \le 11$. State variables 2 through $m + 1$ correspond to constraints 1 through m, respectively, with s_i indicating the amount of the (i–1)th resource b_{i-1} used by a partial solution prior to the current decision. When the exhaustive procedure is used, the total number of states is $|\boldsymbol{S}| = n(b_1+1)\cdots(b_m+1)$ which is an exponential function of m. This value is (11)(21)(21)(21) = 101,871 for the example[2].

To highlight the differences in the size of the state space as more constraints are added, we first consider the problem with only constraint 1, then with constraints 1 and 2, and finally with all three constraints.

For the forward generation procedure the initial state is $\mathbf{s}$ = (1, 0) for the one constraint case, $\mathbf{s}$ = (1, 0, 0) for the two constraint case and $\mathbf{s}$ = (1, 0, 0, 0) for the three constraint case. Table 2 shows the number of states identified using the exhaustive generation and forward generation methods. If we consider the latter, there are only two states reachable from $\mathbf{s}$ = (1, 0, 0, 0). To see this, we use the transition function from Table 5 in the DP Models chapter to determine $\mathbf{s}'$. Given $\mathbf{s}$ = (1, 0, 0, 0), two decisions are possible, either d = 0 implying item 1 is excluded (x_1 = 0) or d = 1 implying item 1 is included (x_1 = 1). When d = 0 the state is $\mathbf{s}'_1$ = (2, 0, 0, 0). Alternatively, when d = 1 the state is $\mathbf{s}'_2$ = (2, 9, 9, 0). All other states of the form (2, s_2, s_3, s_4), where $1 \le s_2, s_3, s_4 \le 20$, are unreachable from (1, 0, 0, 0) and so are not generated. The exhaustive procedure enumerates them all.

[2] The Teach DP add-in exhaustive enumeration option includes only the initial states for s_1 = 1. It computes 92,611 states for this example.

Table 2. Comparison of generation methods

Number of constraints	Exhaustive generation of states	Forward generation of states
1	231	130
2	4,851	409
3	101,871	436

The results in Table 2 clearly demonstrate the exponential growth in the state space when the exhaustive procedure is used, underscoring the impracticality of this approach for all but the smallest knapsack problems. Computation times are generally proportional to the number of states. Thus a large state space is expensive in both computer time and storage.

Considering only reachable states as in forward generation yields a much smaller state space. Although the growth indicated in Table 2 is modest, it can still be exponential in the worst case so it would be inappropriate to generalize from the example. Nevertheless, it is interesting to consider the effect of the number of constraints on the number of feasible solutions and the number of states required to represent them. When there are no constraints the binary allocation problem has 2^n feasible solutions, where n is the number of variables. When a single constraint is added, the number of feasible solutions must be reduced if the constraint has any effect at all. As additional constraints are added, the number of solutions that satisfies all the constraints must decline or at least grow no larger. Thus despite the fact that the dynamic programming model becomes all but impossible to solve when only a few constraints are present, the actual number of solutions that are candidates for the optimum declines with the number of constraints.

The forward generation procedure only generates feasible states. The reason this number does not decline with the number of constraints is that fewer feasible decision sequences can be represented by the same states. With a single constraint a large number of feasible sequences or partial sequences may have the same resource usage. With two constraints, two solutions that use the save amount of the first resource may use different amounts of the second resource thus generating two states rather than one.

No general statement can be made concerning the growth in the number of reachable states as a function of the number of constraints. It is generally true, however, that the number of reachable states will be considerably less than the number of states generated exhaustively. The savings is at a cost of additional computer time, since forward generation requires repeated use of the transition function while exhaustive generation does not. Whether the expense is justified depends on the complexity of the transition function and the number of reachable states. For some problem classes forward generation will result in no savings at all, while for others it will be decidedly effective.

20.3 Backward Recursion

The principle of optimality leads to the computational procedures of dynamic programming. The most common, called backward recursion, is described in this section. As defined in the DP Models chapter, the problem is to find a sequence of alternating states and decisions from an initial state in $\boldsymbol{I}$ to a final state in $\boldsymbol{F}$. The sequence is called a path. Our goal is to determine the optimal path.

The Recursive Equation

Let n be the dimensions of the decision space and let $\mathbf{P}_j$ denote the jth path through the state space; $\mathbf{P}_j$ is identified by the sequence $\mathbf{s}_{1j}$, $\mathbf{d}_{1j}$, $\mathbf{s}_{2j}$, $\mathbf{d}_{2j}, \ldots, \mathbf{s}_{nj}$, $\mathbf{d}_{nj}$, $\mathbf{s}_{n+1,j}$, where $\mathbf{s}_{n+1,j}$ is a final state. Assume for purposes of presentation an additive objective function of the form

$$z(\mathbf{P}_j) = \sum_{i=1}^{n} z(\mathbf{s}_{ij}, \mathbf{d}_{ij}) + f(\mathbf{s}_{n+1,j}).$$

We assume our goal is to minimize the objective.

The optimal path $\mathbf{P}^*$ is the one that solves the following problem

$$z(\mathbf{P}^*) = \text{Minimize}_{j \in J} z(\mathbf{P}_j)$$

where J is the set of all feasible paths. However, this is only a conceptual representation of the problem because the set J is not known. We do not search explicitly over all feasible paths but try to generate the optimal path algorithmically. Therefore, there is no need to refer to a specific path j in the presentation so we will drop the subscript j from the state and decision vectors.

Define $f(\mathbf{s})$ as the length of the optimal path from state $\mathbf{s}$ to a final state $s_F \in \boldsymbol{F}$. The initial state is $\mathbf{s}_1$ and the initial decision is $\mathbf{d}_1$. The state following $\mathbf{d}_1$ is determined by the transition function to be

$$\mathbf{s}_2 = T(\mathbf{s}_1, \mathbf{d}_1) \qquad (1)$$

The principle of optimality says that whatever the first decision is the remaining decisions should be made optimally. Thus given $\mathbf{d}_1$, the cost of the optimal path is

$$r(\mathbf{s}_1, \mathbf{d}_1) \equiv z(\mathbf{s}_1, \mathbf{d}_1) + f(\mathbf{s}_2)$$

where $\mathbf{s}_2$ is determined by Eq. (1). The optimal initial decision $\mathbf{d}_1$ is found by evaluating $r(\mathbf{s}_1, \mathbf{d})$ for all feasible decisions $\mathbf{d} \in \boldsymbol{D}(\mathbf{s}_1)$ and choosing the one that yields the minimum cost. Thus we must solve the following optimization problem, typically by enumerating all elements of $\boldsymbol{D}(\mathbf{s}_1)$.

$$f(\mathbf{s}_1) = \text{Minimize}\{f(\mathbf{s}_1, \mathbf{d}) : \mathbf{d} \in \boldsymbol{D}(\mathbf{s}_1)\}$$

$$= \text{Minimize}\{z(\mathbf{s}_1, \mathbf{d}) + f(\mathbf{s}_2) : \mathbf{d} \quad \boldsymbol{D}(\mathbf{s}_1)\} \tag{2}$$

What is curious about problem (2) is that $f(\mathbf{s}_1)$ depends on $f(\mathbf{s}_2)$. Functions that depend on themselves are called "recursive." To solve this recursive equation for $\mathbf{s}_1$ it is necessary to know the values of $f(\cdot)$ at all states $\mathbf{s}_2$ that can be reached from $\mathbf{s}_1$ with a feasible decision. A similar recursive equation can be derived for each successor state. In general, the recursive equation for state $\mathbf{s}$ can be written as

$$f(\mathbf{s}) = \text{Minimize}\{r(\mathbf{s}, \mathbf{d}) : \mathbf{d} \quad \boldsymbol{D}(\mathbf{s})\}$$

where $r(\mathbf{s}, \mathbf{d}) = z(\mathbf{s}, \mathbf{d}) + f(\mathbf{s}')$

and $\mathbf{s}' = T(\mathbf{s}, \mathbf{d})$

so $$f(\mathbf{s}) = \text{Minimize}\{z(\mathbf{s}, \mathbf{d}) + f(\mathbf{s}') : \mathbf{d} \quad \boldsymbol{D}(\mathbf{s})\}. \tag{3}$$

We call $z(\mathbf{s}, \mathbf{d})$ the decision objective, $r(\mathbf{s}, \mathbf{d})$ the path objective for decision $\mathbf{d}$ taken from state $\mathbf{s}$, and $f(\mathbf{s})$ the optimal value function for state $\mathbf{s}$. Equation (3) is referred to as the *recurrence relation*.

Solving the Recursive Equation

For an acyclic network the recursive equation can be easily solved by starting at the final states in the set $\boldsymbol{F}$ and working backwards. Before beginning the computations, it is necessary to specify the *boundary conditions* which follow directly from the statement of the problem or from the definition of f. When working backwards, values of $f(\cdot)$ must be assigned for all $\mathbf{s}_F \quad \boldsymbol{F}$. For an additive path objective function, we often have $f(\mathbf{s}_F) = 0$ but the assignment might actually be a function of $\mathbf{s}_F$.

Once the values of the terminal states are known, the recursive equation (3) can be solved for all states whose decisions lead only to final states. For an acyclic network, there must be some states that have this property. The process continues until the optimal value function $f(\mathbf{s})$ is determined for each $\mathbf{s} \quad \boldsymbol{S}$. When the state variables are defined so that they satisfy the requirements of Section 20.1, the recursive equations can be solved by considering the states in reverse lexicographic order.

Associated with each state is an optimal decision or policy, $\mathbf{d}^*(\mathbf{s})$. This is the decision that optimizes the recursive equation. The function of optimal decisions defined over the state space is called the optimum policy function.

The process of solving Eq. (3) for an acyclic network is called backward recursion. The procedure moves backwards through the decision network (in the reverse direction of the arcs) until the optimal path value and accompanying decision is found for every state.

Algorithm for Backward Recursion

Step 1. Find the state with the largest lexicographic value. Let this be state $\mathbf{s}$.

Step 2. If the state is a final state, assign it a value $f(\mathbf{s})$ by a rule or equation peculiar to the problem being solved. If it is not a final state, solve recursive equation (3) for $f(\mathbf{s})$ to obtain $\mathbf{d}^*(\mathbf{s})$. Store these values for future use.

Step 3. Find the next lexicographically smaller state than $\mathbf{s}$.

i. If none exists, stop, the backward recursion is complete. Solve

$$f(\mathbf{s}_1) = \text{Minimize}\{f(\mathbf{s}) : \mathbf{s} \quad \boldsymbol{I}\}$$

to determine the optimal path value and the initial state $\mathbf{s}_1$ from which the optimal path originates. Perform forward recovery to determine the optimal sequence of states and decision.

ii. If another state is found, let it be $\mathbf{s}$ and go to Step 2.

Forward Recovery

The algorithm finds the optimal path from every state to a final state but is available only implicitly through the policy function. Given an initial state $\mathbf{s}_1$, the first decision on the optimal path is $\mathbf{d}^*(\mathbf{s}_1)$. The next state on the optimal path is found by evaluating the transition function

$$\mathbf{s}_2 = T(\mathbf{s}_1, \mathbf{d}^*(\mathbf{s}_1))$$

to get $\mathbf{s}_2$. The next decision is the optimal policy for $\mathbf{s}_2$ given by $\mathbf{d}^*(\mathbf{s}_2)$. These operations proceed in an iterative manner, sequentially using the transition function and the optimal policy function until a final state is encountered (there may be several final states but we are only interested in the one on the optimal path). We call this *forward recovery* of the optimum because the calculations proceed in the forward direction (in the same direction as the arcs) through the decision network. Thus backward recursion must be followed by forward recovery. Note that the optimal path is available for every state. Dynamic programming solves not one problem but many problems, one for every $\mathbf{s}$ in $\boldsymbol{S}$. This is why it is so important to reduce the size of the space as much as possible.

Summary of Notation Associated with Backward Recursion

Optimal value function:

$f(\mathbf{s})$ = value of the optimal path from state $\mathbf{s}$ to any final state

Path return:

$r(\mathbf{s}, \mathbf{d})$ = value of the path starting from state $\mathbf{s}$ with the first decision $\mathbf{d}$ such that subsequent decisions are optimal with respect to $\mathbf{s}'$, the state reached from $\mathbf{s}$ with decision $\mathbf{d}$

Optimal policy or decision for state $\mathbf{s}$:

$$\mathbf{d}^*(\mathbf{s}) \text{ where } f(\mathbf{s}) = r(\mathbf{s}, \mathbf{d}^*(\mathbf{s}))$$

Tabular Format for Backward Recursion

To organize the computations, we introduce a tabular format that summarizes the results at each iteration. All the information necessary to perform the steps of backward recursion and recover the optimal solution upon termination is included. The column headings are listed below. Of course, other formats could be adopted for specific problem classes.

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
Index	$\mathbf{s}$	$\mathbf{d}$	$\mathbf{s}'$	z	$f(\mathbf{s}')$	$r(\mathbf{s}, \mathbf{d})$	$f(\mathbf{s})$	$\mathbf{d}^*(\mathbf{s})$

Col. (1) Index: This column assigns a numerical index to each state as it is considered. The number 1 will be assigned to the first state, 2 to the second state, and so on.

Col. (2) $\mathbf{s}$: This is the vector of state variables describing the state under consideration.

Col. (3) $\mathbf{d}$: This is the vector of decision variables describing the decision under consideration. For each value of $\mathbf{s}$, all the decisions in the set $\boldsymbol{D}(\mathbf{s})$ will be listed.

Col. (4) $\mathbf{s}'$: This is the state that is entered when decision $\mathbf{d}$ is taken in state $\mathbf{s}$. It is determined by the transition function $T(\mathbf{s}, \mathbf{d})$.

Col. (5) z: This is the decision objective. It will, in general, be a function of $\mathbf{s}$, $\mathbf{d}$ and $\mathbf{s}'$.

Col. (6) $f(\mathbf{s}')$: This is the optimal value function for state $\mathbf{s}'$. Given a current state $\mathbf{s}$, its value will have been determined at a previous iteration.

Col. (7) $R(\mathbf{s}, \mathbf{d})$: This is the path return obtained by making decision $\mathbf{d}$ in state $\mathbf{s}$ and thereafter following the optimal path from state $\mathbf{s}'$. With an additive path objective, this column will be the sum of columns (5) and (6).

Col. (8) $f(\mathbf{s})$: This is the optimal value function for state $\mathbf{s}$. It is obtained by taking the minimum (or maximum) of the values in Col. (7) for a state $\mathbf{s}$, and provides the solution to the recursive equation.

Col. (9) $\mathbf{d}^*(\mathbf{s})$: This is the optimal policy for state $\mathbf{s}$. It is the decision for which the optimum was obtained in determining the value in Col. (8).

Example 1

To illustrate the use of this format, the simple path problem in Fig. 3 is solved to find the shortest path from state (0, 0) to state (4, 0). Although we considered this problem in Chapter 19, the model for the rotated problem is slightly different because we have added a component to describe the recursive equation (see Table 3).

