Dynamic Programming Methods.S2
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

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

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

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 $T(\mathbf{s}, \mathbf{d})$. Restating

(6) in these terms first requires the replacement of **s** by **s**', and then the replacement of $\mathbf{s}_b$ by **s**. This yields

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

Note that the minimum is taken over all **s** such that **s**' can be reached from **s** when decision **d**   ***D***(**s**) is taken.

When a new state **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 **s**'. As the generation process continues a particular state **s**' may be reached through other combinations of **s** and **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 **d.**

Reaching Algorithm

The generation process starts at the initial states in ***I*** and proceeds lexicographically through the list of states already identified. For algorithmic purposes we will use the same symbol, ***S***, used to denote the complete state space, to represent this list. When a particular state **s**   ***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 **s** must come from a state that is lexicographically smaller than **s**. When **s** is reached in the generation process, all immediate predecessors of **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 $f_{\mathrm{b}}(\mathbf{s})$ for all $\mathbf{s} \quad \boldsymbol{I}$. These values must be given or implied in the problem definition. Let $\mathbf{s} \quad \boldsymbol{S}$ 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

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

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 $\mathbf{s}'$.

$$\bar{f}_{\mathrm{b}}(\mathbf{s}') = z(\mathbf{s}, \mathbf{d}, \mathbf{s}') + f_{\mathrm{b}}(\mathbf{s})$$

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

$$\bar{f}_{\mathrm{b}}(\mathbf{s}') > z(\mathbf{s}, \mathbf{d}, \mathbf{s}') + f_{\mathrm{b}}(\mathbf{s})$$

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

Step 4. Put $k \leftarrow k + 1$. If $k > l$, put $f_{\mathrm{b}}(\mathbf{s}) \leftarrow \bar{f}_{\mathrm{b}}(\mathbf{s})$ 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 $k \le l$, 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

$$f_{\mathrm{b}}(\mathbf{s}_{\mathrm{F}}) = \text{Minimize}\{f_{\mathrm{b}}(\mathbf{s}) : \mathbf{s} \quad \boldsymbol{F}\}$$

to determine the optimal path value and the final state $\mathbf{s}_{\mathrm{F}}$ 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 8. We have rearranged and renamed some of the columns to better reflect the order of the process.

Table 8. 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 ***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 ***S*** which is **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 **s**' = (2,–2) and **s**' = (2, 0), respectively. The corresponding values of $\bar{f}_b$(**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 ***S*** = {(0,0), (1, –1), (1, 1), (2, –2), (2, 0)} and we move to **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 **s**' = (2, 2) and one previously encountered state **s**' = (2, 0). For **s**' = (2, 2) we have

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

and for **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 ***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 ***F*** is single-valued, implying that $\mathbf{s}_F$ = (4, 0). The optimal path is identified by backward recovery with the help of $\mathbf{d}^*$(**s**), the optimal decision leading into each state **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 7.

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 13.3 and 13.4 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 $j = i + 1, \ldots, n$.
            $v_j \leftarrow \min\{v_j, v_i + a_{ij}\}$        (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 $c_{hi} \leq c_{ij}$). 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 $c_{hi} \leq c_{ij}$. 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. In this section, we 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 dynamic program whose state space can be represented by an acyclic network where the objective is to determine the minimum cost path from any initial state (a member of ***I***) to any final state (a member of ***F***). Let ***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 $\mathbf{s}\quad \boldsymbol{S}$ and a feasible path $\mathbf{P_s}$ that passes through **s**. Divide the path into two subpaths: $\mathbf{P}_\mathbf{s}^1$ which goes from an initial state to **s**, and $\mathbf{P}_\mathbf{s}^2$ which goes from **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

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

The reaching method starts from an initial state and derives for each state, in increasing lexicographic order, the value $f_\mathrm{b}(\mathbf{s})$. This is the objective for the optimal path from an initial state to **s**. Thus, if $\boldsymbol{P}^1$ is the set of all feasible subpaths from any state in $\boldsymbol{I}$ to **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 **s** to a final state. Letting $\boldsymbol{P}^2$ be the set of all subpaths from **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, $z_{\text{LB}}(\mathbf{s})$, on $f(\mathbf{s})$; that is,

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

We also assume that it is possible to find a feasible path $\mathbf{P}_{\text{B}}$ 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 **s** considered above. Let $z_{\text{B}} = z(\mathbf{P}_{\text{B}})$ 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 **s** compares the sum of $f_b(\mathbf{s})$ and $z_{\text{LB}}(\mathbf{s})$ to z_{B}. If

$$z_{\text{B}} \leq f_b(\mathbf{S}) + z_{\text{LB}}(\mathbf{s})$$

then any path through **s** can be no better than the solution already available, $\mathbf{P}_{\text{B}}$, and **s** can be fathomed. If the above inequality does not hold, **s** may be part of a path that is better than $\mathbf{P}_{\text{B}}$ and cannot be fathomed. This is called the *bounding elimination test,* because the lower bound $z_{\text{LB}}(\mathbf{s})$ is used in the test, and z_{B} represents an upper bound on the optimum.

The effectiveness of the test depends on the quality of the bounds z_{B} and $z_{\text{LB}}(\mathbf{s})$ and the ease with which they are determined. The value z_{B} should be as small as possible, and the value of $z_{\text{LB}}(\mathbf{s})$ 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 ROUND, and note that they are problem dependent. RELAX provides the lower bound $z_{\text{LB}}(\mathbf{s})$, 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 ROUND 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

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

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

The name ROUND is used because the procedure for determining $\mathbf{P}_{\mathbf{s}}^2$ is often the “rounding” of the solution obtained by RELAX. This is illustrated in Example 4 below. Any heuristic that provides a good feasible solution can be used to obtain the path $\mathbf{P}_{\mathbf{s}}^2$.

