1 Introduction to Dynamic Programming

This book concerns the use of a method known as *dynamic programming* (DP) to solve large classes of optimization problems. We will focus on discrete optimization problems for which a set or sequence of decisions must be made to optimize (minimize or maximize) some function of the decisions. There are of course numerous methods to solve discrete optimization problems, many of which are collectively known as mathematical programming methods. Our objective here is not to compare these other mathematical programming methods with dynamic programming. Each has advantages and disadvantages, as discussed in many other places. However, we will note that the most prominent of these other methods is *linear programming.* As its name suggests, it has limitations associated with its linearity assumptions whereas many problems are nonlinear. Nevertheless, linear programming and its variants and extensions (some that allow nonlinearities) have been used to solve many real world problems, in part because very early in its development software tools (based on the simplex method) were made available to solve linear programming problems. On the other hand, no such tools have been available for the much more general method of dynamic programming, largely due to its very generality. One of the objectives of this book is to describe a software tool for solving dynamic programming problems that is general, practical, and easy to use, certainly relative to any of the other tools that have appeared from time to time.

One reason that simplex-based tools for solving linear programming problems have been successful is that, by the nature of linear programming, problem specification is relatively easy. A basic LP problem can be specified essentially as a system or matrix of equations with a finite set of numerical variables as unknowns. That is, the input to an LP software tool can be provided in a tabular form, known as a *tableaux.* This also makes it easy to formulate LP problems as a spreadsheet. This led to spreadsheet system providers to include in their product an LP solver, as is the case with Excel.

A software tool for solving dynamic programming problems is much more difficult to design, in part because the problem specification task in itself presents difficulties. A DP problem specification is usually in the form of a complex (nonlinear) *recursive* equation, called the *dynamic programming functional equation* (DPFE), where the DPFE often involves nonnumerical variables that may include sets or strings. Thus, the input to a DP tool must necessarily be general enough to allow for complex DPFEs, at the expense therefore of the simplicity of a simple table. The DP tool described in this book assumes that the input DPFE is provided in a text-based specification language that does not rely on mathematical symbols. This decision conforms to that made for other mathematical programming languages, such as AMPL and LINGO.

In this introductory chapter, we first discuss the basic principles underlying the use of dynamic programming to solve discrete optimization problems. The key task is to formulate the problem in terms of an equation, the DPFE, such that the solution of the DPFE is the solution of the given optimization problem. We then illustrate the computational solution of the DPFE for a specific problem (for linear search), either by use of a computer program written in a conventional programming language, or by use of a spreadsheet system. It is not easy to generalize these examples to solve DP problems that do not resemble linear search. Thus, for numerous dissimilar DP problems, a significant amount of additional effort is required to obtain their computational solutions. One of the purposes of this book is to reduce this effort.

In Chap. 2, we show by example numerous types of optimization problems that can be solved using DP. These examples are given, first to demonstrate the general utility of DP as a problem solving methodology. Other books are more specialized in the kinds of applications discussed, often focusing on applications of interest mainly to operations research or to computer science. Our coverage is much more comprehensive. Another important reason for providing numerous examples is that it is often difficult for new students of the field to see from a relatively small sample of problems how DP can be applied to other problems. How to apply DP to new problems is often learned by example; the more examples learned, the easier it is to generalize. Each of the sample problems presented in Chap. 2 was computationally solved using our DP tool. This demonstrates the generality, flexibility, and practicality of the tool.

In Part II of this book, we show how each of the DPFEs given in Chap. 2 can be expressed in a text-based specification language, and then show how these DPFEs can be formally modeled by a class of Petri nets, called Bellman nets. Bellman nets serve as the theoretical underpinnings for the DP tool we later describe, and we describe our research into this subject area.

In Part III of this book, we describe the design and implementation of our DP tool. This tool inputs DPFEs, as given in Part II, and produces numerical solutions, as given in Part IV.

In Part IV of this book, we present computational results. Specifically, we give the numerical solutions to each of the problems discussed in Chap. 2, as provided by our DP tool.

Appendix A of this book provides program listings for key portions of our DP tool. Appendix B of this book is a User/Reference Manual for our DP tool.

This book serves several purposes.

1. It provides a practical introduction to how to solve problems using DP. From the numerous and varied examples we present in Chap. 2, we expect readers to more easily be able to solve new problems by DP. Many other books provide far fewer or less diverse examples, hoping that readers can generalize from their small sample. The larger sample provided here should assist the reader in this process.
2. It provides a software tool that can be and has been used to solve all of the Chap. 2 problems. This tool can be used by readers in practice, certainly to solve academic problems if this book is used in coursework, and to solve many real-world problems, especially those of limited size (where the state space is not excessive).
3. This book is also a research monograph that describes an important application of Petri net theory. More research into Petri nets may well result in improvements in our tool.

1.1 Principles of Dynamic Programming

Dynamic programming is a method that in general solves optimization problems that involve making a sequence of decisions by determining, for each decision, subproblems that can be solved in like fashion, such that an optimal solution of the original problem can be found from optimal solutions of subproblems. This method is based on Bellman's Principle of Optimality, which he phrased as follows [1, p.83].

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

More succinctly, this principle asserts that "optimal policies have optimal subpolicies." That the principle is valid follows from the observation that, if a policy has a subpolicy that is not optimal, then replacement of the subpolicy by an optimal subpolicy would improve the original policy. The principle of optimality is also known as the "optimal substructure" property in the literature. In this book, we are primarily concerned with the computational solution of problems for which the principle of optimality is given to hold. For DP to be computationally efficient (especially relative to evaluating all possible sequences of decisions), there should be common subproblems such that subproblems of one are subproblems of another. In this event, a solution to a subproblem need only be found once and reused as often as necessary; however, we do not incorporate this requirement as part of our definition of DP.

In this section, we will first elaborate on the nature of sequential decision processes and on the importance of being able to separate the costs for each of the individual decisions. This will lead to the development of a general equation, the *dynamic programming functional equation* (DPFE), that formalizes the principle of optimality. The methodology of dynamic programming requires deriving a special case of this general DPFE for each specific optimization problem we wish to solve. Numerous examples of such derivations will be presented in this book. We will then focus on how to numerically solve DPFEs, and will later describe a software tool we have developed for this purpose.