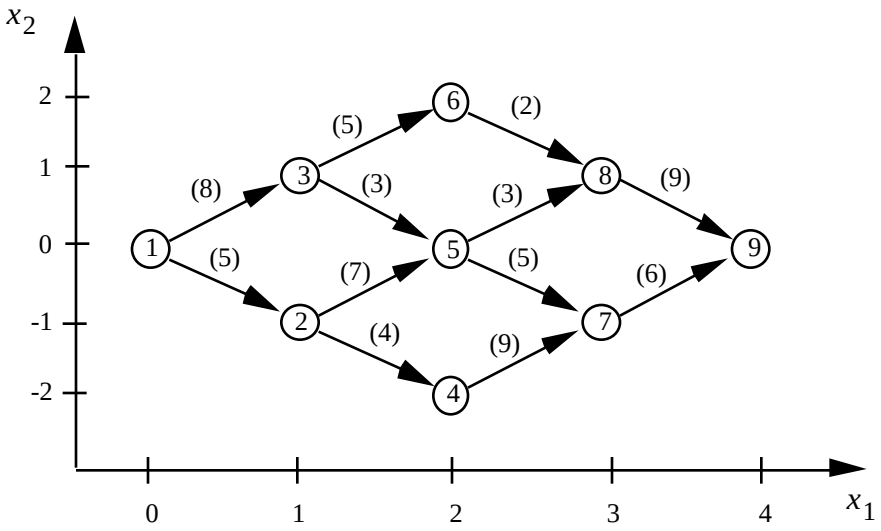


Figure 3. Rotated path problem with arc lengths assigned

Table 3. Backward recursion model for rotated path problem

Component	Description
State	The states are the nodes of the network identified by their coordinates. $\mathbf{s} = (s_1, s_2)$, where $s_1 = x_1$-coordinate and $s_2 = x_2$-coordinate
Initial state set	$\boldsymbol{I} = \{(0, 0)\}$
Final state set	$\boldsymbol{F} = \{(4, 0)\}$
State space	$\boldsymbol{S} = \{(0, 0), (1, -1), (1, 1), (2, -2), (2, 0), (2, 2), (3, -1), (3, 1), (4, 0)\}$
Decision	$\mathbf{d} = (d)$, where d indicates the direction traveled $d = \begin{cases} +1 & \text{go up at } +45^o \\ -1 & \text{go down at } -45^o \end{cases}$
Feasible decision set	$\boldsymbol{D}(\mathbf{s}) = \{+1, -1 : \mathbf{s}' \in \boldsymbol{S}\}$

Transition function	$\mathbf{s}' = T(\mathbf{s}, \mathbf{d})$ where $s_1' = s_1 + 1$ and $s_2' = s_2 + d$
Decision objective	$z(\mathbf{s}, \mathbf{d}) = a(\mathbf{s}, d)$, arc length indicated in Fig. 3
Path objective	Minimize $z(\mathbf{P}) = \sum_{\mathbf{s} \in \mathbf{S}, \mathbf{d} \in \mathbf{D}(\mathbf{s})} z(\mathbf{s}, \mathbf{d}) + f(\mathbf{s}_f)$ where $\mathbf{s}_f$ is some state in $\boldsymbol{F}$
Final value function	$f(\mathbf{s}) = 0$ for $\mathbf{s} \in \boldsymbol{F}$ Specification of boundary conditions.
Recursive equation	$f(\mathbf{s}) = \text{Minimize}\{a(\mathbf{s}, d) + f(\mathbf{s}') : d \in \mathbf{D}(\mathbf{s})\}$

The calculations for the example are given in Table 4 which is constructed by listing the states in the order in which they must be considered for backward recursion; that is, in reverse lexicographic order. For each state, all the feasible decisions are listed together with all the information necessary to solve the recursive equation. For example, let us consider $\mathbf{s} = (3, 1)$ which is node 8 in Fig. 3. The only feasible decision at this node is $d = -1$ so it is not necessary to solve Eq. (3). The transition leads to $\mathbf{s}' = (4, 0)$ which is a final state. The return function is computed as follows: $r(\mathbf{s}, \mathbf{d}) = a(\mathbf{s}, d) + f(\mathbf{s}') = a((3, 1), -1) + f((4, 0)) = 9 + 0 = 9$. Thus $f(3, 1) = 9$. The same analysis applies at $\mathbf{s} = (3, -1)$. At $\mathbf{s} = (2, 0)$, the computations are more informative because there are two possible decisions. The components of the transition function are $s_1' = s_1 + 1 = 3$ and $s_2' = s_2 + d = d$ so when $d = 1$, $\mathbf{s}' = (3, 1)$ and when $d = -1$, $\mathbf{s}' = (3, -1)$. We now compute the corresponding return functions. For

$$d = 1: \; r((2, 0), 1) = a((2, 0), 1) + f(3, 1) = 3 + 9 = 12,$$

and for

$$d = -1: \; r((2, 0), -1) = a((2, 0), -1) + f(3, -1) = 5 + 6 = 11.$$

Solving recursive equation (3) gives

$$f(2, 0) = \text{Minimize}\begin{Bmatrix} r((2,0),1) = 12 \\ r((r,0),-1) = 11 \end{Bmatrix} = 11$$

implying that the optimal decision vector at this state is $\mathbf{d}^*(2, 0) = -1$.

In general, when all the relevant values of $r(\mathbf{s}, \mathbf{d})$ are available, the recursive equation (3) is solved to find the optimal value function $f(\mathbf{s})$ and the optimal policy $\mathbf{d}^*(\mathbf{s})$ for each state. The bold arrows in Fig. 4 depict the optimal policy at each node.

Table 4. Backward recursion for the path problem

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
Index	**s**	**d**	**s**'	z	$f(\mathbf{s}')$	$r(\mathbf{s}, \mathbf{d})$	$f(\mathbf{s})$	$\mathbf{d}^*(\mathbf{s})$
1	(4, 0)	—					0	∞
2	(3, 1)	–1	(4, 0)	9	0	9	9	–1
3	(3, –1)	1	(4, 0)	6	0	6	6	1
4	(2, 2)	–1	(3, 1)	2	9	11	11	–1
5	(2, 0)	1	(3, 1)	3	9	12		
		–1	(3, –1)	5	6	11	11	–1
6	(2, –2)	1	(3, –1)	9	6	15	15	1
7	(1, 1)	1	(2, 2)	5	11	16		
		–1	(2, 0)	3	11	14	14	–1
8	(1, –1)	1	(2, 0)	7	11	18	18	1
		–1	(2, –2)	4	15	19		
9	(0, 0)	1	(1, 1)	8	14	22	22	1
		–1	(1, –1)	5	18	23		

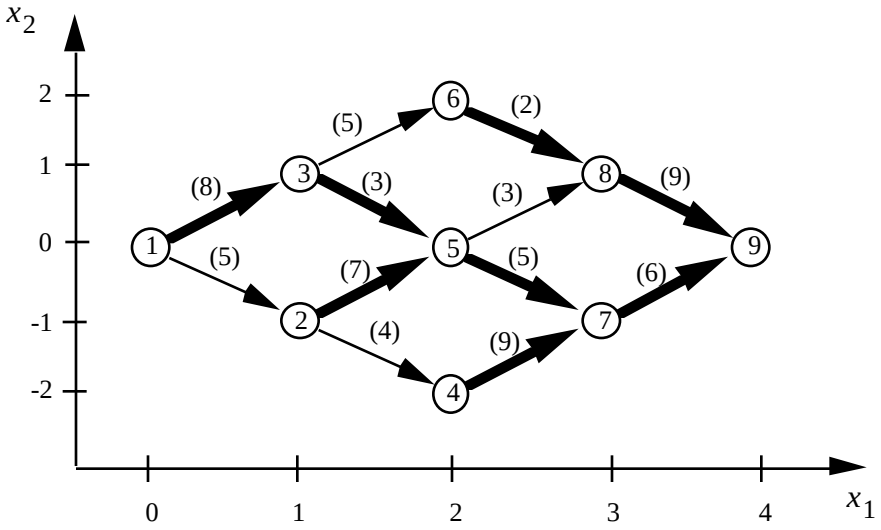


Figure 4. Optimal policy using backward recursion

Once the table has been constructed, the only remaining task is to recover the optimal path from the initial node. Its length has already been determined and appears as the last entry in column (8). In particular, $f(0,0) = 22$. Figure 4 shows that an optimal path is available for *every* state to a final state. This is always the case when backward recursion is used.

The forward recovery algorithm can also be implemented in tabular format with the following four columns: $\mathbf{s}$, $\mathbf{d}^*(\mathbf{s})$, $\mathbf{s}'$ and $z(\mathbf{P}^*)$. Table 5 shows the results for the example. The first entry for $\mathbf{s}$ is a state in the initial set $\boldsymbol{I}$. The value of $\mathbf{d}^*(\mathbf{s})$ for this state is found in Table 4 to be 1. The transition equation then determines the next state $\mathbf{s}'$ which is equal to (1,1). This state is listed in the third column of the table and also as $\mathbf{s}$ in the next row. The forth column lists the cumulative path length, where it is seen once again that $z(\mathbf{P}^*) = 22$. The process continues until $\mathbf{s}'$ is a member of the final state set $\boldsymbol{F}$. When this event occurs, the solution has been completely determined. The optimal path can be read from the first and second columns in the table. The solution process of backward recursion is always followed by forward recovery.

Table 5. Forward recovery

$\mathbf{s}$	$\mathbf{d}^*(\mathbf{s})$	$\mathbf{s}'$	$z(\mathbf{P}^*)$
(0, 0)	1	(1, 1)	—
(1, 1)	–1	(2, 0)	8
(2, 0)	–1	(3, –1)	11
(3, –1)	1	(4, 0)	16
(4, 0)	—	—	22

20.4 Forward Recursion

Instead of starting at a final state and working backwards, for many problems it is possible to determine the optimum by an opposite procedure called forward recursion. Backward recovery is then used to identify the optimal path. The procedure may be more convenient for some classes of problems, but for those in which the final state set is unknown, it is the only feasible approach. Forward recursion also leads to a very powerful procedure called reaching, described in the next section.

Backward Decision Set, Transition Function and Decision Objective

The backward decision set is denoted for each state by $\boldsymbol{D}_b(\mathbf{s})$ with the subscript "b" being used to distinguish it from the comparable set $\boldsymbol{D}(\mathbf{s})$ previously defined for backward recursion. $\boldsymbol{D}_b(\mathbf{s})$ is the set of decisions that can be used to enter state $\mathbf{s}$. For the simple rotated path problem depicted in Fig. 3, $\boldsymbol{D}_b(\mathbf{s}) = \{-1, +1\}$, where $d = -1$ implies entering $\mathbf{s}$ from above and $d = +1$ implies entering $\mathbf{s}$ from below. Note that the elements of $\boldsymbol{D}(\mathbf{s})$ and $\boldsymbol{D}_b(\mathbf{s})$ are the same for this path problem but their meaning is quite different; $\boldsymbol{D}(\mathbf{s})$ is the set of decisions that lead out of state $\mathbf{s}$ and $\boldsymbol{D}_b(\mathbf{s})$ is the set of decisions that lead in.

The backward transition function is defined as

$$\mathbf{s}_b = T_b(\mathbf{s}, \mathbf{d}), \text{ where } \mathbf{d} \quad \boldsymbol{D}_b(\mathbf{s}).$$

The function T_b determines the state that came before $\mathbf{s}$ when the decision made to reach state $\mathbf{s}$ is $\mathbf{d}$. We use $\mathbf{s}_b$ to denote the previous state. Very often the backward transition function can be obtained directly from the forward transition function. For some problems, the forward transition equation $\mathbf{s}' = T(\mathbf{s}, \mathbf{d})$ can be solved for $\mathbf{s}$ in terms of $\mathbf{s}'$ and $\mathbf{d}$. Substituting first $\mathbf{s}_b$ for $\mathbf{s}$ and then $\mathbf{s}$ for $\mathbf{s}'$, one obtains the backward transition function. For instance, the rotated path problem has the forward transition function components

$$s'_1 = s_1 + 1 \text{ and } s'_2 = s_2 + d.$$

Solving these equations for $\mathbf{s}$ in terms of $\mathbf{s}'$ yields

$$s_1 = s'_1 - 1 \text{ and } s_2 = s'_2 - d.$$

Substituting first $\mathbf{s}_b$ for $\mathbf{s}$ and then $\mathbf{s}$ for $\mathbf{s}'$, we obtain

$$s_{b1} = s_1 - 1 \text{ and } s_{b2} = s_2 - d.$$

These equations give the backward transition function for the rotated path problem.

For a given definition of the state variables, it is not always true that there exists both forward and backward transition functions for a particular problem. For instance, consider the path problem with turn penalties (Section 19.5). The components of its forward transition function are

$$s'_1 = s_1 + 1,\ \ s'_2 = s_2 + d,\ \ s'_3 = d.$$

Thus it is not possible to solve for s_3 in terms of $\mathbf{s}'$ and d for the simple reason that s_3 does not appear in these equations. A little reflection will show that it is impossible to know what the previous state was (in terms of the third state variable) given the current state and the decision made to reach it. Defining the set of state variables differently, though, will provide both forward and backward transition functions. For the turn penalty problem, the first two state variables remain unchanged, but the third state variable s_{b3} should be defined as the direction to be traveled rather than the direction last traveled.

The backward decision objective, $z_b(\mathbf{s}_b, \mathbf{d}, \mathbf{s})$ is the immediate cost of decision $\mathbf{d}$ when it leads to state $\mathbf{s}$ from state $\mathbf{s}_b$. For most problems of interest, z_b is a simple function of $\mathbf{s}_b$ and $\mathbf{d}$.

Forward Recursive Equation

The forward recursive equation is written

$$f_b(\mathbf{s}) = \text{Minimize}\{r_b(\mathbf{s}, \mathbf{d}) : \mathbf{d} \quad \boldsymbol{D}_b(\mathbf{s})\}$$

where $r_b(\mathbf{s}, \mathbf{d}) = z_b(\mathbf{d}, \mathbf{s}) + f_b(\mathbf{s}_b)$

and $\mathbf{s}_b = T_b(\mathbf{s}, \mathbf{d})$

so
$$f_b(\mathbf{s}) = \text{Minimize}\{z_b(\mathbf{d}, \mathbf{s}) + f_b(\mathbf{s}_b) : \mathbf{d} \quad \boldsymbol{D}_b(\mathbf{s})\}. \qquad (4)$$

The factors in these relationships have analogous definitions to their counterparts in the backward recursive equation. In particular, $f_b(\mathbf{s})$ is the cost of the optimal path arriving at $\mathbf{s}$ from an initial state as a consequence of decision $\mathbf{d}$. Sometimes it is convenient to express the decision return as an explicit function of $\mathbf{s}_b$. In that case, we use the notation $z_b(\mathbf{s}_b, \mathbf{d}, \mathbf{s})$.

The first step in solving (4) is to assign values to the initial states (those with no predecessors). This is equivalent to specifying the boundary conditions. Then the equation can be solved for each state that is an immediate successor of the initial states. The procedure continues in the forward direction until all states have been considered. The policy function $\mathbf{d}^*(\mathbf{s})$ now holds the information on the optimal decision entering each state $\mathbf{s} \quad \boldsymbol{S}$.

The optimum is identified by a backward recovery procedure. Starting at the final state associated with the minimum cost, the policy function identifies the optimal decision that led to that state. The backward transition function then identifies the predecessor state in the optimal path. The policy function, in turn, identifies the optimal decision entering that state. The process continues sequentially using the optimal policy function and the backward transition function until an initial state is encountered. At this point the optimal path to the specified final state from an initial state has been found. Note that the forward recursion procedure implicitly discovers the optimal path to every state from an initial state.

Example 2

For the rotated path problem described in the previous section, Table 6 provides the additional definitions associated with forward recursion. The calculations are shown in Table 7. Note that this table is constructed by listing the states in the order in which they must be considered for forward recursion, that is, in lexicographic order. All the calculations required to obtain $f_b(\mathbf{s})$ and $\mathbf{d}^*(\mathbf{s})$, the optimal path value and the optimum policy, respectively, are included.