Note that for state **s** if

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

then $\mathbf{P}_{\mathbf{s}}^2$ must be an optimal path from **s** to a state in ***F***. The concatenation of the optimal path to **s** and the subpath $\mathbf{P}_{\mathbf{s}}^2$ yields an optimal path through **s**. In this case, **s** can also be fathomed because further exploration of the state space from state **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 **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 $f_b(\mathbf{s})$ for all $\mathbf{s}$ $\boldsymbol{I}$. These values must be given in the problem definition. Let $\mathbf{s}$ $\boldsymbol{S}$ be the lexicographically smallest state available. Set $z_B = \infty$.

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

i. If $f_b(\mathbf{s}) + z_{LB}(\mathbf{s}) \geq z_B$ 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 $\mathbf{P}^2_{\mathbf{s}}$ and compute the objective value $z(\mathbf{P}^2_{\mathbf{s}})$. If a feasible path cannot be found go to Step 3.

ii. If $f_b(\mathbf{s}) + z(\mathbf{P}^2_{\mathbf{s}}) \geq z_B$ go to Step 3.

iii. If $f_b(\mathbf{s}) + z(\mathbf{P}^2_{\mathbf{s}}) < z_B$ then put $z_B \leftarrow f_b(\mathbf{s}) + z(\mathbf{P}^2_{\mathbf{s}})$ and let $\mathbf{P}_B$ be the concatenation of $\mathbf{P}^2_{\mathbf{s}}$ and the optimal path to $\mathbf{s}$.

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

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

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 $\boldsymbol{S} \leftarrow \boldsymbol{S}$ $\{\mathbf{s}'\}$, $\mathbf{d}^*(\mathbf{s}') \leftarrow \mathbf{d}_k$ 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 $\mathbf{s}'$ $\boldsymbol{S}$ 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 **s** so we can examine the next state; also the optimal path to **s** is known). If $k \le l$, go to Step 4.

Step 6. Find the next lexicographically larger state than **s** on the list $\boldsymbol{S} \setminus \boldsymbol{F}$. Rename this state **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 ***I*** to all states in ***S*** (note that ***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 ***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 **s** ***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 BOUND Procedure

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 constraint 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 BOUND 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.

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

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

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 BOUND 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 $z_{\text{LB}}(\mathbf{s})$ at some state $\mathbf{s}$.

Job Sequencing Problem

A relaxation of the sequencing problem with due dates discussed in Section 12.5 can be obtained by redefining the objective function. Consider the nonsmooth cost function $c(j, t)$ for job j as shown in Fig. 6. 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. 7. 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 $d(j)$.

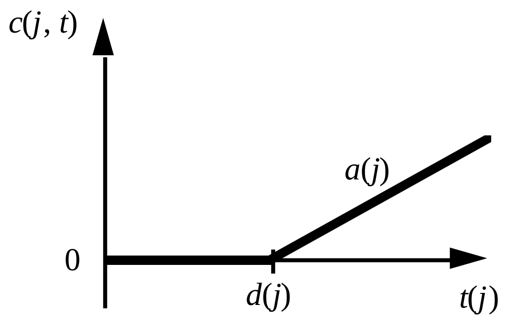

Figure 6. Cost function for sequencing problem

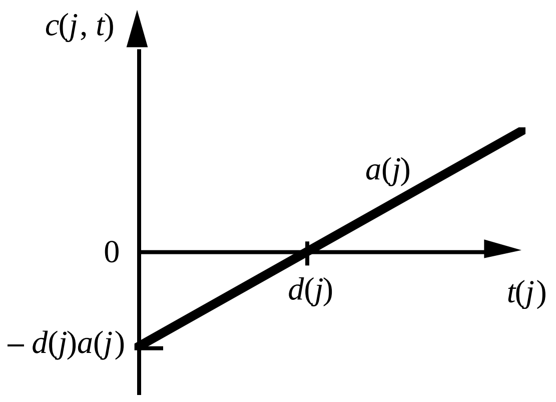

Figure 7. Lower bound cost function

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

$$z = \sum_{k=1}^{n} -d(j_k)a(j_k) + \sum_{k=1}^{n} a(j_k)t(j_k) \tag{6}$$

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 9 (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 $z_{\text{TSP}} = 63$. The highlighted cells in Table 9 are the arcs in this solution.

Table 9. 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 9 is {(1, 4), (2, 1), (3, 5), (4, 2), (5, 6), (6, 3)} with objective value $z_{\text{AP}} = 54$. The corresponding cells are highlighted in Table 10.

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 BOUND procedure of dynamic programming.

Table 10. 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 9, one of several solutions that yields the same objective value is {(1, 4), (2, 4), (3, 6), (4, 5), (5, 6), (6, 3)} with $z_{\text{RAP}} = 45$. This solution was found with almost no effort by scanning each row for the minimum cost cell. The general relationship between the three solutions is $z_{\text{RAP}} \leq z_{\text{AP}} \leq z_{\text{TSP}}$ 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.

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

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

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

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 ROUND subroutine for the example. For the RELAX subroutine we solve the original problem without the integrality requirements on x_j. That is, we substitute $0 \le x_j \le 1$ ($j = 1, \dots , 8$) 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 12.3. In Table 11, 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 11. 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 12. 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 12. Comparison of computational procedure

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