1.1.1 Sequential Decision Processes

For an optimization problem of the form $\mathrm{opt}_{d\in\Delta}\{H(d)\}$, d is called the *decision*, which is chosen from a set of eligible decisions Δ, the optimand H is called the *objective function*, and $H^* = H(d^*)$ is called the *optimum*, where d^* is that value of $d \in \Delta$ for which $H(d)$ has the optimal (minimum or maximum) value. We also say that d^* *optimizes* H, and write $d^* = \arg\mathrm{opt}_d\{H(d)\}$. Many optimization problems consist of finding a set of decisions $\{d_1, d_2, \ldots, d_n\}$, that taken together yield the optimum H^* of an objective function $h(d_1, d_2, \ldots, d_n)$. Solution of such problems by *enumeration*, i.e., by evaluating $h(d_1, d_2, \ldots, d_n)$ concurrently, for all possible combinations of values of its decision arguments, is called the "brute force" approach; this approach is manifestly inefficient. Rather than making decisions concurrently, we assume the decisions may be made in some specified sequence, say $(d_1, d_2, \ldots, d_n)$, i.e., such that

$$\begin{aligned} H^* &= \mathrm{opt}_{(d_1,d_2,\ldots,d_n)\in\Delta}\{h(d_1, d_2, \ldots, d_n)\} \\ &= \mathrm{opt}_{d_1\in D_1}\{\mathrm{opt}_{d_2\in D_2}\{\ldots\{\mathrm{opt}_{d_n\in D_n}\{h(d_1, d_2, \ldots, d_n)\}\}\ldots\}\}, \end{aligned} \tag{1.1}$$

in what are known as *sequential decision processes*, where the ordered set $(d_1, d_2, \ldots, d_n)$ belongs to some decision space $\Delta = D_1 \times D_2 \times \ldots \times D_n$, for $d_i \in D_i$. Examples of decision spaces include: $\Delta = B^n$, the special case of Boolean decisions, where each decision set D_i equals $B = \{0, 1\}$; and $\Delta = \Pi(\mathbf{D})$, a permutation of a set of eligible decisions $\mathbf{D}$. The latter illustrates the common situation where decisions d_i are interrelated, e.g., where they satisfy constraints such as $d_i \neq d_j$ or $d_i + d_j \leq M$. In general, each decision set D_i depends on the decisions $(d_1, d_2, \ldots, d_{i-1})$ that are earlier in the specified sequence, i.e., $d_i \in D_i(d_1, d_2, \ldots, d_{i-1})$. Thus, to show this dependence explicitly, we rewrite (1.1) in the form

$$\begin{aligned} H^* &= \mathrm{opt}_{(d_1,d_2,\ldots,d_n)\in\Delta}\{h(d_1, d_2, \ldots, d_n)\} \\ &= \mathrm{opt}_{d_1\in D_1}\{\mathrm{opt}_{d_2\in D_2(d_1)}\{\ldots\{\mathrm{opt}_{d_n\in D_n(d_1,\ldots,d_{n-1})}\{h(d_1, \ldots, d_n)\}\}\ldots\}\}. \end{aligned} \tag{1.2}$$

This nested set of optimization operations is to be performed from inside-out (right-to-left), the innermost optimization yielding the optimal choice for d_n as a function of the possible choices for $d_1, \ldots, d_{n-1}$, denoted $d_n^*(d_1, \ldots, d_{n-1})$, and the outermost optimization $\mathrm{opt}_{d1 \in D1}\{h(d_1, d_2^*, \ldots, d_n^*)\}$ yielding the optimal choice for d_1, denoted d_1^*. Note that while the initial or "first" decision d_1 in the specified sequence is the outermost, the optimizations are performed inside-out, each depending upon outer decisions. Furthermore, while the optimal solution may be the same for any sequencing of decisions, e.g.,

$$\begin{aligned}&\mathrm{opt}_{d_1 \in D_1}\{\mathrm{opt}_{d_2 \in D_2(d_1)}\{\ldots\{\mathrm{opt}_{d_n \in D_n(d_1,\ldots,d_{n-1})}\{h(d_1, \ldots, d_n)\}\}\ldots\}\}\\ &= \mathrm{opt}_{d_n \in D_n}\{\mathrm{opt}_{d_{n-1} \in D_{n-1}(d_n)}\{\ldots\{\mathrm{opt}_{d_1 \in D_1(d_2,\ldots,d_n)}\{h(d_1, \ldots, d_n)\}\}\ldots\}\}\end{aligned} \tag{1.3}$$

the decision sets D_i may differ since they depend on different outer decisions. Thus, efficiency may depend upon the order in which decisions are made.

Referring to the foregoing equation, for a given sequencing of decisions, if the outermost decision is "tentatively" made initially, whether or not it is optimal depends upon the ultimate choices d_i^* that are made for subsequent decisions d_i; i.e.,

$$\begin{aligned}H^* &= \mathrm{opt}_{d_1 \in D_1}\{\mathrm{opt}_{d_2 \in D_2(d_1)}\{\ldots\{\mathrm{opt}_{d_n \in D_n(d_1,\ldots,d_{n-1})}\{h(d_1, \ldots, d_n)\}\}\ldots\}\}\\ &= \mathrm{opt}_{d1 \in D1}\{h(d_1, d_2^*(d_1), \ldots, d_n^*(d_1))\}\end{aligned} \tag{1.4}$$

where each of the choices $d_i^*(d_1)$ for $i = 2, \ldots, n$ is constrained by — i.e., is a function of — the choice for d_1. Note that determining the optimal choice $d_1^* = \arg \mathrm{opt}_{d_1 \in D_1}\{h(d_1, d_2^*(d_1), \ldots, d_n^*(d_1))\}$ requires evaluating h for all possible choices of d_1 unless there is some reason that certain choices can be excluded from consideration based upon a priori (given or derivable) knowledge that they cannot be optimal. One such class of algorithms would choose $d_1 \in D_1$ independently of (but still constrain) the choices for $d_2, \ldots, d_n$, i.e., by finding the solution of a problem of the form $\mathrm{opt}_{d_1 \in D_1}\{H'(d_1)\}$ for a function H' of d_1 that is myopic in the sense that it does not depend on other choices d_i. Such an algorithm is optimal if the locally optimal solution of $\mathrm{opt}_{d_1}\{H'(d_1)\}$ yields the globally optimal solution H^*.

Suppose that the objective function h is *(strongly) separable* in the sense that

$$h(d_1, \ldots, d_n) = C_1(d_1) \circ C_2(d_2) \circ \ldots \circ C_n(d_n) \tag{1.5}$$

where the decision-cost functions C_i represent the costs (or profits) associated with the individual decisions d_i, and where $\circ$ is an associative binary operation, usually addition or multiplication, where $\mathrm{opt}_d\{a \circ C(d)\} = a \circ \mathrm{opt}_d\{C(d)\}$ for any a that does not depend upon d. In the context of sequential decision processes, the cost C_n of making decision d_n may be a function not only of the decision itself, but also of the state $(d_1, d_2, \ldots, d_{n-1})$ in which the decision is made. To emphasize this, we will rewrite (1.5) as

$$h(d_1,\ldots,d_n) = C_1(d_1|\emptyset) \circ C_2(d_2|d_1) \circ \ldots \circ C_n(d_n|d_1,\ldots,d_{n-1}). \tag{1.6}$$

We now define h as *(weakly) separable* if

$$h(d_1,\ldots,d_n) = C_1(d_1) \circ C_2(d_1,d_2) \circ \ldots \circ C_n(d_1,\ldots,d_n). \tag{1.7}$$