Table 6. Forward recursion model for rotated path problem

Component	Description
Backward decision set	$\boldsymbol{D}_b(\mathbf{s}) = \{+1, -1\}$
Backward transition function	$\mathbf{s}_b = T_b(\mathbf{d}, \mathbf{s})$ $s_{b1} = s_1 - 1$ and $s_{b2} = s_2 - d$
Backward decision objective	$z_b(\mathbf{s}, \mathbf{d}) = a(\mathbf{s}, d)$, where $a(\mathbf{s}, d)$ is the length of the arc entering $\mathbf{s}$ from decision d, taken from Fig. 3.
Initial value function	$f(\mathbf{s}) = 0$ for $\mathbf{s} \quad \boldsymbol{I}$ Specification of boundary conditions.
Forward recursive equation	$f_b(\mathbf{s}) = \text{Minimize}\{z_b(\mathbf{s}, \mathbf{d}) + f_b(\mathbf{s}_b) : \mathbf{d} \quad \boldsymbol{D}_b(\mathbf{s})\}$

To better understand how the calculations are performed, assume that we are at $\mathbf{s} = (3,1)$ which is node 8 in Fig. 3. The decision set $\boldsymbol{D}_b(3,1) = \{1,-1\}$ and the backward transition function for each of the two state components is $s_{b1} = s_1 - 1$ and $s_{b2} = s_2 - d$. To solve the recursive equation (4) if is first necessary to evaluate the return function for all d $\boldsymbol{D}_b(3, 1)$. The two cases are as follows.

$d = 1 \quad \mathbf{s}_b = (2,0)$ and

$$r_b((3,1), 1) = a((2,0), 1) + f(2,0) = 3 + 11 = 14$$

$d = -1 \quad \mathbf{s}_b = (2,2)$ and

$$r_b((3,1), -1) = a((2,2), -1) + f(2,2) = 2 + 13 = 15$$

Combining these results leads to

$$f(3,1) = \text{Minimize}\left\{\begin{array}{l} r_b((3,1),1) = 14 \\ r_b((3,1),-1) = 15 \end{array}\right\} = 14$$

so the optimal decision $\mathbf{d}^*(3,1) = 1$ which means it is best to reach (3,1) from (2,0). Figure 5 depicts the optimal policy for each state, which is the arc that should be traversed to enter that state. By implication, an optimal path is available for every state *from* the initial state.

Table 7. Forward recursion results for the rotated path problem

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
Index	**s**	**d**	$\mathbf{s}_b$	z_b	$f_b(\mathbf{s}_b)$	$r_b(\mathbf{s}, \mathbf{d})$	$f_b(\mathbf{s})$	$\mathbf{d}^*(\mathbf{s})$
1	(0, 0)	—					0	∞
2	(1, –1)	–1	(0, 0)	5	0	5	5	–1
3	(1, 1)	1	(0, 0)	8	0	8	8	1
4	(2, –2)	–1	(1, –1)	4	5	9	9	–1
5	(2, 0)	1	(1, –1)	7	5	12		
		–1	(1, 1)	3	8	11	11	–1
6	(2, 2)	1	(1, 1)	5	8	13	13	1
7	(3, –1)	1	(2, –2)	9	9	18		
		–1	(2, 0)	5	11	16	16	–1
8	(3, 1)	1	(2, 0)	3	11	14	14	1
		–1	(2, 2)	2	13	15		
9	(4, 0)	1	(3, –1)	6	16	22	22	1
		–1	(3, 1)	9	14	23		

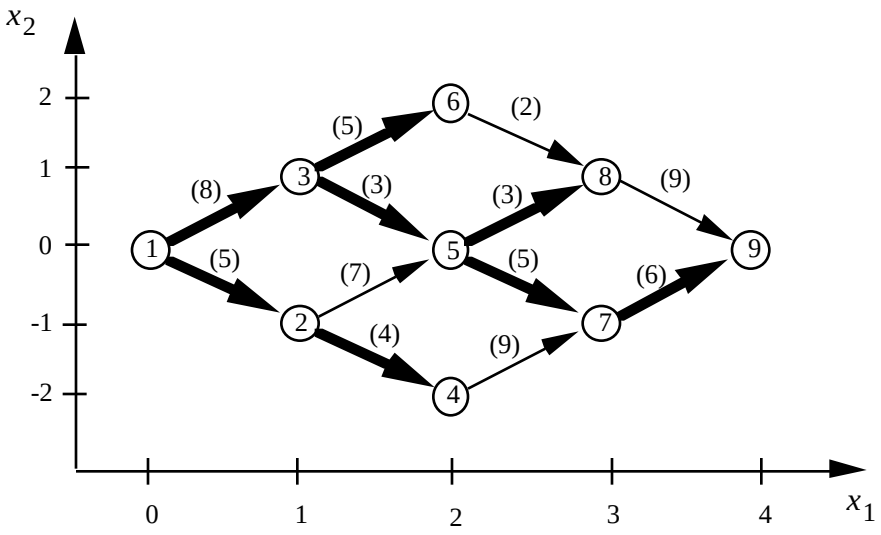


Figure 5. Optimal policy using forward recursion

The length of the optimal path is the last entry in column (8) of Table 7; in particular, $f_b(4,0) = 22$. The optimal solution is identified by using a *backward recovery* algorithm. The procedure starts at the final state and follows the optimal policy backward until an initial state is encountered. The computations are given in Table 8. The last column lists the cumulative path length, confirming once again that when we arrive at the initial state, $z(\mathbf{P}^*) = 22$.

Table 8. Backward recovery

$\mathbf{s}$	$\mathbf{d}^*(\mathbf{s})$	$\mathbf{s}_b$	$z(\mathbf{P}^*)$
(4, 0)	1	(3, –1)	—
(3, –1)	–1	(2, 0)	6
(2, 0)	–1	(1, 1)	11
(1, 1)	1	(0, 0)	14
(0, 0)	—	—	22

20.5 Reaching

Backward and forward recursion are known as *pulling* methods because the optimal decision policy $\mathbf{d}^*(\mathbf{s})$ tells us how to pull ourselves into a particular state from a predecessor state. Reaching is an alternative solution technique that combines the forward generation of states with forward recursion on an acyclic network. In effect, the optimal decision policy is derived while the state space is being generated, a concurrent operation. When all the states have been generated, the optimization is complete. The solution is found with the backward recovery procedure.

Reaching can be extremely efficient especially when paired with bounding techniques as discussed presently; however, it may not be appropriate for all problems. The principal requirement is the availability of both forward and backward transition functions and the forward recursive equation. At the end of the section, several situations are identified in which reaching outperforms both backward and forward recursion.

Development

The forward recursive equation (4) can be written as

$$f_b(\mathbf{s}) = \text{Minimize}\{z_b(\mathbf{s}_b, \mathbf{d}, \mathbf{s}) + f_b(\mathbf{s}_b) : \mathbf{d} \quad \boldsymbol{D}_b(\mathbf{s}), \mathbf{s}_b = T_b(\mathbf{s}, \mathbf{d})\} \qquad (5)$$

In this section we write the decision objective explicitly in terms of the previous state $\mathbf{s}_b$ and the current state $\mathbf{s}$. A few substitutions result in a more convenient form for the computational procedure outlined below. Note that

$$z_b(\mathbf{s}_b, \mathbf{d}, \mathbf{s}) = z(\mathbf{s}_b, \mathbf{d}, \mathbf{s})$$

$$\text{and that} \qquad \mathbf{s}_b = T_b(\mathbf{s}, \mathbf{d}), \quad \mathbf{s} = T(\mathbf{s}_b, \mathbf{d}).$$

Using these relationships, we write (5) alternatively as

$$f_b(\mathbf{s}) = \text{Minimize}\{z(\mathbf{s}_b, \mathbf{d}, \mathbf{s}) + f_b(\mathbf{s}_b) : \mathbf{d} \quad \boldsymbol{D}(\mathbf{s}_b), \mathbf{s} = T(\mathbf{s}_b, \mathbf{d})\} \qquad (6)$$

where the minimum is take over all states $\mathbf{s}_b$ and decisions $\mathbf{d}$ at $\mathbf{s}_b$ that lead to the given state $\mathbf{s}$.

The strategy behind reaching is to solve Eq. (6) while the states are being generated in the forward direction. Recall that the forward generation procedure starts with the set of initial states $\boldsymbol{I}$ and the boundary conditions $f_b(\mathbf{s})$ for all $\mathbf{s} \quad \boldsymbol{I}$. For any state $\mathbf{s} \quad \boldsymbol{S}$ and feasible decision $\mathbf{d} \quad \boldsymbol{D}(\mathbf{s})$, a new state $\mathbf{s}'$ is created by the transition function $T(\mathbf{s}, \mathbf{d})$. Re stating (6) in these terms first requires the replacement of $\mathbf{s}$ by $\mathbf{s}'$, and then the replacement of $\mathbf{s}_b$ by $\mathbf{s}$. This yields

$$f_b(\mathbf{s}') = \text{Minimize}\{z(\mathbf{s}, \mathbf{d}, \mathbf{s}') + f_b(\mathbf{s}) : \mathbf{d} \quad \boldsymbol{D}(\mathbf{s}), \mathbf{s}' = T(\mathbf{s}, \mathbf{d})\} \qquad (7)$$

Note that the minimum is taken over all $\mathbf{s}$ such that $\mathbf{s}'$ can be reached from $\mathbf{s}$ when decision $\mathbf{d} \quad \boldsymbol{D}(\mathbf{s})$ is taken.

When a new state $\mathbf{s}'$ is first generated the temporary assignment

$$\bar{f}_b(\mathbf{s}') = z(\mathbf{s}, \mathbf{d}, \mathbf{s}') + f_b(\mathbf{s})$$

is made. This value is an upper bound on the actual value of $f_b(\mathbf{s}')$ because it is only one of perhaps many possible ways of entering $\mathbf{s}'$. As the generation process continues a particular state $\mathbf{s}'$ may be reached through other combinations of $\mathbf{s}$ and $\mathbf{d}$. Whenever a state that already exists is generated again, the current path value is checked against the cost of the new path just found. If the new path has a lower value, $\bar{f}_b(\mathbf{s}')$ and $\mathbf{d}^*(\mathbf{s}')$ are replaced. That is, when

$$\bar{f}_b(\mathbf{s}') > z(\mathbf{s}, \mathbf{d}, \mathbf{s}') + f_b(\mathbf{s})$$

for some $\mathbf{s}' = T(\mathbf{s}, \mathbf{d})$, we replace $\bar{f}_b(\mathbf{s}')$ with $z(\mathbf{s}, \mathbf{d}, \mathbf{s}') + f_b(\mathbf{s})$ and $\mathbf{d}^*(\mathbf{s}')$ with $\mathbf{d}$.

Reaching Algorithm

The generation process starts at the initial states in $\boldsymbol{I}$ and proceeds lexicographically through the list of states already identified. For algorithmic purposes we will use the same symbol $\boldsymbol{S}$ used to denote the complete state space, to represent this list. When a particular state $\mathbf{s} \in \boldsymbol{S}$ is selected for forward generation, the true value of $f_b(\mathbf{s})$ must be the value of the optimal path function as determined by (7). This follows because all possible paths that enter $\mathbf{s}$ must come from a state that is lexicographically smaller than $\mathbf{s}$. When $\mathbf{s}$ is reached in the generation process, all immediate predecessors of $\mathbf{s}$ must have been considered in the determination of $\bar{f}_b(\mathbf{s}')$ so this must be the optimal value.

The algorithm below provides the details of the reaching procedure. As can be seen, it simultaneously generates and optimizes the state space.

Step 1. Start with a set of initial states $\boldsymbol{I}$ and store them as list in memory. Call the list $\boldsymbol{S}$. Initialize $f_b(\mathbf{s})$ for all $\mathbf{s} \in \boldsymbol{I}$. These values must be given or implied in the problem definition. Let $\mathbf{s} \in \boldsymbol{S}$ be the lexicographically smallest state available.

Step 2. Find the decision set $\boldsymbol{D}(\mathbf{s})$ associated with $\mathbf{s}$. Assume that there are l feasible decisions and let

$$\boldsymbol{D}(\mathbf{s}) = \{\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_l\}.$$

Set $k = 1$.

Step 3. Let the decision $\mathbf{d} = \mathbf{d}_k$ and find the successor state $\mathbf{s}'$ to $\mathbf{s}$ using the forward transition function:

$$\mathbf{s}' = T(\mathbf{s}, \mathbf{d}).$$

If $\mathbf{s}$ is not feasible go to Step 4; otherwise, search the list $\boldsymbol{S}$ to determine if $\mathbf{s}'$ has already been generated.

i. If not, put $\boldsymbol{S} \leftarrow \boldsymbol{S} \quad \{\mathbf{s}'\}$, $\mathbf{d}^*(\mathbf{s}') \leftarrow \mathbf{d}_k$ and compute the initial estimate of the optimal path value for state $\mathbf{s}'$.

$$\bar{f}_b(\mathbf{s}') = z(\mathbf{s}, \mathbf{d}, \mathbf{s}') + f_b(\mathbf{s})$$

ii. If $\mathbf{s}' \quad \boldsymbol{S}$ indicating that the state already exists, whenever

$$\bar{f}_b(\mathbf{s}') > z(\mathbf{s}, \mathbf{d}, \mathbf{s}') + f_b(\mathbf{s})$$

put $\bar{f}_b(\mathbf{s}') \leftarrow z(\mathbf{s}, \mathbf{d}, \mathbf{s}') + f_b(\mathbf{s})$ and $\mathbf{d}^*(\mathbf{s}') \leftarrow \mathbf{d}$ and go to Step 4.

Step 4. Put $k \leftarrow k + 1$. If $k > l$, put $f_b(\mathbf{s}) \leftarrow \bar{f}_b(\mathbf{s})$ and go to Step 5 (all feasible decisions have been considered at $\mathbf{s}$ so we can examine the next state; also the optimal path to $\mathbf{s}$ is known).

If $k \leq l$, go to Step 3.

Step 5. Find the next lexicographically larger state than $\mathbf{s}$ on the list $\boldsymbol{S}$. Rename this state $\mathbf{s}$ and go to Step 2. If no state can be found go to Step 6.

Step 6. Stop, the state space is complete and the optimal path has been found from all initial states in $\boldsymbol{I}$ to all states in $\boldsymbol{S}$. Let $\boldsymbol{F} \quad \boldsymbol{S}$ be the set of final states. Solve

$$f_b(\mathbf{s}_F) = \text{Minimize}\{f_b(\mathbf{s}) : \mathbf{s} \quad \boldsymbol{F}\}$$

to determine the optimal path value and the final state $\mathbf{s}_F$ on the optimal path. Perform backward recovery to determine the optimal sequence of states and decisions.

Example 3

Consider again the path problem in Fig. 3. Applying the reaching algorithm to this problem gives the results shown in Table 9. We have rearranged and renamed some of the columns to better reflect the order of the process.

Table 9. Reaching solution for rotated path problem

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
Index	$\mathbf{s}$	$f_b(\mathbf{s})$	$\mathbf{d}$	z	$\mathbf{s}'$	$r(\mathbf{s}, \mathbf{d})$	$\bar{f}_b(\mathbf{s}')$	$\mathbf{d}^*(\mathbf{s}')$
1	(0, 0)	0	–1	5	(1, –1)	5	5	–1
			+1	8	(1, 1)	8	8	+1
2	(1, –1)	5	–1	4	(2, –2)	9	9	–1
			+1	7	(2, 0)	12	12	+1
3	(1, 1)	8	–1	3	(2, 0)	11	Min(12, 11) = 11	–1
			+1	5	(2, 2)	13	13	+1
4	(2, –2)	9	+1	9	(3, –1)	18	18	+1
5	(2, 0)	11	–1	5	(3, –1)	16	Min(18, 16) = 16	–1
			+1	3	(3, 1)	14	14	+1
6	(2, 2)	13	–1	2	(3, 1)	15	Min(15, 14) = 14	+1
7	(3, –1)	16	+1	6	(4, 0)	22	22	+1
8	(3, 1)	14	–1	9	(4, 0)	23	Min(22, 23) = 22	+1
9	(4, 0)	22	---					

We begin the process at the initial state $\mathbf{s} = (0,0)$, with $f_b(0,0) = 0$. Two decisions are possible: $d = -1$ and $d = 1$. The former leads to $\mathbf{s}' = (1,-1)$ and the latter to $\mathbf{s}' = (1,1)$. Both of these states are added to the list $\boldsymbol{S}$ and temporary values of $\bar{f}_b(\mathbf{s}')$ are computed:

$$\bar{f}_b(1,-1) = a((0,0), -1) + f_b(0,0) = 5 + 0 = 5$$

$$\bar{f}_b(1, 1) = a((0,0), 1) + f_b(0,0) = 8 + 0 = 8$$

At iteration 2, we move to the next lexicographically larger state on $\boldsymbol{S}$ which is $\mathbf{s} = (1,-1)$. The value of $f_b(1,-1)$ is fixed to $\bar{f}_b(1,-1) = 5$ since the temporary value must be the solution of the forward recursive equation. The corresponding value of $\mathbf{d}^*(1,-1)$ is also fixed to its current value. Once again, we consider the two possible decisions $d = -1$ and $d = 1$ and use the forward transition function to generate states $\mathbf{s}' = (2,-2)$ and $\mathbf{s}' = (2, 0)$, respectively. The corresponding values of $\bar{f}_b(\mathbf{s}')$ are

$$\bar{f}_b(2,-2) = a((1,-1), -1) + f_b(1,-1) = 4 + 5 = 9$$

and

$$\bar{f}_b(2, 0) = a((1, -1), 1) + f_b(1, -1) = 7 + 5 = 12.$$

This completes iteration 2 so we can fix $\bar{f}_b(1,-1) = 5$ and $\mathbf{d}^*(1,-1) = -1$. The state list is $\boldsymbol{S} = \{(0,0), (1,-1), (1,1), (2,-2), (2,0)\}$ and we move to $\mathbf{s} = (1, 1)$, the next larger state as determined by the lexicographical order. Reaching out from (1, 1) with decisions $d = 1$ and $d = -1$, respectively, leads to one new state $\mathbf{s}' = (2, 2)$ and one previously encountered state $\mathbf{s}' = (2, 0)$. For $\mathbf{s}' = (2, 2)$ we have

$$\bar{f}_b(2, 2) = a((1, 1), 1) + f_b(1, 1) = 5 + 8 = 13$$

and for $\mathbf{s}' = (2, -2)$ we have

$$\bar{f}_b(2, 0) = a((1, 1), -1) + f_b(1, 1) = 3 + 8 = 11.$$

We show in the table that $\bar{f}_b(2, 0)$ is replaced by the minimum of its previous value 12 and the new value 11. The optimal decision is also updated. We continue in the same manner until the entire state space is explored. The process only generates reachable states.

At the completion of iteration 8, the only remaining state is (4, 0) which is in $\boldsymbol{F}$. One more iteration is required to verify that the termination condition is met; that is, there are no more states to explore. We now go to Step 6 and would ordinarily solve the indicated optimization problem to determine the optimal path value and the final state $\mathbf{s}_F$ on the optimal path. This is not necessary, though, because $\boldsymbol{F}$ is single-valued, implying that $\mathbf{s}_F = (4, 0)$. The optimal path is identified by backward recovery with the help of $\mathbf{d}^*(\mathbf{s})$, the optimal decision leading into each state $\mathbf{s}$. For the example, backward recovery begins with the unique terminal state (4, 0). The optimal path is the same as the one identified by the forward recursion procedure given in Table 8.

Reaching vs. Pulling

For the types dynamic programming models that we have addressed, it is natural to ask which algorithmic approach will provide the best results. To answer this question, consider again a finite acyclic network whose node set consists of the integers 1 through n and whose arc set consist of all pairs (i, j), where i and j are integers having $1 \le i < j \le n$. As before, arcs point to higher-numbered nodes. Let arc (i, j) have length a_{ij}, where $-\infty \le a_{ij} \le \infty$. If (i, j) cannot be traversed it is either omitted from the network or assigned the length $-\infty$ or $+\infty$, depending on the direction of optimization.

For a minimization problem the goal is to find the shortest path from node 1 to node j, for $j = 2, \ldots, n$, while for a maximization problem the goal is to find the longest path to each node j. To determine under what circumstances the reaching method provides an advantage over the pulling methods in Sections 20.3 and 20.4 for a minimization objective, let f_j be the

length of the shortest path from node 1 to some node j. By the numbering convention, this path has some final arc (i, j) such that $i < j$. Hence, with $f_1 = 0$,

$$f_j = \min\{f_i + a_{ij} : i < j\}, \quad j = 2, \ldots, n \tag{8}$$

Forward recursion computes the right-hand side of Eq. (8) in ascending j, and eventually finds the shortest path from node 1 to every other node. The process is described in the following algorithm.

Pulling method (*forward recursion*):

1. Set $v_1 = 0$ and $v_j = \infty$ for $j = 2, \ldots, n$.
2. DO for $j = 2, \ldots, n$.
    3. DO for $i = 1, \ldots, j - 1$.

$$v_j \leftarrow \min\{v_j, v_i + a_{ij}\} \tag{9}$$

To be precise, the expression $x \leftarrow y$ means that x is to assume the value now taken by y. Thus (9) replaces the label v_j by $v_i + a_{ij}$ whenever the latter is smaller. A "DO" statement means that the subsequent instructions in the algorithm are to be executed for the specific sequence of index values. Step 2 says that Step 3 is to be executed $n - 1$ times, first with $j = 2$, then with $j = 3$, and so on. At the completion of Step 3 for each j, we have $f_j = v_j$.

Alternatively, we can solve Eq. (8) by reaching out from a node i to update the labels v_j for all $j > i$. Again, the algorithm finds the shortest path from node 1 to every other node as follows.

Reaching method:

1. Set $v_1 = 0$ and $v_j = \infty$ for $j = 2, \ldots, n$.
2. DO for $i = 1, \ldots, n - 1$.
    3. DO for $j = i + 1, \ldots, n$.

$$v_j \leftarrow \min\{v_j, v_i + a_{ij}\} \tag{10}$$

Expressions (9) and (10) are identical, but the DO loops are interchanged. Reaching executes (10) in ascending i and, for each i, in ascending j. One can show by induction that the effect of i executions of the nested DO loops is to set each label v_k equal to the length of the shortest path from node 1 to node k whose final arc (l, k) has $l \leq i$. As a consequence, reaching stops with $v_k = f_k$ for all k.

The above pulling and reaching algorithms require work (number of computational operations) proportional to n^2. For sparse or structured networks, faster procedures are available (see Denardo 1982).

When Reaching Runs Faster

Given that (9) and (10) are identical, and that each is executed exactly once per arc in the network, we can conclude that reaching and pulling entail the same calculations. Moreover, when (9) or (10) is executed for arc (i, j), label v_i is known to equal f_i but f_j is not yet known.

The key difference between the two methods is that i varies on the outer loop of reaching, but on the inner loop of pulling. We now describe a simple (and hypothetical) situation in which reaching may run faster. Suppose it is known prior to the start of the calculations that the f-values will be properly computed even if we don't execute (9) or (10) for all arcs (i, j) having $f_i > 10$. To exclude these arcs when doing reaching, it is necessary to add a test to Step 2 that determines whether or not v_i exceeds 10. If so, Step 3 is omitted. This test is performed on the *outer* loop; it gets executed roughly n times. To exclude these arcs when doing pulling, we must add the same test to Step 3 of the algorithm. If v_i is greater than 10, (9) is omitted. This test is necessarily performed on the *inner* loop which gets executed roughly $n^2/2$ times. (The exact numbers of these tests are $(n - 1)$ and $(n - 1)n/2$, respectively; the latter is larger for all $n > 2$.)

In the following discussion from Denardo, we present four optimization problems whose structure accords reaching an advantage. In each case, this advantage stems from the fact that the known f-value is associated with a node on the outer loop of the algorithm. Consequently, the savings is $O(n^2/2)$ rather than $O(n)$, as it would be if the known f-value were associated with a node on the inner loop.

First, suppose it turns out that no finite-length path exists from node 1 to certain other nodes. One has $v_j = f_j = \infty$ for these nodes. It is not necessary to execute (9) or (10) for any arc (i, j) having $v_j = \infty$. If reaching is used, fewer tests are needed because i varies on the outer loop.

Second, consider a shortest-path problem in which one is solely interested in f_k for a particular k, and suppose that all arc lengths are nonnegative. In this case, it is not necessary to execute (9) or (10) for any arc (i, j) having $v_i \geq v_k$. Checking for this is quicker with reaching because i varies on the outer loop.

Third, consider a shortest-route problem in a directed acyclic network where the initial node set $\boldsymbol{I} = \{1\}$, but where the nodes are *not* numbered so that each arc (i, j) satisfies $i < j$. To solve the recursion, one could relabel the nodes and then use the pulling method. The number of computer operations (arithmetic operations, comparisons, and memory accesses) needed to relabel the nodes, however, is roughly the same as the number of computer operations needed to determine the shortest path tree. A saving can be obtained by combining relabeling with reaching as follows. For each node i, initialize label $b(i)$ by equating it to the number of arcs that terminate at node i. Maintain a list L consisting of those nodes i having $b(i) = 0$. Node 1 is the only beginning node, so $L = \{1\}$ initially. Remove any node i from L. Then, for each arc (i, j) emanating from node i, execute (10),

subtract 1 from $b(j)$, and then put j onto L if $b(j) = 0$. Repeat this procedure until L is empty.

Fourth, consider a model in which node i describes the state of the system at time t_i. Suppose that each arc (r, s) reflects a control (proportional to cost) c_{rs} which is in effect from time t_r to time t_s, with $t_r < t_s$. Suppose it is known a priori that a monotone control is optimal (e.g., an optimal path $(1,\ldots,h, i, j,\ldots)$ has $c_{hi} \le c_{ij}$). This would occur, for instance, in an inventory model where the optimal stock level is a monotone function of time. The optimal path from node 1 to an intermediary node i has some final arc (h, i). When reaching is used, it is only necessary to execute (10) for those arcs (i, j) having $c_{hi} \le c_{ij}$. This allows us to *prune* the network during the computations on the basis of structural information associated with the problem. The test is accomplished more quickly with reaching because i varies on the outer loop.

In the next section, we generalize the hypothetical above situation and show how reaching can be combined with a bounding procedure to reduce the size of the network. The main disadvantage of reaching is that it can entail more memory accesses than traditional recursion during the computations. A memory access refers to the transfer of data between the storage registers of a computer and its main memory.

Elimination By Bounds

Our ability to solve dynamic programs is predominantly limited by the size of the state space and the accompanying need to store excessive amounts of information. This contrasts sharply with our ability to solve other types of mathematical programs where the computational effort is the limiting factor. One way to reduce the size of the state space is to identify, preferably as early as possible in the calculations, those states that cannot possibly lie on an optimal path. By eliminating those states, we also eliminate the paths (and hence the states on those paths) emanating from them. We now describe a fathoming test that can be performed at each iteration of a DP algorithm. The reaching procedure is the best choice for implementation. If the test is successful, the state is fathomed and a corresponding reduction in the state space is realized.

To begin, consider a general DP whose state space can be represented by an acyclic network where the objective is to determine the minimum cost path from any initial state to any final state. Let $\boldsymbol{P}$ be the collection of all feasible paths and let $\mathbf{P}^*$ be the optimal path. The problem can be stated as follows.

$$z(\mathbf{P}^*) = \text{Minimize}_{\mathbf{P} \quad \boldsymbol{P}}\, z(\mathbf{P})$$

Now consider an arbitrary state $\mathbf{s}$ $\boldsymbol{S}$ and a feasible path $\mathbf{P_s}$ that passes through $\mathbf{s}$. Divide the path into two subpaths: $\mathbf{P}_\mathbf{s}^1$ which goes from an initial state to $\mathbf{s}$, and $\mathbf{P}_\mathbf{s}^2$ which goes from $\mathbf{s}$ to a final state. The objective for path $\mathbf{P_s}$ is also divided into two components $z(\mathbf{P}_\mathbf{s}^1)$ and $z(\mathbf{P}_\mathbf{s}^2)$, where

$$z(\mathbf{P}_S) = z(\mathbf{P}_{\mathbf{s}}^1) + z(\mathbf{P}_{\mathbf{s}}^2).$$

The reaching method starts from an initial state and derives for each state, in increasing lexicographic order, the value $f_b(\mathbf{s})$. This is the objective for the optimal path from an initial state to $\mathbf{s}$. Thus, if $\boldsymbol{P}^1$ is the set of all feasible subpaths from any state in $\boldsymbol{I}$ to $\mathbf{s}$

$$f_b(\mathbf{s}) = \text{Minimize}_{\mathbf{P}_{\mathbf{s}}^1 \quad \boldsymbol{P}^1} z(\mathbf{P}_{\mathbf{s}}^1).$$

We can also define the quantity $f(\mathbf{s})$ as the objective for the optimal path from $\mathbf{s}$ to a final state. Letting $\boldsymbol{P}^2$ be the set of all subpaths from $\mathbf{s}$ to a state in $\boldsymbol{F}$, we have

$$f(\mathbf{s}) = \text{Minimize}_{\mathbf{P}_{\mathbf{s}}^2 \quad \boldsymbol{P}^2} z(\mathbf{P}_{\mathbf{s}}^2).$$

Recall that $f(\mathbf{s})$ is the value obtained by backward recursion; however, in the reaching procedure this value is not available. Rather we assume the availability of some computationally inexpensive procedure for obtaining a lower bound, $z_{LB}(\mathbf{s})$, on $f(\mathbf{s})$; that is,

$$z_{LB}(\mathbf{s}) \le f(\mathbf{s}).$$

We also assume that it is possible to find a feasible path $\mathbf{P}_B$ from some state in $\boldsymbol{I}$ to some state in $\boldsymbol{F}$ with relatively little computational effort. It is not necessary for this path to pass through the state $\mathbf{s}$ considered above. Let $z_B = z(\mathbf{P}_B)$ be the objective value for the path. The subscript "B" stands for "best" to indicate the "best solution found so far"; that is, the *incumbent.*

The bounding test for state $\mathbf{s}$ compares the sum of $f_b(\mathbf{s})$ and $z_{LB}(\mathbf{s})$ to z_B. If

$$z_B \le f_b(\mathbf{s}) + z_{LB}(\mathbf{s})$$

then any path through $\mathbf{s}$ can be no better than the solution already available, $\mathbf{P}_B$, and $\mathbf{s}$ can be fathomed. If the above inequality does not hold, $\mathbf{s}$ may be part of a path that is better than $\mathbf{P}_B$ and cannot be fathomed. This is called the *bounding elimination test* because the lower bound $z_{LB}(\mathbf{s})$ is used in the test, and z_B represents an upper bound on the optimum. These ideas are illustrated in Fig. 6 which shows two paths from $\boldsymbol{I}$ to $\boldsymbol{F}$.