(Strong separability is, of course, a special case of weak separability.) If h is (weakly) separable, we then have

$$\begin{aligned}
&\mathrm{opt}_{d_1\in D_1}\{\mathrm{opt}_{d_2\in D_2(d_1)}\{\ldots\{\mathrm{opt}_{d_n\in D_n(d_1,\ldots,d_{n-1})}\{h(d_1,\ldots,d_n)\}\}\ldots\}\}\\
&=\mathrm{opt}_{d_1\in D_1}\{\mathrm{opt}_{d_2\in D_2(d_1)}\{\ldots\{\mathrm{opt}_{d_n\in D_n(d_1,\ldots,d_{n-1})}\{C_1(d_1|\emptyset)\circ C_2(d_2|d_1)\circ\ldots\\
&\quad\ldots\circ C_n(d_n|d_1,\ldots,d_{n-1})\}\}\ldots\}\}\\
&=\mathrm{opt}_{d_1\in D_1}\{C_1(d_1|\emptyset)\circ\mathrm{opt}_{d_2\in D_2(d_1)}\{C_2(d_2|d_1)\circ\ldots\\
&\quad\ldots\circ\mathrm{opt}_{d_n\in D_n(d_1,\ldots,d_{n-1})}\{C_n(d_n|d_1,\ldots,d_{n-1})\}\ldots\}\}.
\end{aligned}\tag{1.8}$$

Let the function $f(d_1,\ldots,d_{i-1})$ be defined as the optimal solution of the sequential decision process where the decisions $d_1,\ldots,d_{i-1}$ have been made and the decisions $d_i,\ldots,d_n$ remain to be made; i.e.,

$$\begin{aligned}
f(d_1,\ldots,d_{i-1}) = \mathrm{opt}_{d_i}\{&\mathrm{opt}_{d_{i+1}}\{\ldots\{\mathrm{opt}_{d_n}\{C_i(d_i|d_1,\ldots,d_{i-1})\circ\\
&C_{i+1}(d_{i+1}|d_1,\ldots,d_i)\circ\ldots\circ C_n(d_n|d_1,\ldots,d_{n-1})\}\}\ldots\}\}.
\end{aligned}\tag{1.9}$$

Explicit mentions of the decision sets D_i are omitted here for convenience. We have then

$$\begin{aligned}
f(\emptyset) &= \mathrm{opt}_{d_1}\{\mathrm{opt}_{d_2}\{\ldots\{\mathrm{opt}_{d_n}\{C_1(d_1|\emptyset)\circ C_2(d_2|d_1)\circ\ldots\\
&\quad\ldots\circ C_n(d_n|d_1,\ldots,d_{n-1})\}\}\ldots\}\}\\
&=\mathrm{opt}_{d_1}\{C_1(d_1|\emptyset)\circ\mathrm{opt}_{d_2}\{C_2(d_2|d_1)\circ\ldots\\
&\quad\ldots\circ\mathrm{opt}_{d_n}\{C_n(d_n|d_1,\ldots,d_{n-1})\}\ldots\}\}\\
&=\mathrm{opt}_{d_1}\{C_1(d_1|\emptyset)\circ f(d_1)\}.
\end{aligned}\tag{1.10}$$

Generalizing, we conclude that

$$f(d_1,\ldots,d_{i-1}) = \mathrm{opt}_{d_i\in D_i(d_1,\ldots,d_{i-1})}\{C_i(d_i|d_1,\ldots,d_{i-1})\circ f(d_1,\ldots,d_i)\}. \tag{1.11}$$

Equation (1.11) is a recursive functional equation; we call it a *functional equation* since the unknown in the equation is a function f, and it is *recursive* since f is defined in terms of f (but having different arguments). It is the *dynamic programming functional equation* (DPFE) for the given optimization problem. In this book, we assume that we are given DPFEs that are properly formulated, i.e., that their solutions exist; we address only issues of how to obtain these solutions.

1.1.2 Dynamic Programming Functional Equations

The problem of solving the DPFE for $f(d_1, \ldots, d_{i-1})$ depends upon the subproblem of solving for $f(d_1, \ldots, d_i)$. If we define the state $S = (d_1, \ldots, d_{i-1})$ as the sequence of the first $i-1$ decisions, where $i = |S|+1 = |\{d_1, \ldots, d_{i-1}\}|+1$, we may rewrite the DPFE in the form

$$f(S) = \mathrm{opt}_{d_i \in D_i(S)}\{C_i(d_i|S) \circ f(S')\}, \tag{1.12}$$

where S is a state in a set $\mathcal{S}$ of possible states, $S' = (d_1, \ldots, d_i)$ is a next-state, and $\emptyset$ is the initial state. Since the DPFE is recursive, to terminate the recursion, its solution requires *base cases* (or "boundary" conditions), such as $f(S_0) = b$ when $S_0 \in \mathcal{S}_{base}$, where $\mathcal{S}_{base} \subset \mathcal{S}$. For a *base* (or *terminal*) state S_0, $f(S_0)$ is not evaluated using the DPFE, but instead has a given numerical constant b as its value; this value b may depend upon the base state S_0.

It should be noted that the sequence of decisions need not be limited to a fixed length n, but may be of indefinite length, terminating when a base case is reached. Different classes of DP problems may be characterized by how the states S, and hence the next-states S', are defined. It is often convenient to define the state S, not as the sequence of decisions made so far, with the next decision d chosen from $D(S)$, but rather as the set from which the next decision can be chosen, so that $D(S) =$ or $d \in S$. We then have a DPFE of the form

$$f(S) = \mathrm{opt}_{d \in S}\{C(d|S) \circ f(S')\}. \tag{1.13}$$

We shall later show that, for some problems, there may be multiple next-states, so that the DPFE has the form

$$f(S) = \mathrm{opt}_{d \in S}\{C(d|S) \circ f(S') \circ f(S'')\} \tag{1.14}$$

where S' and S'' are both next-states. A DPFE is said to be *r-th order* (or *nonserial* if $r > 1$) if there may be r next-states.

Simple serial DP formulations can be modeled by a state transition system or directed graph, where a state S corresponds to a node (or vertex) and a decision d that leads from state S to next-state S' is represented by a branch (or arc or edge) with label $C(d_i|S)$. $D(S)$ is the set of possible decisions when in state S, hence is associated with the successors of node S. More complex DP formulations require a more general graph model, such as that of a Petri net, which we discuss in Chap. 5.

Consider the directed graph whose nodes represent the states of the DPFE and whose branches represent possible transitions from states to next-states, each such transition reflecting a decision. The label of each branch, from S to S', denoted $b(S, S')$, is the cost $C(d|S)$ of the decision d, where $S' = T(S, d)$, where $T : \mathcal{S} \times D \rightarrow \mathcal{S}$ is a next-state transition or transformation function. The DPFE can then be rewritten in the form

$$f(S) = \text{opt}_{S'}\{b(S, S') + f(S')\}, \tag{1.15}$$

where $f(S)$ is the length of the shortest path from S to a terminal or *target* state S_0, and where each decision is to choose S' from among all (eligible) successors of S. (Different problems may have different eligibility constraints.) The base case is $f(S_0) = 0$.

For some problems, it is more convenient to use a DPFE of the "reverse" form

$$f'(S) = \text{opt}_{S'}\{f'(S') + b(S', S)\}, \tag{1.16}$$

where $f'(S)$ is the length of the shortest path from a designated state S_0 to S, and S' is a predecessor of S; S_0 is also known as the *source* state, and $f(S_0) = 0$ serves as the base case that terminates the recursion for this alternative DPFE. We call these *target-state* and *designated-source* DPFEs, respectively. We also say that, in the former case, we go "backward" from the target to the source, whereas, in the latter case, we go forward from the "source" to the target.