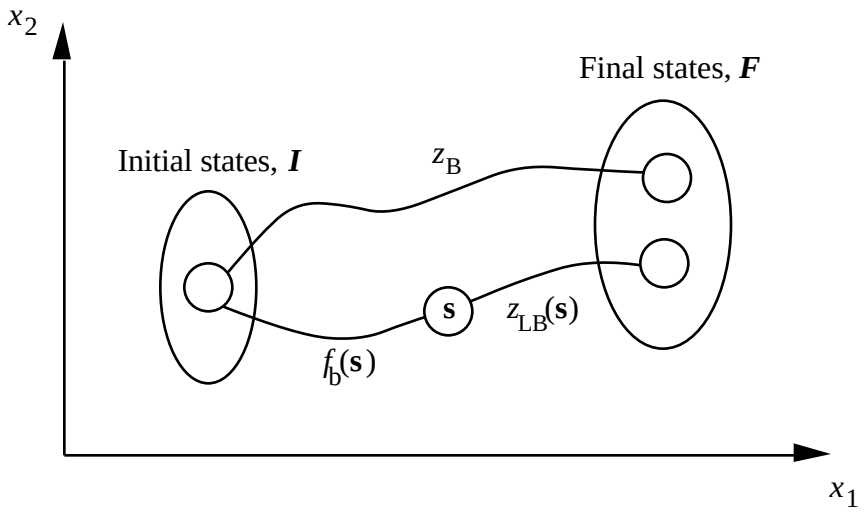


Figure 6. Components of bounding elimination test

The effectiveness of the test depends on the quality of the bounds z_B and $z_{LB}(\mathbf{s})$ and the ease with which they can be computed. The value z_B should be as small as possible, and the value of $z_{LB}(\mathbf{s})$ should be as large as possible. There is always a tradeoff between the quality of the bounds and the computational effort involved in obtaining them.

To implement the bounding procedures, two new user supplied subroutines are required. We will call them RELAX and FEASIBLE, and note that they are problem dependent. RELAX provides the lower bound $z_{LB}(\mathbf{s})$, and usually involves solving a relaxed version of the original problem. The solution to a minimization problem with relaxed constraints, for example, will always yield the desired result.

The purpose of FEASIBLE is to determine a feasible path from **s** to a state in ***F***. Call this path $\mathbf{P}_\mathbf{s}^2$. The fact that $\mathbf{P}_\mathbf{s}^2$ is feasible but not necessarily optimal means that the objective $z(\mathbf{P}_\mathbf{s}^2)$ is an upper bound on $f(\mathbf{s})$. The combination of the optimal subpath to **s** which has the objective $f_b(\mathbf{s})$ and the subpath $\mathbf{P}_\mathbf{s}^2$ provides us with a feasible path through the state space with objective value

$$f_b(\mathbf{s}) + z(\mathbf{P}_\mathbf{s}^2).$$

This is an upper bound on the optimal solution $z(\mathbf{P}^*)$ and is a candidate for the best solution z_B. In particular, if

$$f_b(\mathbf{s}) + z(\mathbf{P}_\mathbf{s}^2) < z_B$$

then let $\quad z_B = f_b(\mathbf{s}) + z(\mathbf{P}_\mathbf{s}^2)$

and replace the incumbent $\mathbf{P}_B$ with the concatenation of $\mathbf{P}_\mathbf{s}^2$ and the optimal path that yields $f_b(\mathbf{s})$.

Any heuristic that provides a good feasible solution can be used to obtain the path $\mathbf{P}_\mathbf{s}^2$. A common implementation of FEASIBLE often involves rounding the solution obtained by RELAX. This is illustrated in Example 4 below.

Note that for state $\mathbf{s}$ if

$$z(\mathbf{P}_\mathbf{s}^2) = z_{\mathrm{LB}}(\mathbf{s})$$

then $\mathbf{P}_\mathbf{s}^2$ must be an optimal path from $\mathbf{s}$ to a state in $\boldsymbol{F}$. The concatenation of the optimal path to $\mathbf{s}$ and the subpath $\mathbf{P}_\mathbf{s}^2$ yields an optimal path through $\mathbf{s}$. In this case, $\mathbf{s}$ can also be fathomed because further exploration of the state space from state $\mathbf{s}$ cannot yield a better solution.

The bounding elimination test is an addition to the reaching procedure. Recall that this procedure considers each state in increasing lexicographic order. The tests are performed immediately after a state $\mathbf{s}$ is selected for the reaching operation but before any decisions from that state have been considered. The idea is to eliminate the state, if possible, before any new states are generated from it. The reaching algorithm is repeated below for a minimization objective with the bounding modifications added.

Reaching Algorithm with Elimination Test

Step 1. Start with a set of initial states $\boldsymbol{I}$ and store them as list in memory. Call the list $\boldsymbol{S}$. Initialize $f_{\mathrm{b}}(\mathbf{s})$ for all $\mathbf{s} \in \boldsymbol{I}$. These values must be given in the problem definition. Let $\mathbf{s} \in \boldsymbol{S}$ be the lexicographically smallest state available. Set $z_{\mathrm{B}} = \infty$.

Step 2. Compute a lower bound $z_{\mathrm{LB}}(\mathbf{s})$ on the optimal subpath from $\mathbf{s}$ to a state in $\boldsymbol{F}$.

i. If $f_{\mathrm{b}}(\mathbf{s}) + z_{\mathrm{LB}}(\mathbf{s}) \geq z_{\mathrm{B}}$ then fathom state $\mathbf{s}$ and go to Step 6. Otherwise, try to find a feasible path from $\mathbf{s}$ to a final state in $\boldsymbol{F}$. Call it $\mathbf{P}_\mathbf{s}^2$ and compute the objective value $z(\mathbf{P}_\mathbf{s}^2)$. If a feasible path cannot be found go to Step 3.

ii. If $f_{\mathrm{b}}(\mathbf{s}) + z(\mathbf{P}_\mathbf{s}^2) \geq z_{\mathrm{B}}$ go to Step 3.

iii. If $f_{\mathrm{b}}(\mathbf{s}) + z(\mathbf{P}_\mathbf{s}^2) < z_{\mathrm{B}}$ then put $z_{\mathrm{B}} \leftarrow f_{\mathrm{b}}(\mathbf{s}) + z(\mathbf{P}_\mathbf{s}^2)$ and let $\mathbf{P}_{\mathrm{B}}$ be the concatenation of $\mathbf{P}_\mathbf{s}^2$ and the optimal path to $\mathbf{s}$.

iv. If $z(\mathbf{P}_\mathbf{s}^2) = z_{\mathrm{LB}}(\mathbf{s})$ then fathom $\mathbf{s}$ and go to Step 6; otherwise, go to Step 3.

Step 3. Find the set of decisions $\boldsymbol{D}(\mathbf{s})$ associated with $\mathbf{s}$. Assume that there are l feasible decisions and let

$$\boldsymbol{D}(\mathbf{s}) = \{\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_l\}.$$

Set $k = 1$ and go to Step 4.

Step 4. Let the current decision be $\mathbf{d} = \mathbf{d}_k$ and find the successor state $\mathbf{s}'$ to $\mathbf{s}$ using the forward transition function:

$$\mathbf{s}' = T(\mathbf{s}, \mathbf{d}).$$

If $\mathbf{s}'$ is not feasible go to Step 5; otherwise, search the list $\boldsymbol{S}$ to determine if $\mathbf{s}'$ has already been generated.

i. If not, put $\boldsymbol{S} \leftarrow \boldsymbol{S} \quad \{\mathbf{s}'\}$, $\mathbf{d}^*(\mathbf{s}') \leftarrow \mathbf{d}_k$ and compute the initial estimate of the optimal path value for state $\mathbf{s}'$.

$$\bar{f}_b(\mathbf{s}') = f_b(\mathbf{s}) + z(\mathbf{s}, \mathbf{d}, \mathbf{s}')$$

ii. If $\mathbf{s}' \quad \boldsymbol{S}$ indicating that the state already exists, whenever

$$\bar{f}_b(\mathbf{s}') > f_b(\mathbf{s}) + z(\mathbf{s}, \mathbf{d})$$

put $\bar{f}_b(\mathbf{s}') \leftarrow f_b(\mathbf{s}) + z(\mathbf{s}, \mathbf{d})$, $\mathbf{d}^*(\mathbf{s}') \leftarrow \mathbf{d}$ and go to Step 5.

Step 5. Put $k \leftarrow k + 1$. If $k > l$, put $f_b(\mathbf{s}) \leftarrow \bar{f}_b(\mathbf{s})$ and go to Step 6 (all feasible decisions have been considered at $\mathbf{s}$ so we can examine the next state; also the optimal path to $\mathbf{s}$ is known). If $k \le l$, go to Step 4.

Step 6. Find the next lexicographically larger state than $\mathbf{s}$ on the list $\boldsymbol{S} \setminus \boldsymbol{F}$. Rename this state $\mathbf{s}$ and go to Step 2. If no state can be found go to Step 7.

Step 7. Stop, the state generation process is complete and the optimal path has been found from all initial states in $\boldsymbol{I}$ to all states in $\boldsymbol{S}$ (note that $\boldsymbol{S}$ will not in general contain all feasible states because some will have been fathomed). The optimal path is $\mathbf{P}_B$. Perform backward recovery to determine the optimal sequence of states and decisions.

At Step 6, a test is needed to determine if the next lexicographically larger state is in $\boldsymbol{F}$. It is always possible to test a state $\mathbf{s}'$ when it is generated at Step 4 but this would be inefficient because some states are generated many times. When we arrive at Step 7, the optimal path is the incumbent $\mathbf{P}_B$. Therefore, it is not necessary to examine all $\mathbf{s} \quad \boldsymbol{F}$ and pick the one associated with the smallest value of $f_b(\mathbf{s})$ as was the case for the reaching algorithm without the bounding subroutines.

Lower Bounding Procedure - RELAX

There are two conflicting goals to consider when selecting a procedure to obtain the lower bound $z_{LB}(\mathbf{s})$. The first is that it should be computational inexpensive because it will be applied to every state. The second is that it should provide as large a value as possible. The larger the lower bound, the more likely the current state can be fathomed.

There are also at least two ways to find a lower bound for a particular problem. First, some constraints on the problem can be relaxed so that the set of feasible solutions is larger. Solving the relaxed problem

will yield an optimal solution with a value that can be no greater than the optimum for the original problem. Second, the objective function can be redesigned so that it lies entirely below the objective of the original problem for every feasible solution. Then, the optimal solution of the new problem will have a smaller value than the original.

Employing either of these two approaches, independently or in combination, results in a relaxed problem whose solution is obtained with what we called the RELAX procedure. There are many ways to form a relaxed problem; however, ease of solution and bound quality are the primary criteria used to guide the selection.

Knapsack Problem

We have already seen that a relaxation of the one constraint knapsack problem can be obtained by replacing the integrality requirement $x_j = 0$ or 1 by the requirement $0 \le x_j \le 1$ for $j = 1, \ldots, n$. This is a relaxation because all the noninteger values of the variables between 0 and 1 have been added to the feasible region.

A multiple constraint binary knapsack problem is shown below.

$$\begin{aligned} \text{Maximize} \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} x_j \le b_i \qquad i = 1, \ldots, m \\ & x_j = 0 \text{ or } 1 \qquad j = 1, \ldots, n \end{aligned} \tag{11}$$

Again, an obvious relaxation is to replace the 0-1 variables with continuous variables bounded below by zero and above by one. The resulting problem is a linear program. The simplex method, say, would be the RELAX procedure.

A simpler relaxation can be obtained by multiplying each constraint by an arbitrary positive constant λ_i ($i = 1, \ldots, m$) and then summing. This leads to what is called a *surrogate* constraint.

$$\sum_{j=1}^{n} \sum_{i=1}^{m} \lambda_i a_{ij} x_j \le \sum_{i=1}^{m} \lambda_i b_i \tag{12}$$

Replacing (11) with (12) results in a single constraint knapsack problem. Although the single and multiple knapsack problems require the same amount of computational effort to solve in theory, in practice the former is much easier. In any case, solving the relaxation as a single constraint dynamic program or a single constraint LP will give a lower bound $z_{\mathrm{LB}}(\mathbf{s})$ at some state $\mathbf{s}$.

Job Sequencing Problem

A relaxation of the sequencing problem with due dates discussed in Section 19.5 can be obtained by redefining the objective function. Consider the nonsmooth cost function $c(j, t)$ for job j as shown in Fig. 7. The cost is zero if the job is finished before the due date $d(j)$ but rises linearly at a rate $a(j)$ as the due date is exceeded. A linear function that provides a lower bound to $c(j, t)$ is shown in Fig. 8. This function intersects the cost axis at the point $-d(j)a(j)$ and rises at a rate $a(j)$. For completion times $t(j)$ less than $d(j)$ the function has negative values indicating a benefit, while it is the same for times above $d(j)$.

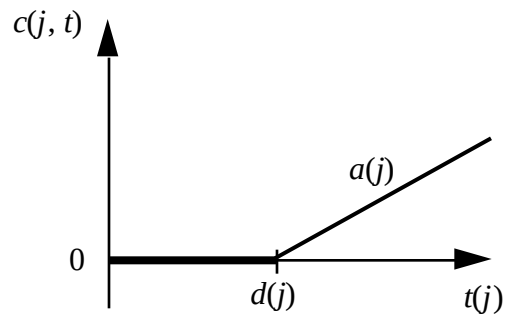


Figure 7. Cost function for sequencing problem

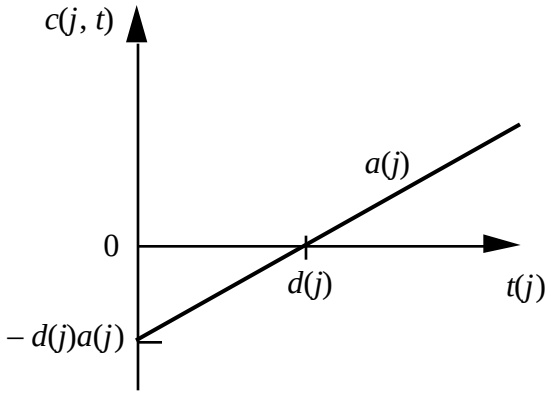


Figure 8. Lower bound cost function

The problem with the new cost function is much easier to solve than the original. To see this, let $p(j)$ be the time to perform job j and denote by j_k the kth job in a sequence. The objective function can now be written as

$$z = \sum_{k=1}^{n} -d(j_k)a(j_k) + \sum_{k=1}^{n} a(j_k)t(j_k)$$

for some sequence $(j_1, j_2, \ldots, j_n)$, where $t(j_k)$ is the completion time of job j_k.

The first term in the expression for z does not depend on the sequence chosen, and is constant. Therefore, it can be dropped in the optimization process. The remaining term is the objective function for the linear sequencing problem. The optimal solution of this problem can be found by ranking the jobs in the order of decreasing ratio $a(j)/p(j)$ and scheduling the jobs accordingly. Adding back the constant term provides the desired lower bound on the optimum.

Traveling Salesman Problem

Recall that the traveling salesman problem (TSP) is described by a point to point travel cost matrix, as illustrated for a six city example in Table 10 (see Section 8.6). The salesman must start at a given home base, traverse a series of arcs that visits all other cities exactly once, and then return to the home base. The optimal tour for this example is (1,4) → (4,3) → (3,5) → (5,6) → (6,2) → (2,1)} with objective value $z_{TSP} = 63$. The highlighted cells in Table 10 are the arcs in this solution.

Table 10. Cost matrix for city pairs

	1	2	3	4	5	6
1	—	27	43	16	30	26
2	7	—	16	1	30	25
3	20	13	—	35	5	0
4	21	16	25	—	18	18
5	12	46	27	48	—	5
6	23	5	5	9	5	—

A characteristic of a tour is that one and only cell must appear in every row and column of the cost matrix. This suggests that the classical assignment problem (AP), whose solutions have the same characteristic, can be used as a relaxation for the TSP. For the AP the goal is to assign each row to one and only one of the columns so that the total cost of the assignment is minimized. The solution to the assignment problem for the matrix in Table 10 is {(1,4), (2,1), (3,5), (4,2), (5,6), (6,3)} with objective value $z_{AP} = 54$. The corresponding cells are highlighted in Table 11.

Relating this solution to the TSP we see that two separate routes or subtours can be identified: (1,4) → (4,2) → (2,1) and (3,5) → (5,6) → (6,3). But TSP solutions are not allowed to have subtours so the AP solution is not feasible to the TSP. The restriction that the salesman follow a single tour has been dropped. The AP is thus a relaxation of the TSP, and because it is much easier to solve, it is a good candidate for the RELAX procedure of dynamic programming.

Table 11. Assignment problem solution

	1	2	3	4	5	6
1	—	27	43	16	30	26
2	7	—	16	1	30	25
3	20	13	—	35	5	0
4	21	16	25	—	18	18
5	12	46	27	48	—	5
6	23	5	5	9	5	—

An even simpler bounding procedure can be devised by considering a relaxation of the assignment problem where we select the minimum cost cell in each row. The corresponding solution may have more than one cell

chosen from a column so, in effective, we have dropped the requirement that one and only cell from each column be selected. For the cost matrix in Table 10, one of several solutions that yields the same objective value is {(1,4), (2,4), (3,6), (4,5), (5,6), (6,3)} with $z_{RAP} = 45$. This solution was found with almost no effort by scanning each row for the minimum cost cell. The general relationship between the three objective function values is $z_{RAP} \le z_{AP} \le z_{TSP}$ which was verified by the results: $45 < 54 < 63$.

This example shows that there may be many relaxations available for a given problem. There is usually a tradeoff between the effort required to obtain a solution and the quality of the bound. By going from the assignment problem to the relaxed assignment problem, for example, the bound deteriorated by 16.7%. Because very large assignment problems can be solved quite quickly, the reduced computational effort associated with solving the relaxed assignment would not seem to offset the deterioration in the bound, at least in this instance. Nevertheless, the selection of the best bounding procedure can only be determined by empirical testing.

Example 4

Consider the knapsack problem below written with a minimization objective to be consistent with the foregoing discussion.

$$\text{Minimize } z = -8x_1 - 3x_2 - 7x_3 - 5x_4 - 6x_5 - 9x_6 - 5x_7 - 3x_8$$

$$\text{subject to } x_1 + x_2 + 4x_3 + 4x_4 + 5x_5 + 9x_6 + 8x_7 + 8x_8 \le 20$$

$$x_j = 0 \text{ or } 1,\ j = 1,\ldots,8$$

The variables are ordered such that the "bang/buck" ratio is decreasing, where

$$\text{bang/buck} = \left[\frac{\text{objective coefficient}}{\text{constraint coefficient}}\right] = \left[\frac{c_j}{a_j}\right].$$

This term derives from the maximization form of the problem where the "bang" is the benefit from a project and the "buck" is its cost. Thus the bang per buck ratio is the benefit per unit cost. If the goal is to maximize benefits subject to a budget constraint on total cost, a greedy approach is to choose the projects (set the corresponding variables to 1) with the greatest bang per buck ratio. In our case, such a heuristic first sets x_1 to 1 and then in order $x_2 = 1$, $x_3 = 1$, $x_4 = 1$ and $x_5 = 1$. At this point the cumulative benefit is 29 and the cumulative cost (resource usage) is 15. An attempt to set x_6 to 1 violates the constraint so we stop with the solution $\mathbf{x}$ = (1, 1, 1, 1, 1, 0, 0, 0) which is not optimal.

Although this greedy heuristic rarely yields the optimum, it is extremely easy to execute and will form the basis of our FEASIBLE subroutine for the example. For the RELAX subroutine we solve the original problem without the integrality requirements on x_j. That is, we

substitute $0 \le x_j \le 1$ ($j = 1,\ldots,8$) for the 0-1 constraint. What results is a single constraint bounded variable linear program whose solution also depends on the bang per buck ratios. The following greedy algorithm yields the LP optimum.

Step 1. Set $z_0 = 0$, $b_0 = 0$ and $j = 1$.

Step 2. If $j > n$, then stop with $z_{LB} = z_{j-1}$.

If $b_{j-1} + a_j \le b$, then

set $x_j = 1$,

$z_j = z_{j-1} + c_j$,

$b_j = b_{j-1} + a_j$,

and go to Step 3.

If $b_{j-1} + a_j > b$, then

set $x_j = (b - b_{j-1})/a_j$

$z_j = z_{j-1} + c_j x_j$,

and stop with $z_{LB} = z_j$.

Step 3. Put $j \leftarrow j + 1$ and go to Step 2.

Note that z_{LB} is a lower bound on the optimum of the original problem because the integrality requirement on the variables has been relaxed. The solution will not be feasible, though, when one of the variables is fractional. If all variables turn out to be 0 or 1, the solution is both feasible and optimal to the original problem.

Example 5

We consider again the 15 variable binary knapsack problem discussed in Section 19.3 but with the items renumbered. In Table 12, the items are arranged in decreasing order of the benefit/weight ratio to simplify the determination of bounds. The weight limit is 30 and the objective is to maximize total benefit.

Table 12. Data for binary knapsack problem

Item	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Benefit	32	37.2	33	29.1	34.6	22.4	18.6	23.6	19.8	25	11.5	12.8	8	9.5	14.1
Weight	16	19	17	15	18	12	10	13	11	14	7	8	5	6	9
Benefit/ Weight	2.00	1.96	1.94	1.94	1.92	1.87	1.86	1.82	1.80	1.79	1.64	1.60	1.60	1.58	1.57

The problem was solved three different ways to provide a comparison of methods: (i) backward recursion with an exhaustively generated state space, (ii) reaching without bounds, and (iii) reaching with bounds. The bounds procedures were similar to those described in the previous example. To allow for multiple optimal solutions, we did not eliminate states whose upper and lower bounds were equal.

The results of the computations are shown in Table 13. The reduced number of states obtained with reaching without bounds is due to the fact that states not reachable from the initial state are not generated. When the bounds procedures are added, a further reduction is achieved due to fathoming.

Table 13. Comparison of computational procedure

Method	Number of states
Exhaustive generation/backward recursion	466
Reaching without bounds	230
Reaching with bounds	68

20.6 Stochastic Dynamic Programming

Up until now it has been assumed that a decision taken from a given state leads to a known result represented by a subsequent state in the network. In many situations, this is not the case. Before the decision is made, outcomes may not be known with certainty although it may be possible to describe them in probabilistic terms. For example, travel time on a network arc may vary with the traffic, maintenance costs for a machine may be depend on usage, or investment returns may be tied to the general health of the economy. In each such instance, the most that we may able to say about a particular outcome is that it occurs with a known probability, thus permitting us to specify a probability distribution over the set of transitions at a given state. For the decision maker, this can be the most challenging and frustrating aspect of the problem. Fortunately, the dynamic programming procedures previously described can be modified in a straightforward manner to handle this type of uncertainty giving rise to the field of stochastic dynamic programming.

Development

We continue to work with a finite, acyclic state space. Given a state $\mathbf{s}$ and a decision $\mathbf{d} \in \boldsymbol{D}(\mathbf{s})$, we will assume that the transition to a new state $\mathbf{s}'$ is not known a priori but is governed by a specified probability distribution. Let $P(\mathbf{s}':\mathbf{s},\mathbf{d})$ be the mass density function that describes the probability of a transition to the state $\mathbf{s}'$ given the decision $\mathbf{d}$ is made at state $\mathbf{s}$. For all possible outcomes $\mathbf{s}'$, it is required that

$$\sum_{\mathbf{s}' \in \boldsymbol{S}} P(\mathbf{s}':\mathbf{s},\mathbf{d}) = 1$$

and $\quad P(\mathbf{s}':\mathbf{s},\mathbf{d}) = 0$ if $\mathbf{s}' \notin \boldsymbol{S}$.

We redefine several quantities formerly used for the deterministic case.

$z(\mathbf{s}, \mathbf{d}, \mathbf{s}')$ = cost of starting in state $\mathbf{s}$, making decision $\mathbf{d}$ and moving to state $\mathbf{s}'$

$f(\mathbf{s})$ = minimum *expected* cost of arriving at a final state given the path starts in state $\mathbf{s}$

$\mathbf{d}^*(\mathbf{s})$ = decision for state $\mathbf{s}$ that results in the minimum expected cost

Given these definitions we obtain the expected path objective for state $\mathbf{s}$ and decision $\mathbf{d}$.

$$r(\mathbf{s},\mathbf{d}) = \sum_{\mathbf{s}' \in \boldsymbol{S}} P(\mathbf{s}':\mathbf{s},\mathbf{d})(z(\mathbf{s},\mathbf{d},\mathbf{s}') + f(\mathbf{s}')) \qquad (13)$$

The recursive equation that defines the function $f(\mathbf{s})$ is as follows.

$$f(\mathbf{s}) = \text{Minimize}\{r(\mathbf{s},\mathbf{d}) : \mathbf{d} \in \boldsymbol{D}(\mathbf{s})\}$$

This equation can be solved using a slight variation of the procedure developed for backward recursion. The solution that is obtained, though, is

no longer an alternating series of states and decision. The randomness in the problem prevents us from knowing which state the system will be in from one transition to the next. In fact, a solution corresponds to a conditional strategy that indicates what the best course of action is should we find ourselves in a particular state. Given that we are in state **x**, for example, the optimal strategy might say to take action **y**. In most cases, the uncertainty of transition makes it difficult to limit the state space in the forward generation process. Thus it is often necessary to generate the complete state space and associate a decision with each element.

Note that the use of the expected value criterion (13) in the objective function is based on the implicit assumption that the decision maker is risk neutral. This would be the case for a large organization like a government agency, or when the same (or similar) problem had to be solved repeatedly. If these conditions aren't present, it would be best to use a different objective, such as minimizing the maximum possible loss or maximizing expected utility. The former is an extremely conservative strategy because it ignores the probability of loss and only tries to minimize the worst possible outcome. The latter requires that the decision maker specify his or her attitude toward risk in the form of a utility function. Someone who is optimistic or willing to take risks would have a convex utility function. For this person, unit increases in gains provide increasingly greater rewards. Someone who is conservative or wishes to avoid risk would have a concave utility function where unit increases in gains provide decreasingly fewer rewards. A decision maker who is risk neutral would have a linear utility function.

Stochastic Equipment Replacement

To illustrate how dynamic programming can be applied to situations where outcomes are uncertain, consider a problem where a machine of a certain type is required for n years. Different machines are available with different life expectancies. The life expectancy is not a fixed number but rather a random variable defined over several years with accompanying probabilities. The data for three options are given in Table 14. Machine 1, for example, costs \$100 and will last anywhere from one to three years with probabilities $p_1 = 0.3$, $p_2 = 0.5$ and $p_3 = 0.2$, respectively. For simplicity, we do not consider the time value of money but it should be borne in mind that spending \$100 today is not the same as spending \$100 five years from now.

Table 14. Data for machine replacement example

	Machine 1			Machine 2				Machine 3			
Cost (\$)	100			200				300			
Life expectancy (yr)	1	2	3	2	3	4	5	4	5	6	7
Probability	0.3	0.5	0.2	0.1	0.3	0.4	0.2	0.1	0.2	0.4	0.3

A machine is replaced at the end of its life with one of the three options. At time n the existing machine is discarded. No salvage value is

assumed. The goal is to determine an optimal strategy over the n-year period. We will solve the problem for $n = 10$.

The DP model is outlined in Table 15 where it can be seen that the state vector **s** has a single component corresponding to the number of years that have transpired since time 0. The state space $\boldsymbol{S}$ ranges from 0 to 16. Although the problem ends once the state $s = 10$ is reached, it might be that at the end of year 9 we buy machine 3 and it lasts 7 years. This would put us at the end of year 16, hence the need for states 11 – 16 in the definition of $\boldsymbol{S}$. To account for the uncertainty in the problem, we introduce a random variable, call it x, corresponding to life expectancy of a machine. For each of the three options, x follows the appropriate distribution given in Table 14. If we are in state 3 and purchase machine 2, for example, there is a 0.4 chance ($p_4 = 0.4$ for machine 2) that we will wind up in state 7. In general, $s' = s + x$ which defines the transition function. Thus s' is also a random variable.

Table 15. DP model for the stochastic equipment replacement problem

Component	Description
State	**s** = (s), where s is the years since the start of the problem.
Initial state set	$\boldsymbol{I}$ = {(0)}; the process begins at time 0.
Final state set	$\boldsymbol{F}$ = {(10,…,16)}; the process ends at the end of year 10. We add states 11 through 16 to take care of the possibility that an option lasts beyond the planning horizon.
State space	$\boldsymbol{S}$ = {0, 1, 2,…,16}; the state space has 17 elements.
Decision	**d**(s) = (d), the machine option to be installed at time s.
Feasible decision	$\boldsymbol{D}(s)$ = {1, 2, 3} for $s < 10$ = for $s \geq 10$
Transition function	$s' = T(s, d)$ or $s' = s + x$, where x is the life of the option, a random variable whose probability distribution depends on d as given in Table 12.
Decision objective	$z(\mathbf{s}, \mathbf{d}, \mathbf{s}') = c(d)$, the cost of decision d as given in Table 12.
Final value function	$f(\mathbf{s}) = 0$ for **s** $\boldsymbol{F}$ Boundary conditions; no salvage value is assumed.
Recursive equation	$f(\mathbf{s}) = \text{Minimize}\{E[c(d) + f(s')] : d \quad \boldsymbol{D}(s)\}$, where $E[c(d) + f(s')] = r(s, d) = \sum_{s' \ \boldsymbol{S}} P(s':d)(c(d) + f(s''))$

The computations for the example are carried out in Table 16. The columns for $\mathbf{s}'$ and $f(\mathbf{s}')$ are not included because the transitions are governed by a probability distribution and therefore uncertain. The values of $r(\mathbf{s}, \mathbf{d})$ are determine through an expected value computation and, as indicated by Eq. (13), are the sum of products.

Table 16. Backward recursion for equipment replacement problem

(1)	(2)	(3)	(4)	(5)	(6)	(7)
Index	$\mathbf{s}$	$\mathbf{d}$	z	$r(\mathbf{s}, \mathbf{d})$	$f(\mathbf{s})$	$\mathbf{d}^*(\mathbf{s})$
1	10 - 16	—	—	—	0	—
2	9	1	100	100	100	1
3	8	1	100	130	130	1
		2	200	200		
4	7	1	100	189	189	1
		2	200	210		
		3	300	300		
5	6	1	100	241.7	241.7	1
		2	200	243		
		3	300	300		
6	5	1	100	293	293	1
		2	200	297.9		
		3	300	310		
7	4	1	100	346.6	333	3
		2	200	352.9		
		3	300	333		
8	3	1	100	394.7	384.9	3
		2	200	403.4		
		3	300	384.9		
9	2	1	100	440.6	440.6	1
		2	200	455.7		
		3	300	444.		
10	1	1	100	491.2	491.2	1
		2	200	503.9		
		3	300	492.2		
11	0	1	100	544.6	544.6	1
		2	200	551.3		
		3	300	545.3		

After setting $f(\mathbf{s}) = 0$ for all $\mathbf{s}$ in $\boldsymbol{F}$, the backward recursion begins at the end of year 9 (beginning of year 10) with $s = 9$. Although any of the three machines can be purchased at this point, the decision $d = 1$ dominates

the other two options because all lead to a state in $\boldsymbol{F}$ and Machine 1 is the least expensive; that is, $c(1) = \$100 < c(2) = \$200 < c(3) = \$300$. Therefore, we only include $d = 1$ in Table 16.