Different classes of DP formulations are distinguished by the nature of the decisions. Suppose each decision is a number chosen from a set $\{1, 2, \ldots, N\}$, and that each number must be chosen once and only once (so there are N decisions). Then if states correspond to possible permutations of the numbers, there are $O(N!)$ such states. Here we use the "big-O" notation ([10, 53]): we say $f(N)$ is $O(g(N))$ if, for a sufficiently large N, $f(N)$ is bounded by a constant multiple of $g(N)$. As another example, suppose each decision is a number chosen from a set $\{1, 2, \ldots, N\}$, but that not all numbers must be chosen (so there may be less than N decisions). Then if states correspond to subsets of the numbers, there are $O(2^N)$ such states. Fortuitously, there are many practical problems where a reduction in the number of relevant states is possible, such as when only the final decision d_{i-1} in a sequence $(d_1, \ldots, d_{i-1})$, together with the time or stage i at which the decision is made, is significant, so that there are $O(N^2)$ such states. We give numerous examples of the different classes in Chap. 2.

The solution of a DP problem generally involves more than only computing the value of $f(S)$ for the goal state S^*. We may also wish to determine the initial optimal decision, the optimal second decision that should be made in the next-state that results from the first decision, and so forth; that is, we may wish to determine the optimal sequence of decisions, also known as the optimal "policy" , by what is known as a reconstruction process. To reconstruct these optimal decisions, when evaluating $f(S) = \text{opt}_{d \in D(S)}\{C(d|S) \circ f(S')\}$ we may save the value of d, denoted d^*, that yields the optimal value of $f(S)$ at the time we compute this value, say, tabularly by entering the value $d^*(S)$ in a table for each S. The main alternative to using such a policy table is to reevaluate $f(S)$ as needed, as the sequence of next-states are determined; this is an example of a space versus time tradeoff.

1.1.3 The Elements of Dynamic Programming

The basic form of a dynamic programming functional equation is

$$f(S) = \text{opt}_{d \in D(S)} \{R(S, d) \circ f(T(S, d))\}, \tag{1.17}$$

where S is a *state* in some *state space* $\mathcal{S}$, d is a *decision* chosen from a *decision space* $D(S)$, $R(S, d)$ is a *reward function* (or *decision cost*, denoted $C(d|S)$ above), $T(S, d)$ is a next-state *transformation* (or *transition*) function, and $\circ$ is a binary operator. We will restrict ourselves to discrete DP, where the state space and decision space are both discrete sets. (Some problems with continuous states or decisions can be handled by discretization procedures, but we will not consider such problems in this book.) The elements of a DPFE have the following characteristics.

State The state S, in general, incorporates information about the sequence of decisions made so far. In some cases, the state may be the complete sequence, but in other cases only partial information is sufficient; for example, if the set of all states can be partitioned into equivalence classes, each represented by the last decision. In some simpler problems, the length of the sequence, also called the stage at which the next decision is to be made, suffices. The initial state, which reflects the situation in which no decision has yet been made, will be called the *goal state* and denoted S^*.

Decision Space The decision space $D(S)$ is the set of possible or "eligible" choices for the next decision d. It is a function of the state S in which the decision d is to be made. Constraints on possible next-state transformations from a state S can be imposed by suitably restricting $D(S)$. If $D(S) = \emptyset$, so that there are no eligible decisions in state S, then S is a terminal state.

Objective Function The objective function f, a function of S, is the optimal profit or cost resulting from making a sequence of decisions when in state S, i.e., after making the sequence of decisions associated with S. The goal of a DP problem is to find $f(S)$ for the goal state S^*.

Reward Function The reward function R, a function of S and d, is the profit or cost that can be attributed to the next decision d made in state S. The reward $R(S, d)$ must be separable from the profits or costs that are attributed to all other decisions. The value of the objective function for the goal state, $f(S^*)$, is the combination of the rewards for the complete optimal sequence of decisions starting from the goal state.

Transformation Function(s) The transformation (or transition) function T, a function of S and d, specifies the next-state that results from making a decision d in state S. As we shall later see, for nonserial DP problems, there may be more than one transformation function.

Operator The operator is a binary operation, usually addition or multiplication or minimization/maximization, that allows us to combine the returns of separate decisions. This operation must be associative if the returns of decisions are to be independent of the order in which they are made.

Base Condition Since the DPFE is recursive, base conditions must be specified to terminate the recursion. Thus, the DPFE applies for S in a state space $\mathcal{S}$, but

$$f(S_0) = b,$$

for S_0 in a set of base-states not in $\mathcal{S}$. Base-values b are frequently zero or infinity, the latter to reflect constraints. For some problems, setting $f(S_0) = \pm\infty$ is equivalent to imposing a constraint on decisions so as to disallow transitions to state S_0, or to indicate that $S_0 \notin \mathcal{S}$ is a state in which no decision is eligible.

To solve a problem using DP, we must define the foregoing elements to reflect the nature of the problem at hand. We give several examples below. We note first that some problems require certain generalizations. For example, some problems require a second-order DPFE having the form

$$f(S) = \mathrm{opt}_{d \in D(S)}\{R(S,d) \circ f(T_1(S,d)) \circ f(T_2(S,d))\}, \tag{1.18}$$

where T_1 and T_2 are both transformation functions to account for the situation in which more than one next-state can be entered, or

$$f(S) = \mathrm{opt}_{d \in D(S)}\{R(S,d) \circ p_1.f(T_1(S,d)) \circ p_2.f(T_2(S,d))\}, \tag{1.19}$$

where T_1 and T_2 are both transformation functions and p_1 and p_2 are multiplicative weights. In probabilistic DP problems, these weights are probabilities that reflect the probabilities associated with their respective state-transitions, only one of which can actually occur. In deterministic DP problems, these weights can serve other purposes, such as "discount factors" to reflect the time value of money.

1.1.4 Application: Linear Search

To illustrate the key concepts associated with DP that will prove useful in our later discussions, we examine a concrete example, the optimal "linear search" problem. This is the problem of permuting the data elements of an array A of size N, whose element x has probability p_x, so as to optimize the linear search process by minimizing the "cost" of a permutation, defined as the expected number of comparisons required. For example, let $A = \{a, b, c\}$ and $p_a = 0.2$, $p_b = 0.5$, and $p_c = 0.3$. There are six permutations, namely, $abc, acb, bac, bca, cab, cba$; the cost of the fourth permutation bca is 1.7, which can be calculated in several ways, such as

$$1p_b + 2p_c + 3p_a \text{ [using Method S]}$$

and

$$(p_a + p_b + p_c) + (p_a + p_c) + (p_a) \text{ [using Method W]}.$$

This optimal permutation problem can be regarded as a sequential decision process where three decisions must be made as to where the elements of A are to be placed in the final permuted array A'. The decisions are: which element is to be placed at the beginning of A', which element is to be placed in the middle of A', and which element is to be placed at the end of A'. The order in which these decisions are made does not necessarily matter, at least insofar as obtaining the correct answer is concerned; e.g., to obtain the permutation bca, our first decision may be to place element c in the middle of A'. Of course, some orderings of decisions may lead to greater efficiency than others. Moreover, the order in which decisions are made affects later choices; if c is chosen in the middle, it cannot be chosen again. That is, the decision set for any choice depends upon (is constrained by) earlier choices. In addition, the cost of each decision should be separable from other decisions. To obtain this separability, we must usually take into account the order in which decisions are made. For Method S, the cost of placing element x in the i-th location of A' equals ip_x regardless of when the decision is made. On the other hand, for Method W, the cost of a decision depends upon when the decision is made, more specifically upon its decision set. If the decisions are made in order from the beginning to the end of A', then the cost of deciding which member d_i of the respective decision set D_i to choose next equals $\sum_{x\in D_i} p_x$, the sum of the probabilities of the elements in $D_i = A - \{d_1, \ldots, d_{i-1}\}$. For example, let d_i denote the decision of which element of A to place in position i of A', and let D_i denote the corresponding decision set, where $d_i \in D_i$. If the decisions are made in the order $i = 1, 2, 3$ then $D_1 = A, D_2 = A - \{d_1\}, D_3 = A - \{d_1, d_2\}$. For Method S, if the objective function is written in the form $h(d_1, d_2, d_3) = 1p_{d_1} + 2p_{d_2} + 3p_{d_3}$, then

$$\begin{aligned} f(\emptyset) &= \min_{d_1\in A}\{\min_{d_2\in A-\{d_1\}}\{\min_{d_3\in A-\{d_1,d_2\}}\{1p_{d_1} + 2p_{d_2} + 3p_{d_3}\}\}\} \\ &= \min_{d_1\in A}\{1p_{d_1} + \min_{d_2\in A-\{d_1\}}\{2p_{d_2} + \min_{d_3\in A-\{d_1,d_2\}}\{3p_{d_3}\}\}\} \end{aligned} \quad (1.20)$$

For Method W, if the objective function is written in the form $h(d_1, d_2, d_3) = \sum_{x\in A} p_x + \sum_{x\in A-\{d_1\}} p_x + \sum_{x\in A-\{d_1,d_2\}} p_x$, then

$$\begin{aligned} &f(\emptyset) \\ &= \min_{d_1\in A}\{\min_{d_2\in A-\{d_1\}}\{\min_{d_3\in A-\{d_1,d_2\}}\{\sum_{x\in A} p_x + \sum_{x\in A-\{d_1\}} p_x + \sum_{x\in A-\{d_1,d_2\}} p_x\}\}\} \\ &= \min_{d_1\in A}\{\sum_{x\in A} p_x + \min_{d_2\in A-\{d_1\}}\{\sum_{x\in A-\{d_1\}} p_x + \min_{d_3\in A-\{d_1,d_2\}}\{\sum_{x\in A-\{d_1,d_2\}} p_x\}\}\}. \end{aligned} \quad (1.21)$$

However, if the decisions are made in reverse order $i = 3, 2, 1$, then $D_3 = A, D_2 = A - \{d_3\}, D_1 = A - \{d_2, d_3\}$, and the above must be revised accordingly. It should also be noted that if $h(d_1, d_2, d_3) = 0 + 0 + (1p_{d_1} + 2p_{d_2} + 3p_{d_3})$, where all of the cost is associated with the final decision d_3, then

$$f(\emptyset) = \min_{d_1 \in A}\{0 + \min_{d_2 \in A-\{d_1\}}\{0 + \min_{d_3 \in A-\{d_1,d_2\}}\{1p_{d_1} + 2p_{d_2} + 3p_{d_3}\}\}\}, \tag{1.22}$$

which is equivalent to enumeration. We conclude from this example that care must be taken in defining decisions and their interrelationships, and how to attribute separable costs to these decisions.

1.1.5 Problem Formulation and Solution

The optimal linear search problem of permuting the elements of an array A of size N, whose element x has probability p_x, can be solved using DP in the following fashion. We first define the state S as the set of data elements from which to choose. We then are to make a sequence of decisions as to which element of A should be placed next in the resulting array. We thus arrive at a DPFE of the form

$$f(S) = \min_{x \in S}\{C(x|S) + f(S - \{x\})\}, \tag{1.23}$$

where the reward or cost function $C(x|S)$ is suitably defined. Note that $S \in 2^A$, where 2^A denotes the power set of A. Our goal is to solve for $f(A)$ given the base case $f(\emptyset) = 0$. (This is a target-state formulation, where $\emptyset$ is the target state.)

This DPFE can also be written in the complementary form

$$f(S) = \min_{x \notin S}\{C(x|S) + f(S \cup \{x\})\}, \tag{1.24}$$

for $S \in 2^A$, where our goal is to solve for $f(\emptyset)$ given the base case $f(A) = 0$.

One definition of $C(x|S)$, based upon Method W, is as follows:

$$C_W(x|S) = \sum_{y \in S} p_y.$$

This function depends only on S, not on the decision x. A second definition, based upon Method S, is the following:

$$C_S(x|S) = (N + 1 - |S|)p_x.$$

This function depends on both S and x. These two definitions assume that the first decision is to choose the element to be placed first in the array. The solution of the problem is 1.7 for the optimal permutation *bca*. (Note: If we assume instead that the decisions are made in reverse, where the first decision chooses the element to be placed last in the array, the same DPFE applies but with $C'_S(x|S) = |S|p_x$; we will call this the *inverted* linear search problem. The optimal permutation is *acb* for this inverted problem.) If we order S by descending probability, it can be shown that the first element x^* in this ordering of S (that has maximum probability) minimizes the set $\{C(x|S) + f(S - x)\}$. Use of this "heuristic", also known as a *greedy* policy, makes performing the minimization operation of the DPFE unnecessary; instead, we need only find the maximum of a set of probabilities $\{p_x\}$. There are many optimization problems solvable by DP for which there are also greedy policies that reduce the amount of work necessary to obtain their solutions; we discuss this further in Sec. 1.1.14.

The inverted linear search problem is equivalent to a related problem associated with ordering the elements of a set A, whose elements have specified lengths or weights w (corresponding to their individual retrieval or processing times), such that the sum of the "sequential access" retrieval times is minimized. This optimal *permutation* problem is also known as the "tape storage" problem [22, pp.229–232], and is equivalent to the "shortest processing time" scheduling (SPT) problem. For example, suppose $A = \{a, b, c\}$ and $w_a = 2$, $w_b = 5$, and $w_c = 3$. If the elements are arranged in the order *acb*, it takes 2 units of time to sequentially retrieve a, 5 units of time to retrieve c (assuming a must be retrieved before retrieving c), and 10 units of time to retrieve b (assuming a and c must be retrieved before retrieving b). The problem of finding the optimal permutation can be solved using a DPFE of the form

$$f(S) = \min_{x \in S}\{|S|w_x + f(S - \{x\})\}, \tag{1.25}$$

as for the inverted linear search problem. $C(x|S) = |S|w_x$ since choosing x contributes a cost of w_x to each of the $|S|$ decisions that are to be made.