Moving to $s = 8$, we can either purchase Machine 1 or Machine 2 (the third option is dominated by the second so is not considered). The computations are as follows:

$d = 1$:

$$\begin{aligned} r(8, 1) &= p_1(c(1) + f(9)) + p_2(c(1) + f(10)) + p_3(c(1) + f(11)) \\ &= 0.3(100 + 100) + 0.5(100 + 0) + 0.2(100 + 0) \\ &= 130 \end{aligned}$$

$d = 2$:

$$\begin{aligned} r(8, 2) &= p_2(c(2) + f(10)) + p_3(c(2) + f(11)) + p_4(c(2) + f(12)) \\ &\quad + p_5(c(2) + f(13)) \\ &= 0.1(200 + 0) + 0.3(200 + 0) + 0.4(200 + 0) \\ &\quad + 0.2(200 + 0) \\ &= 200 \end{aligned}$$

The recursive equation is

$$f(8) = \text{Minimize}\begin{Bmatrix} r(8,1) = 130 \\ r(8,2) = 200 \end{Bmatrix} = 130.$$

These results are shown in columns (5) and (6) of Table 16. The optimal decision given that we are in state $s = 8$ is to buy Machine 1. Therefore, $\mathbf{d}^*(8) = 1$ as indicated in column (7).

Working backwards, it is necessary to consider all three options for states $s = 7$ through $s = 0$. The computations for the sole initial state $s = 0$ are given below.

$d = 1$:

$$\begin{aligned} r(0, 1) &= p_1(c(1) + f(1)) + p_2(c(1) + f(2)) + p_3(c(1) + f(3)) \\ &= 0.3(100 + 491.2) + 0.5(100 + 440.6) + 0.2(100 + 384.9) \\ &= 544.6 \end{aligned}$$

$d = 2$:

$$\begin{aligned} r(0, 2) &= p_2(c(2) + f(2)) + p_3(c(2) + f(3)) + p_4(c(2) + f(4)) \\ &\quad + p_5(c(2) + f(5)) \\ &= 0.1(200 + 440.6) + 0.3(200 + 384.9) + 0.4(200 + 333) \\ &\quad + 0.2(200 + 293) \\ &= 551.3 \end{aligned}$$

$d = 3$:

$$r(0, 3) = p_4(c(3) + f(4)) + p_5(c(3) + f(5)) + p_6(c(3) + f(6))$$

$$+ p_7(c(3) + f(7))$$
$$= 0.1(300 + 333) + 0.3(300 + 293) + 0.4(300 + 241.7)$$
$$+ 0.2(300 + 189)$$
$$= 545.3$$

The recursive equation is

$$f(8) = \text{Minimize} \begin{Bmatrix} r(0,1) = 544.6 \\ r(0,2) = 551.3 \\ r(0,3) = 545.3 \end{Bmatrix} = 544.6.$$

The optimal policy is to buy Machine 1 at the initial state ($\mathbf{d}^*(0) = 1$) and at every state except $s = 3$ or 4 when it is optimal to buy Machine 3 ($\mathbf{d}^*(3) = \mathbf{d}^*(4) = 3$). Once again we point out that no optimal path through the state space can be identified because the transition at each state is uncertain.

A more realistic example would include an age-dependent salvage value for each machine still functioning when it enters a final state in $\boldsymbol{F}$. One way to incorporate this feature in the model is to redefine the decision objective $z(\mathbf{s}, \mathbf{d}, \mathbf{s}')$ in Table 15 to reflect salvage values. For any decision d and state s that leads to a state s' in $\boldsymbol{F}$, $z(\mathbf{s}, \mathbf{d}, \mathbf{s}')$ would no longer be $c(d)$ but, say $c(s'-s, d)$, where $s' - s$ is the age of the machine when it enters state s'. However, because the problem ends when we enter state 10, the cost should really be a function of $10 - s$ rather than $s' - s$. For example, if Machine 1 has a salvage value of \$40 after one year of use, \$20 after two years of use and \$0 after three years of use, $z(9,1,10) = 100 - 40 = 60$, $z(9,1,11) = z(9,1,12) = 60$, while $z(8,1,10) = 100 - 20 = 80$, $z(8,1,11) = 80$, and $z(7,1,10) = 100 - 0 = 100$.

Other cost functions are possible but it would not be correct to simply assign a terminal value corresponding to a salvage value to $f(\mathbf{s})$ for $\mathbf{s} \in \boldsymbol{F}$ unless $f(\mathbf{s})$ where machine- and age-independent. Finally, if it were assumed that a machine had a salvage value at every state it would be necessary to redefine $z(\mathbf{s}, \mathbf{d}, \mathbf{s}')$ accordingly for all $\mathbf{s}$ and $\mathbf{s}'$.

20.7 Exercises

1. For the rectangular arrangement of Fig. 1, what state variable definitions are acceptable?

2. For the binary knapsack problem with two resource constraints, give a state variable definition that is acceptable and one that is unacceptable according to the requirements stated in Section 20.1.

3. Consider the general sequencing problem presented in Section 19.6 and let $n = 4$. We want to add the condition that job 3 must precede job 2 in the sequence. Incorporate this constraint in the definition of $\boldsymbol{D}(\mathbf{s})$ and use forward generation to find the set of all reachable states. The initial state is (0, 0, 0, 0).

4. An integer knapsack problem is given below. The values of the decisions are unbounded from above.

$$\begin{aligned} \text{Maximize } z = \quad & 6x_1 + 8x_2 + 11x_3 \\ \text{subject to} \quad & 3x_1 + 5x_2 + 7x_3 \le 10 \\ & x_j \ge 0 \text{ and integer}, j = 1, 2, 3 \end{aligned}$$

   Consider the dynamic programming model based on the line partitioning problem discussed in Section 19.4.

   a. Write out the state space obtained with exhaustive generation.

   b. Write out the state space obtained with forward generation.

5. Consider the knapsack problem in Exercise 4 but restrict the variables to be either 0 or 1. Consider the dynamic programming model based on the resource allocation problem presented in Section 19.3.

   a. Write out the state space obtained with exhaustive generation.

   b. Write out the state space obtained with forward generation.

Use backward recursion for Exercises 6 - 10. Show the table format introduced in Section 20.3 with all the states. Recover the optimal path and identify the accompanying states and decisions. Perform the calculations manually and verify with the Teach DP add-in.

6. Solve the path problem in Fig.1 with a rectangular arrangement of states; let s_1 be the x_1-coordinate and s_2 be the x_2-coordinate. Use the arc lengths shown in Fig. 3.

7. Solve the reliability problem given as Exercise 29 in the DP Models chapter. Limit the state space to the following.

$$\boldsymbol{S} = \{(1, 0), (2, 0), (2, 100), (2, 200), (3, 0), (3, 50), (3, 100), (3, 150), (3, 200), (3, 250), (3, 300), (4, 0), (4, 40), (4, 80), (4, 50), (4, 90), (4, 100), (4, 130), (4, 140), (4, 150), (4, 180), (4, 190), (4, 200), (4, 230), (4, 240), (4, 250), (4,$$

280), (4, 290), (4, 300), (5, 0), (5, 40), (5, 80), (5, 50), (5, 90), (5, 100), (5, 130), (5, 140), (5, 150), (5, 180), (5, 190), (5, 200), (5, 230), (5, 240), (5, 250), (5, 280), (5, 290), (5, 300)}

8. Solve the line partitioning problem given as Example 4 in the DP Models chapter.

9. Solve the integer knapsack problem given as Exercise 10 in the DP Models chapter.

10. Solve the 4-job sequencing problem given as Example 10 in the DP Models chapter.

11. Find a state variable definition for the path problem with turn penalties that allows both forward and backward transition functions. Use your algorithm to solve the rotated path problem in Fig. 3 with a turn penalty of 10 for up turns and a penalty of 5 for down turns.

12. Find the backward decision set and backward transition function for the binary knapsack problem with one or more constraints.

13. Find the backward decision set and backward transition function for the unbounded integer knapsack problem with one constraint. Use only one state variable.

14. Consider the following optimization problem.

$$\text{Maximize } J = \sum_{i=1}^{n-1} g_i(x_i, x_{i+1})$$

$$\text{subject to } \sum_{i=1}^{n} x_i = b$$

$$x_i \geq 0 \text{ and integer, } i = 1, \ldots, n$$

Give a dynamic programming formulation based on backward recursion. Be sure to indicate the optimal value function, the boundary conditions, and what the solution would be when the algorithm is executed.

15. Use backward recursion to solve the following problem manually. Draw the associated decision network. Verify your answer by solving the same problem with the Teach DP add-in.

$$\text{Maximize } z = 3x_1 + 5x_2 + 4x_3 + 6x_4 + 6x_5 + x_6$$

$$\text{subject to } 2x_1 + 3x_2 + 4x_3 + 7x_4 + 8x_5 + x_6 \leq 15$$

$$x_j = 0 \text{ or } 1, \; j = 1, \ldots, 6$$

16. Solve the problem below by hand using forward recursion.

$$\begin{aligned}\text{Minimize } z = {} & 7x_1 + 4x_2 + 6x_3 + x_4 + 3x_5 + 2x_6 + 5x_7 \\ \text{subject to } & 2x_1 + x_2 + 5x_3 + 4x_4 + 2x_5 + 5x_6 + 3x_7 \geq 13 \\ & x_1 + 3x_2 + 2x_3 + 6x_4 + x_5 + 4x_6 + 5x_7 \geq 15 \\ & x_j = 0 \text{ or } 1,\ j = 1,\ldots,7\end{aligned}$$

17. Solve the problem below by hand with the reaching procedure.

$$\begin{aligned}\text{Minimize } z = {} & 8x_1 + 3x_2 + 7x_3 + 5x_4 + 6x_5 + 9x_6 + 5x_7 + 3x_8 \\ \text{subject to } & x_1 + x_2 + 4x_3 + 4x_4 + 5x_5 + 9x_6 + 8x_7 + 8x_8 \geq 20 \\ & x_j = 0 \text{ or } 1,\ j = 1,\ldots,8\end{aligned}$$

18. Solve the following integer knapsack problem using the reaching method with bound elimination. Compute upper bounds by solving the LP relaxation. Develop your own procedure for finding lower bounds (feasible solutions).

$$\begin{aligned}\text{Maximize } z = {} & 5x_1 + 2x_2 + 8x_3 + 5x_4 + 6x_5 + 7x_6 \\ \text{subject to } & 2x_1 + x_2 + 5x_3 + 3x_4 + 4x_5 + 4x_6 \leq 23 \\ & x_j \geq 0 \text{ and integer},\ j = 1,\ldots,6\end{aligned}$$

19. Solve the job sequencing problem below using the reaching method in conjunction with the bounding elimination test described in the text.

Job j	Processing time $p(j)$	Due date $d(j)$	Cost per day $a(j)$
1	6	11	\$70
2	10	13	50
3	7	15	80
3	5	9	30
4	12	20	60

20. (*Project Scheduling – Longest Path Problem*) A project, such as building a house or making a film, can be described by a set of activities, their duration, and a list of precedent relations that indicate which activities must be completed before others can start. For example, the frame of a house cannot be erected until the foundation is laid. The problem of determining the length of a project can be represented by a directed acyclic network, where the goal is to find the longest path (critical path) from a nominal start node 1 to a nominal finish node n.

In an activity-on-the-node (AON) network model, the nodes correspond to project activities while the arcs indicate precedence relations. One start node and one

finish node are introduced to tie the other nodes together. A single arc is introduced between the start node and each node associated with an activity that has no predecessors. Similarly, a single arc is introduced between each node associated with an activity that has no successors and the finish node.

Consider the set of activities in the following table for the development of a consumer product. The project begins by estimating the potential demand and ends after market testing.

Activity	Description	Immediate predecessors	Duration (weeks)
A	Investigate demand	—	3
B	Develop pricing strategy	—	1
C	Design product	—	5
D	Conduct promotional cost analysis	A	1
E	Manufacture prototype models	C	6
F	Perform product cost analysis	E	1
G	Perform final pricing analysis	B,D,F	2
H	Conduct market test	G	8

a. Draw the AON network for this project. Number the nodes such that if arc (i, j) exists, $i < j$.

b. Develop a forward recursion algorithm for finding the longest path from node 1 to node j, for $j = 2,\ldots,n$. Let ES_k be the earliest time activity k can start, where $ES_1 = 0$, and let L_k be its duration. The recursion should be based on ES_k.

c. Develop a backward recursion algorithm for finding the longest path from node j to node n, for $j = n–1,\ldots,1$. Let LF_k be the latest time activity k can finish, where $LF_n = ES_n$. The recursion should be based on LF_k.

c. Apply your forward and backward algorithms to the AON network developed in part *a.*

d. The *free slack* of an activity is the time that it can be delayed without delaying either the start of any succeeding activity or the end of the project. *Total slack* is the time that the completion of an activity can be delayed without delaying the end of the project. A delay of an activity that has total slack but no free slack reduces the slack of other activities in the project. Activities with no slack are called *critical* and fall on the longest path.

Write out general expressions that can be used to find the free slack (FS_k) and total slack (TS_k) of an activity k. Note that it might be convenient to introduce the early finish time of activity k, EF_k, as well as its late start time, LS_k. Determine FS_k and TS_k for all k in the project.

e. Develop a linear programming model that can be used to find the longest path in an AON network. Let the decision variable be t_j = the earliest time activity j can start.

21. Using forward recursion, solve the 2-dimensional integer knapsack problem for the data given in the table below. Assume a budget of \$13 and a weight capacity of 20 lb.

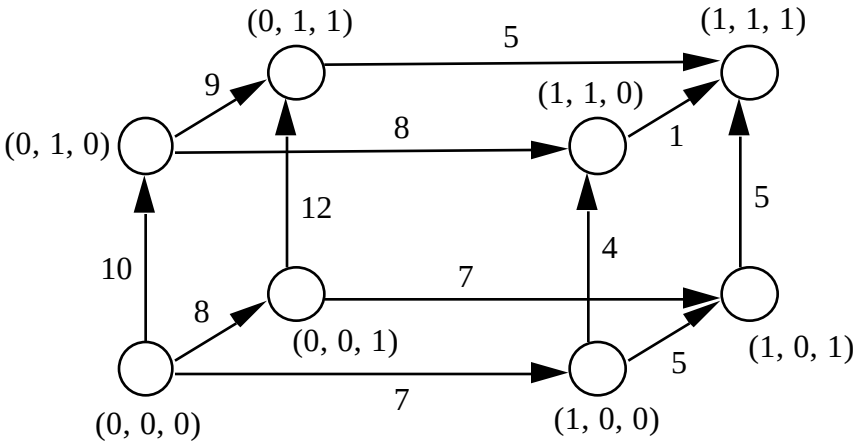
Item	1	2	3
Cost, \$	1.0	2.5	3.0
Benefit, \$	6.0	8.0	11.0
Weight, lb.	3.0	5.0	7.0

22. (*Path Problem with Uncertainty*) Consider the path problem shown in Fig. 3. You have just discovered that the steering mechanism in your car is malfunctioning so it will not always travel in the direction that you want it to. When you decide to go up, you will actually go up with a 0.9 probability and go down with a 0.1 probability. If you decide to go down, you will actually go up with a 0.4 probability and down with a 0.6 probability. These rules do not hold where only one direction of travel is possible (i.e., nodes 4 and 6). At those nodes travel in the feasible direction is assured. In this risky situation, you decide to make decisions that will minimize your expected travel time. Formulate and solve the dynamic programming model for this problem.