Example 1.1. Consider the linear search example where $A = \{a, b, c\}$ and $p_a = 0.2$, $p_b = 0.5$, and $p_c = 0.3$. The target-state DPFE (1.23) may be evaluated as follows:

$$\begin{aligned}
f(\{a,b,c\}) &= \min\{C(a|\{a,b,c\}) + f(\{b,c\}), C(b|\{a,b,c\}) + f(\{a,c\}),\\
&\qquad C(c|\{a,b,c\}) + f(\{a,b\})\}\\
f(\{b,c\}) &= \min\{C(b|\{b,c\}) + f(\{c\}), C(c|\{b,c\}) + f(\{b\})\}\\
f(\{a,c\}) &= \min\{C(a|\{a,c\}) + f(\{c\}), C(c|\{a,c\}) + f(\{a\})\}\\
f(\{a,b\}) &= \min\{C(a|\{a,b\}) + f(\{b\}), C(b|\{a,b\}) + f(\{a\})\}\\
f(\{c\}) &= \min\{C(c|\{c\}) + f(\emptyset)\}\\
f(\{b\}) &= \min\{C(b|\{b\}) + f(\emptyset)\}\\
f(\{a\}) &= \min\{C(a|\{a\}) + f(\emptyset)\}\\
f(\emptyset) &= 0
\end{aligned}$$

For Method W, these equations reduce to the following:

$$\begin{aligned}
f(\{a,b,c\}) &= \min\{C_W(a|\{a,b,c\}) + f(\{b,c\}), C_W(b|\{a,b,c\}) + f(\{a,c\}),\\
&\qquad C_W(c|\{a,b,c\}) + f(\{a,b\})\}\\
&= \min\{1.0 + f(\{b,c\}), 1.0 + f(\{a,c\}), 1.0 + f(\{a,b\})\}
\end{aligned}$$

$$\begin{aligned}
&= \min\{1.0 + 1.1, 1.0 + 0.7, 1.0 + 0.9\} = 1.7 \\
f(\{b,c\}) &= \min\{C_W(b|\{b,c\}) + f(\{c\}), C_W(c|\{b,c\}) + f(\{b\})\} \\
&= \min\{0.8 + f(\{c\}), 0.8 + f(\{b\})\} \\
&= \min\{0.8 + 0.3, 0.8 + 0.5\} = 1.1 \\
f(\{a,c\}) &= \min\{C_W(a|\{a,c\}) + f(\{c\}), C_W(c|\{a,c\}) + f(\{a\})\} \\
&= \min\{0.5 + f(\{c\}), 0.5 + f(\{a\})\} \\
&= \min\{0.5 + 0.3, 0.5 + 0.2\} = 0.7 \\
f(\{a,b\}) &= \min\{C_W(a|\{a,b\}) + f(\{b\}), C_W(b|\{a,b\}) + f(\{a\})\} \\
&= \min\{0.7 + f(\{b\}), 0.7 + f(\{a\})\} \\
&= \min\{0.7 + 0.5, 0.7 + 0.2\} = 0.9 \\
f(\{c\}) &= \min\{C_W(c|\{c\}) + f(\emptyset)\} = \min\{0.3 + f(\emptyset)\} = 0.3 \\
f(\{b\}) &= \min\{C_W(b|\{b\}) + f(\emptyset)\} = \min\{0.5 + f(\emptyset)\} = 0.5 \\
f(\{a\}) &= \min\{C_W(a|\{a\}) + f(\emptyset)\} = \min\{0.2 + f(\emptyset)\} = 0.2 \\
f(\emptyset) &= 0
\end{aligned}$$

For Method S, these equations reduce to the following:

$$\begin{aligned}
f(\{a,b,c\}) &= \min\{C_S(a|\{a,b,c\}) + f(\{b,c\}), C_S(b|\{a,b,c\}) + f(\{a,c\}), \\
&\qquad C_S(c|\{a,b,c\}) + f(\{a,b\})\} \\
&= \min\{1 \times 0.2 + f(\{b,c\}), 1 \times 0.5 + f(\{a,c\}), 1 \times 0.3 + f(\{a,b\})\} \\
&= \min\{0.2 + 1.9, 0.5 + 1.2, 0.3 + 1.6\} = 1.7 \\
f(\{b,c\}) &= \min\{C_S(b|\{b,c\}) + f(\{c\}), C_S(c|\{b,c\}) + f(\{b\})\} \\
&= \min\{2 \times 0.5 + f(\{c\}), 2 \times 0.3 + f(\{b\})\} \\
&= \min\{1.0 + 0.9, 0.6 + 1.5\} = 1.9 \\
f(\{a,c\}) &= \min\{C_S(a|\{a,c\}) + f(\{c\}), C_S(c|\{a,c\}) + f(\{a\})\} \\
&= \min\{2 \times 0.2 + f(\{c\}), 2 \times 0.3 + f(\{a\})\} \\
&= \min\{0.4 + 0.9, 0.6 + 0.6\} = 1.2 \\
f(\{a,b\}) &= \min\{C_S(a|\{a,b\}) + f(\{b\}), C_S(b|\{a,b\}) + f(\{a\})\} \\
&= \min\{2 \times 0.2 + f(\{b\}), 2 \times 0.5 + f(\{a\})\} \\
&= \min\{0.4 + 1.5, 1.0 + 0.6\} = 1.6 \\
f(\{c\}) &= \min\{C_S(c|\{c\}) + f(\emptyset)\} = \min\{3 \times 0.3 + f(\emptyset)\} = 0.9 \\
f(\{b\}) &= \min\{C_S(b|\{b\}) + f(\emptyset)\} = \min\{3 \times 0.5 + f(\emptyset)\} = 1.5 \\
f(\{a\}) &= \min\{C_S(a|\{a\}) + f(\emptyset)\} = \min\{3 \times 0.2 + f(\emptyset)\} = 0.6 \\
f(\emptyset) &= 0
\end{aligned}$$

It should be emphasized that the foregoing equations are to be *evaluated* in reverse of the order they have been presented, starting from the base case $f(\emptyset)$ and ending with the goal $f(\{a,b,c\})$. This evaluation is said to be "bottom-up". The goal cannot be evaluated first since it refers to values not available initially. While it may not be evaluated first, it is convenient to start at the goal to systematically *generate* the other equations, in a "top-down" fashion, and then sort the equations as necessary to evaluate them. We discuss such a generation process in Sect. 1.2.2. An alternative to generating a sequence of equations is to recursively evaluate the DPFE, starting at the goal, as described in Sect. 1.2.1.

As indicated earlier, we are not only interested in the final answer ($f(A) = 1.7$), but also in "reconstructing" the sequence of decisions that yields that answer. This is one reason that it is generally preferable to evaluate DPFEs nonrecursively. Of the three possible initial decisions, to choose a, b, or c first in goal-state $\{a, b, c\}$, the optimal decision is to choose b. Decision b yields the minimum of the set $\{2.1, 1.7, 1.9\}$, at a cost of 1.0 for Method W or at a cost of 0.5 for Method S, and causes a transition to state $\{a, c\}$. For Method W, the minimum value of $f(\{a, c\})$ is 0.7, obtained by choosing c at a cost of 0.5, which yields the minimum of the set $\{0.8, 0.7\}$, and which causes a transition to state $\{a\}$; the minimum value of $f(\{a\})$ is 0.2, obtained by necessarily choosing a at a cost of 0.2, which yields the minimum of the set $\{0.2\}$, and which causes a transition to base-state $\emptyset$. Thus, the optimal policy is to choose b, then c, and finally a, at a total cost of $1.0 + 0.5 + 0.2 = 1.7$. For Method S, the minimum value of $f(\{a, c\})$ is 1.2, obtained by choosing c at a cost of 0.6, which yields the minimum of the set $\{1.3, 1.2\}$, and which causes a transition to state $\{a\}$; the minimum value of $f(\{a\})$ is 0.6, obtained by necessarily choosing a at a cost of 0.6, which yields the minimum of the set $\{0.6\}$, and which causes a transition to base-state $\emptyset$. Thus, the optimal policy is to choose b, then c, and finally a, at a total cost of $0.5 + 0.6 + 0.6 = 1.7$.

1.1.6 State Transition Graph Model

Recall the directed graph model of a DPFE discussed earlier. For any state S, $f(S)$ is the length of the shortest path from S to the target state $\emptyset$. For Method W, the shortest path overall has length $1.0 + 0.5 + 0.2 = 1.7$; for Method S, the shortest path overall has length $0.5 + 0.6 + 0.6 = 1.7$. The foregoing calculations obtain the answer 1.7 by adding the branches in the order $(1.0 + (0.5 + (0.2)))$ or $(0.5 + (0.6 + (0.6)))$, respectively. The answer can also be obtained by adding the branches in the reverse order $(((1.0)+0.5)+0.2)$ or $(((0.5) + 0.6) + 0.6)$. With respect to the graph, this reversal is equivalent to using the designated-source DPFE (1.16), or equivalently

$$f'(S) = \min_{S'}\{f'(S') + C(x|S')\}, \tag{1.26}$$

where S' is a predecessor of S in that some decision x leads to a transition from S' to S, and where $f'(S)$ is the length of the shortest path from the source state S^* to any state S, with goal $f'(\emptyset)$ and base state $S^* = \{a, b, c\}$.

Example 1.2. For the linear search example, the designated-source DPFE (1.26) may be evaluated as follows:

$$\begin{aligned}
f'(\{a,b,c\}) &= 0\\
f'(\{b,c\}) &= \min\{f'(\{a,b,c\}) + C(a|\{a,b,c\})\}\\
f'(\{a,c\}) &= \min\{f'(\{a,b,c\}) + C(b|\{a,b,c\})\}\\
f'(\{a,b\}) &= \min\{f'(\{a,b,c\}) + C(c|\{a,b,c\})\}\\
f'(\{c\}) &= \min\{f'(\{b,c\}) + C(b|\{b,c\}), f'(\{a,c\}) + C(a|\{a,c\})\}\\
f'(\{b\}) &= \min\{f'(\{b,c\}) + C(c|\{b,c\}), f'(\{a,b\}) + C(a|\{a,b\})\}\\
f'(\{a\}) &= \min\{f'(\{a,c\}) + C(c|\{a,c\}), f'(\{a,b\}) + C(b|\{a,b\})\}\\
f'(\emptyset) &= \min\{f'(\{a\}) + C(a|\{a\}), f'(\{b\}) + C(b|\{b\}),\\
&\qquad f'(\{c\}) + C(c|\{c\})\}
\end{aligned}$$

For Method W, these equations reduce to the following:

$$\begin{aligned}
f'(\{a,b,c\}) &= 0\\
f'(\{b,c\}) &= \min\{f'(\{a,b,c\}) + C_W(a|\{a,b,c\})\} = \min\{0 + 1.0\} = 1.0\\
f'(\{a,c\}) &= \min\{f'(\{a,b,c\}) + C_W(b|\{a,b,c\})\} = \min\{0 + 1.0\} = 1.0\\
f'(\{a,b\}) &= \min\{f'(\{a,b,c\}) + C_W(c|\{a,b,c\})\} = \min\{0 + 1.0\} = 1.0\\
f'(\{c\}) &= \min\{f'(\{b,c\}) + C_W(b|\{b,c\}), f'(\{a,c\}) + C_W(a|\{a,c\})\}\\
&= \min\{1.0 + 0.8, 1.0 + 0.5\} = 1.5\\
f'(\{b\}) &= \min\{f'(\{b,c\}) + C_W(c|\{b,c\}), f'(\{a,b\}) + C_W(a|\{a,b\})\}\\
&= \min\{1.0 + 0.8, 1.0 + 0.7\} = 1.7\\
f'(\{a\}) &= \min\{f'(\{a,c\}) + C_W(c|\{a,c\}), f'(\{a,b\}) + C_W(b|\{a,b\})\}\\
&= \min\{1.0 + 0.5, 1.0 + 0.7\} = 1.5\\
f'(\emptyset) &= \min\{f'(\{a\}) + C_W(a|\{a\}), f'(\{b\}) + C_W(b|\{b\}),\\
&\qquad f'(\{c\}) + C_W(c|\{c\})\}\\
&= \min\{1.5 + 0.2, 1.7 + 0.5, 1.5 + 0.3\} = 1.7
\end{aligned}$$

For Method S, these equations reduce to the following:

$$\begin{aligned}
f'(\{a,b,c\}) &= 0\\
f'(\{b,c\}) &= \min\{f'(\{a,b,c\}) + C_S(a|\{a,b,c\})\} = \min\{0 + 0.2\} = 0.2\\
f'(\{a,c\}) &= \min\{f'(\{a,b,c\}) + C_S(b|\{a,b,c\})\} = \min\{0 + 0.5\} = 0.5\\
f'(\{a,b\}) &= \min\{f'(\{a,b,c\}) + C_S(c|\{a,b,c\})\} = \min\{0 + 0.3\} = 0.3\\
f'(\{c\}) &= \min\{f'(\{b,c\}) + C_S(b|\{b,c\}), f'(\{a,c\}) + C_S(a|\{a,c\})\}\\
&= \min\{0.2 + 1.0, 0.5 + 0.4\} = 0.9\\
f'(\{b\}) &= \min\{f'(\{b,c\}) + C_S(c|\{b,c\}), f'(\{a,b\}) + C_S(a|\{a,b\})\}\\
&= \min\{0.2 + 0.6, 0.3 + 0.4\} = 0.7\\
f'(\{a\}) &= \min\{f'(\{a,c\}) + C_S(c|\{a,c\}), f'(\{a,b\}) + C_S(b|\{a,b\})\}\\
&= \min\{0.5 + 0.6, 0.3 + 1.0\} = 1.1\\
f'(\emptyset) &= \min\{f'(\{a\}) + C_S(a|\{a\}), f'(\{b\}) + C_S(b|\{b\}),
\end{aligned}$$

$$f'(\{c\}) + C_S(c|\{c\})\}$$
$$= \min\{1.1 + 0.6, 0.7 + 1.5, 0.9 + 0.9\} = 1.7$$

Here, we listed the equations in order of their (bottom-up) evaluation, with the base case $f'(\{a, b, c\})$ first and the goal $f'(\emptyset)$ last.

1.1.7 Staged Decisions

It is often convenient and sometimes necessary to incorporate stage numbers as a part of the definition of the state. For example, in the linear search problem there are N distinct decisions that must be made, and they are assumed to be made in a specified order. We assume that N, also called the *horizon*, is finite and known. The first decision, made at stage 1, is to decide which data item should be placed first in the array, the second decision, made at stage 2, is to decide which data item should be placed second in the array, etc. Thus, we may rewrite the original DPFE (1.23) as

$$f(k, S) = \min_{x \in S}\{C(x|k, S) + f(k + 1, S - \{x\})\}, \tag{1.27}$$

where the state now consists of a stage number k and a set S of items from which to choose. The goal is to find $f(1, A)$ with base condition $f(N+1, \emptyset) = 0$. Suppose we again define $C(x|k, S) = (N + 1 - |S|)p_x$. Since $k = N + 1 - |S|$, we have $C(x|k, S) = kp_x$. This cost function depends on the stage k and the decision x, but is independent of S.