23. Nickoli Putin is considering a game for which his probability of winning is 0.6 and his probability of losing is 0.4. He has \$50 and decides to play twice. At each play, the bet will be either \$10 or the whole amount currently held. If he wins, the amount bet will be added to his total. If he loses, the amount bet will be deleted from his total. Thus, in the first play the bet can be either \$10 or \$50. If the bet is \$10, Nickoli can end the play with either \$40 or \$60 depending on whether he wins or loses. If the bet is \$50, the results may be either 0 or \$100. After the first play, he will bet again using the same rules. After the second play, he will go home with the amount remaining.

    Use dynamic programming to determine the strategy that will maximize Mr. Putin's expected holdings after two plays. Reduce the number of states by considering only those states that can be reached during the play of the game.

24. The figure below shows a decision network where the numbers in parentheses adjacent to the nodes represent states and the numbers along the arcs represent the costs of decisions. The state vector has three components, $\mathbf{s} = (s_1, s_2, s_3)$, and the decision vector has a single component that takes on the values 1, 2 or 3. The objective is to minimize the cost of going from (0, 0, 0) to (1, 1, 1).



a. Write the general forward transition equation which would define the transitions shown in the figure.

b. Write a general expression that defines the feasible decisions at each state.

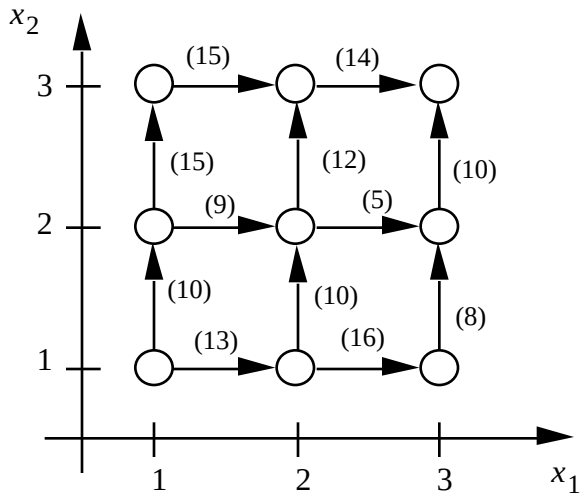
c. Solve the problem with backward recursion.

d. Write the general backward transition equation which would define the transitions shown in the figure.

e. Solve the problem with forward recursion.

f. Solve the problem with the reaching procedure.

25. Consider the model for the path problem with turn penalties described in Section 19.5. A specific network with $n = 3$ is shown below. The values of $a(\mathbf{s}, d)$ are given along the arcs. The penalties are $\pi_1 = 10$ for a left turn and $\pi_2 = 5$ for a right turn.



a. For the initial state (1, 1, 0), show the path **P** and evaluate $z(\mathbf{P})$ for the sequence of decisions 1, 0, 1, 0. Use vector notation to describe **P**.

b. Solve the problem with backward recursion using the tabular format suggested in the text for the computations.

c. Assume the optimal policy function $\mathbf{d}^*(\mathbf{s})$ is given in the table below. Use the forward recovery procedure to find the optimal path starting at the initial state (1,1,1).

State	(1, 1, 0)	(1, 1, 1)	(1, 2, 0)	(1, 3, 0)	(2, 1, 1)	(2, 2, 0)	(2, 2, 1)
$\mathbf{d}^*(\mathbf{s})$	0	1	1	1	0	1	0

State	(2, 3, 0)	(2, 3, 1)	(3, 1, 1)	(3, 2, 0)	(3, 2, 1)	(3, 3, 0)	(3, 3, 1)
$\mathbf{d}^*(\mathbf{s})$	1	1	0	0	0	—	—

26. The tables below show the transition and decision objective functions for a dynamic programming model. A dash indicates that the transition is not allowed. The states are listed in lexicographic order (A, B, C, D, E, F, G, H). States A and H are the initial and final states, respectively. The recursive equation is

$$f(\mathbf{s}) = \text{Minimize}\{z(\mathbf{s}, d) + f(\mathbf{s}') : d \quad \boldsymbol{D}(\mathbf{s}), \mathbf{s}' = T(\mathbf{s}, d)\}$$

and the boundary condition is $f(\text{H}) = 0$.

Use backward recursion to find the optimal value of $f(\text{A})$ and the corresponding path through the state space.

Transition function, $T(\mathbf{s}, d)$

s	A	B	C	D	E	F	G	H
$d = 1$	B	F	F	E	F	H	H	—
$d = 2$	C	C	E	G	H	G	—	—
$d = 3$	D	—	—	—	G	—	—	—

Decision objective, $z(\mathbf{s}, d)$

s	A	B	C	D	E	F	G	H
$d = 1$	2	30	21	8	8	11	15	—
$d = 2$	8	5	12	15	21	3	—	—
$d = 3$	12	—	—	—	6	—	—	—

27. You have a roll of paper 15 ft. long. The data in the table below shows your orders for paper in different sizes. You want to cut the roll to maximize profit. Model this problem as a dynamic program and solve.

Order (no. of rolls)	Length (ft)	Profit ($ per roll)
1	10	5
2	7	4
1	6	3
3	4	2.5

28. Solve the following integer, nonlinear program using backward recursion. Give all alternate optima.

$$\text{Maximize } z = 5x_1 + (x_2)^2 + (x_3)^3$$

$$\text{subject to } x_1 + x_2 + (x_3)^2 \le 6$$

$$x_j \in \{0, 1, 2, 3\},\ j = 1, 2, 3$$

Using the tabularized computational results, find the new solution when the right-hand side of the constraint is changed to 5.

29. (*Deterministic Dynamic Inventory - Production Problem*) A firm wishes to establish a production schedule for an item during the next n periods. Assume that demand d_t ($t = 1, \ldots, n$) is known and that the manufacturing time is negligible. This means that production in period t can be used to fill the demand in that period. If more items are produced than needed, the excess is held in inventory at a cost. The goal is to devise a schedule that minimizes the total production and inventory holding costs subject to the requirement that all demand is satisfied on time and that the inventory at the end of period n is at some predetermined level. To formulate the model, define the following policy variables.

x_i = production quantity in period t

i_t = inventory at the *end* of period t

such that x_t and i_t are nonnegative integers. Now let $c_t(x_t, i_t) = c_t(x_t) + h_t(i_t)$ be the cost functions for each period t, where

$$c_t(x_t) \ge 0, \quad c_t(0) = 0, \quad h_t(i_t) \ge 0, \quad h_t(0) = 0.$$

Finally, the inventory balance equations are

$$i_t = i_{t-1} + x_t - d_t \text{ for } t = 1, \ldots, n$$

where it is assumed that $i_0 = 0$ and that each demand d_t is a nonnegative integer.

For a 4-period planning horizon ($n = 4$), the production cost functions are given in the table below. For example, the total cost when $x_1 = 12$ is 86 (= 10 × 5 + 6 × 5 + 3 × 2). Note that the data in the table imply that the production cost functions are concave; they exhibit economies of scale. Assume the holding cost functions are $h_t(i_t) = h_t i_t$, where $h_1 = 1$, $h_2 = 2$, and $h_3 = 1$. Let the demand requirements be

$$d_1 = 10,\ d_2 = 3,\ d_3 = 17,\ d_4 = 23.$$

a. Develop a dynamic programming model for this problem and then use it to find an optimal production schedule. Be sure to indicate total production x_t and ending inventory i_t in each period t.

b. Indicate the impact of requiring an additional unit in period 1 (that is, let $d_1 = 11$). Also, consider separately, an additional unit in period 2; in period 3; in period 4.

c. Find an optimal schedule where all $h_t = 0$. Also where all $h_t = 5$.

	Unit production cost			
No. of items, k	Period 1	Period 2	Period 3	Period 4
$1 \le k \le 5$	10	8	12	9
$6 \le k \le 10$	6	5	7	8
$11 \le k \le 15$	3	3	6	4
$16 \le k$	1	2	1	4

d. Find an optimal production schedule where the demand requirements are revised such that $d_1 = 3$ and $d_2 = 10$. Also, when $d_3 = 23$ and $d_4 = 17$, and when $d_2 = 17$ and $d_3 = 3$. (Solve each case separately)

e. Find an optimal production schedule for the original data when the holding cost function is $h_t(i_t) = \sqrt{i_t}$ for every Period t.

30. (*Inventory with Backlogging*) Consider the data in the previous exercise and assume that backlogging is permitted; i.e., shortages in period t can be filled by production in some future period. Let the inventory holding and backlog penalty cost function be

$$h_t(i_t) = \begin{cases} h_t i_t & \text{if } i_t \ge 0 \\ -p_t i_t & \text{if } i_t \le 0 \end{cases} \quad \text{for every period } t.$$

(All demand must be met by the end of the horizon.) In each part below, find an optimal production schedule using dynamic programming and indicate ending inventory or backlog in each period.

a. $h_t = p_t = 0$.

b. $h_t = p_t = 1$.

c. Suppose the inventory holding and backlog penalty cost function is $h_t(i_t) = \sqrt{|i_t|}$ for every period t.

31. The following information is available for a stochastic dynamic program.

    i. $\mathbf{s} = (s_1, s_2)$, where $1 \le s_1 \le 3$ and $1 \le s_2 \le 3$

    ii. $\boldsymbol{I} = \{(1,1)\}$ and $\boldsymbol{F} = \{(3, s_2) : 1 \le s_2 \le 3\}$

    iii. $\mathbf{d} = (d)$, where for $\mathbf{s} \in \boldsymbol{S}$, $\boldsymbol{D}(\mathbf{s}) = \{0, 1\}$

    iv. The forward transition function for the first state variable is $s_1' = s_1 + 1$.

    v. The transitions with respect to the second state variable are uncertain and are governed by the probabilities given in the following tables. These probabilities are independent of the value of s_1.

Transition probabilities, $P(s_2' : s_2, d)$

	For $s_2 = 1$			For $s_2 = 2$			For $s_2 = 3$		
s_2'	1	2	3	1	2	3	1	2	3
$d = 0$	0.1	0.3	0.6	0.3	0.5	0.2	0.5	0.5	0.0
$d = 1$	0.5	0.2	0.3	0.2	0.4	0.6	0.0	0.3	0.7

The only costs involved in the problem are those associated with the final states. The values of $f(\mathbf{s})$ for $\mathbf{s} \in \boldsymbol{F}$ are $f(3,1) = 30$, $f(3,2) = 20$ and $f(3,3) = 10$. There are no costs associated with the decisions. Use backward recursion to minimize the expected cost of going from the initial state (1, 1) to a final state $\mathbf{s} \in \boldsymbol{F}$.

32. Consider the stochastic dynamic programming model of a finite inventory problem given in the table below.

Component	Description
State	$\mathbf{s} = (s_1, s_2)$, where s_1 = current period s_2 = inventory level at the beginning of the current period
Initial state set	$\boldsymbol{I} = \{(1, s_2) : s_2 = 0,1,2,3\}$
Final state set	$\boldsymbol{F} = \{(T+1, s_2) : s_2 = 0,1,2,3\}$, where T is the time horizon.
State space	$\boldsymbol{S} = \{(s_1, s_2) : s_1 = 1,\ldots,T+1, s_2 = 0,1,2,3\}$
Decision	$\mathbf{d} = (d)$, where d is the amount of product to order (restricted to integer values).

Feasible decision set	$\boldsymbol{D}(\mathbf{s}) = \{0, 1, 2, 3 : \mathbf{s}' = T(\mathbf{s}, \mathbf{d}) \quad \boldsymbol{S}\}$, such that d must be chosen so that there is a zero probability that the inventory level exceeds 3 or falls below 0.
Transition function	$\mathbf{s}' = T(\mathbf{s}, \mathbf{d})$, where $s_1' = s_1 + 1$ $s_2' = s_2 - \delta(s_1) + d$ Here, $\delta(s_1)$ is the demand in period s_1 and follows the probability distribution given in the next table.
Decision objective	$z(\mathbf{s}, \mathbf{d}, \mathbf{s}') = \begin{cases} 5 + d + 2(s_2')^2 & \text{for } d > 0 \\ 2(s_2')^2 & \text{for } d = 0 \end{cases}$
Path objective	Minimize $z(\mathbf{P}) = E\left[\sum_{\mathbf{s}, \mathbf{d}} z(\mathbf{s}, \mathbf{d}, \mathbf{s}')\right]$
Final value function	$f(\mathbf{s}) = 0$ for $\mathbf{s} \quad \boldsymbol{F}$
Recursive equation	$f(\mathbf{s}) = \text{Minimize } E[\{a(\mathbf{s}, d) + f(\mathbf{s}') : d \quad \mathbf{D}(\mathbf{s})\}]$

Probability distribution for demand in period 1

δ_1	0	1	2
$P(\delta_1)$	0.4	0.4	0.2

Assume that we have already computed the optimal value functions

$$f(2, 0) = 100, \ f(2, 1) = 30, \ f(2, 2) = 40, \ f(2, 3) = 50.$$

Now find the optimal policy for period 1 when the inventory level at the beginning of the period is 2.

33. Solve the soot collection problem (Exercise 28 in the DP Models chapter) with forward recursion. Provide the recursive equation.

34. The matrix below shows the travel cost between city pairs for a traveling salesman (Section 19.6 of the DP Models chapter).

a. Suggest a heuristic that could be used to find an upper bound on the optimal solution.

Cost matrix for city pairs

	1	2	3	4	5	6
1	—	32	30	27	25	40
2	9	—	16	10	30	25
3	15	16	—	21	41	14
4	31	7	24	—	18	21
5	22	31	7	41	—	9
6	14	12	16	15	22	—

b. Solve the corresponding assignment problem to find a lower bound on the cost of the optimal tour.

c. Solve the TSP by hand using a reaching algorithm with bounds.

Bibliography

Bard, J.F, "Assembly Line Balancing with Parallel Workstations and Dead Time," *International Journal of Production Research*, Vol. 27, No. 6, pp. 1005-1018, 1989

Bard, J.F., K. Venkatraman and T.A. Feo, "Single Machine Scheduling with Flow Time and Earliness Penalties," *Journal of Global Optimization*, Vol. 3, pp. 289-309, 1993.

Bellman, R.E., *Dynamic Programming*, Princeton University Press, Princeton, NJ, 1957.

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

Denardo, E.V., *Dynamic Programming: Models and Applications*, Prentice Hall, Engelwood Cliffs, NJ, 1982.

Denardo, E.V. and B.L. Fox, "Shortest Route Methods, 1. Reaching, Pruning and Buckets," *Operations Research*, Vol. 27, pp. 161-186, 1979.

Dijkstra, E.W., "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, Vol. 1, pp. 269-271, 1959.

Dreyfus, S.E. and A.M. Law, *The Art and Theory of Dynamic Programming*, Academic Press, New York, 1977.

Kaufmann, A., *Graphs, Dynamic Programming and Finite Games*, Academic Press, New York, 1967.

Schrage, L. and K.R. Baker, "Dynamic Programming Solution of Sequencing Problems with Precedence Constraints," *Operations Research*, Vol. 26, No. 3, pp. 444-449, 1973.

Wagner, H. and T. Whitin, "Dynamic Version of the Economic Lot Size Model," *Management Science*, Vol. 5, pp. 89-96, 1958.

Yen, J.Y., "Finding the Lengths of All Shortest Paths in N-Node Nonnegative Distance Complete Networks Using $\frac{1}{2}N^3$ Additions and N^3 Comparisons," *Journal of the Association of Computing Machinery*, Vol. 19, pp. 423-424, 1972.