For the inverted linear search (or optimal permutation) problem, where the first decision, made at stage 1, is to decide which data item should be placed last in the array, the second decision, made at stage 2, is to decide which data item should be placed next-to-last in the array, etc., the staged DPFE is the same as (1.27), but where $C(x|k, S) = kw_x$, which is also independent of S. While this simplification is only a modest one, it can be very significant for more complicated problems.

Incorporating stage numbers as part of the definition of the state may also be beneficial in defining base-state conditions. We may use the base condition $f(k, S) = 0$ when $k > N$ (for any S); the condition $S = \emptyset$ can be ignored. It is far easier to test whether the stage number exceeds some limit ($k > N$) than whether a set equals some base value ($S = \emptyset$). Computationally, this involves a comparison of integers rather than a comparison of sets.

Stage numbers may also be regarded as transition times, and DPFEs incorporating them are also called *fixed-time models.* Stage numbers need not be consecutive integers. We may define the stage or *virtual time* k to be some number that is associated with the k-th decision, where k is a sequence counter. For example, adding consecutive stage numbers to the DPFE (1.25) for the (inverted) linear search problem, we have

$$f(k, S) = \min_{x \in S}\{|S|w_x + f(k + 1, S - \{x\})\}, \tag{1.28}$$

where the goal is to find $f(1, A)$ with base-condition $f(k, S) = 0$ when $k > N$. We have $C(x|S) = |S|w_x$ since choosing x contributes a length of w_x to each of the $|S|$ decisions that are to be made. Suppose we define the *virtual time* or stage k as the "length-so-far" when the next decision is to be made. Then

$$f(k, S) = \min_{x \in S}\{(k + w_x) + f(k + w_x, S - \{x\})\}, \tag{1.29}$$

where the goal is to find $f(0, A)$ with base-condition $f(k, S) = 0$ when $k = \sum_{x \in A} w_x$ or $S = \emptyset$. The cost of a decision x in state (k, S), that is $C(x|k, S) = (k + w_x)$, is the length-so-far k plus the retrieval time w_x for the chosen item x, and in the next-state resulting from this decision the virtual time or stage k is also increased by w_x.

Example 1.3. For the linear search problem, the foregoing staged DPFE (1.28) may be evaluated as follows:

$$\begin{aligned}
f(1, \{a, b, c\}) &= \min\{C(a|1, \{a, b, c\}) + f(2, \{b, c\}),\\
&\qquad C(b|1, \{a, b, c\}) + f(2, \{a, c\}), C(c|1, \{a, b, c\}) + f(2, \{a, b\})\}\\
&= \min\{6 + 11, 15 + 7, 9 + 9\} = 17\\
f(2, \{b, c\}) &= \min\{C(b|2, \{b, c\}) + f(3, \{c\}), C(c|2, \{b, c\}) + f(3, \{b\})\}\\
&= \min\{10 + 3, 6 + 5\} = 11\\
f(2, \{a, c\}) &= \min\{C(a|2, \{a, c\}) + f(3, \{c\}), C(c|2, \{a, c\}) + f(3, \{a\})\}\\
&= \min\{4 + 3, 6 + 2\} = 7\\
f(2, \{a, b\}) &= \min\{C(a|2, \{a, b\}) + f(3, \{b\}), C(b|2, \{a, b\}) + f(3, \{a\})\}\\
&= \min\{4 + 5, 10 + 2\} = 9\\
f(3, \{c\}) &= \min\{C(c|3, \{c\}) + f(4, \emptyset)\} = \min\{3 + 0\} = 3\\
f(3, \{b\}) &= \min\{C(b|3, \{b\}) + f(4, \emptyset)\} = \min\{5 + 0\} = 5\\
f(3, \{a\}) &= \min\{C(a|3, \{a\}) + f(4, \emptyset)\} = \min\{2 + 0\} = 2\\
f(4, \emptyset) &= 0
\end{aligned}$$

Example 1.4. In contrast, the foregoing virtual-stage DPFE (1.29) may be evaluated as follows:

$$\begin{aligned}
f(0, \{a, b, c\}) &= \min\{(0 + 2) + f((0 + 2), \{b, c\}), (0 + 5) + f((0 + 5), \{a, c\}),\\
&\qquad (0 + 3) + f((0 + 3), \{a, b\})\}\\
&= \min\{2 + 15, 5 + 17, 3 + 15\} = 17\\
f(2, \{b, c\}) &= \min\{(2 + 5) + f((2 + 5), \{c\}), (2 + 3) + f((2 + 3), \{b\})\}\\
&= \min\{7 + 10, 5 + 10\} = 15\\
f(5, \{a, c\}) &= \min\{(5 + 2) + f((5 + 2), \{c\}), (5 + 3) + f((5 + 3), \{a\})\}\\
&= \min\{7 + 10, 8 + 10\} = 17\\
f(3, \{a, b\}) &= \min\{(3 + 2) + f((3 + 2), \{b\}), (3 + 5) + f((3 + 5), \{a\})\}
\end{aligned}$$

$$\begin{aligned}
&= \min\{5+10, 8+10\} = 15\\
f(7,\{c\}) &= \min\{(7+3) + f((7+3),\emptyset)\} = 10+0 = 10\\
f(5,\{b\}) &= \min\{(5+5) + f((5+5),\emptyset)\} = 10+0 = 10\\
f(8,\{a\}) &= \min\{(8+2) + f((8+2),\emptyset)\} = 10+0 = 10\\
f(10,\emptyset) &= 0
\end{aligned}$$

1.1.8 Path-States

In a graph representation of a DPFE, we may let state S be defined as the ordered sequence of decisions $(d_1, \ldots, d_{i-1})$ made so far, and represent it by a node in the graph. Then each state S is associated with a path in this graph from the initial (goal) state $\emptyset$ to state S. The applicable path-state DPFE, which is of the form (1.24), is

$$f(S) = \min_{x \notin S}\{C(x|S) + f(S \cup \{x\})\}. \qquad (1.30)$$

The goal is to solve for $f(\emptyset)$ given the base cases $f(S_0) = 0$, where each $S_0 \in \mathcal{S}_{base}$ is a terminal state in which no decision remains to be made.

Example 1.5. For the linear search example, the foregoing DPFE (1.30) may be evaluated as follows:

$$\begin{aligned}
f(\emptyset) &= \min\{C(a|\emptyset) + f(a), C(b|\emptyset) + f(b), C(c|\emptyset) + f(c)\}\\
f(a) &= \min\{C(b|a) + f(ab), C(c|a) + f(ac)\}\\
f(b) &= \min\{C(a|b) + f(ba), C(c|b) + f(bc)\}\\
f(c) &= \min\{C(a|c) + f(ca), C(b|c) + f(cb)\}\\
f(ab) &= \min\{C(c|ab) + f(abc)\}\\
f(ac) &= \min\{C(b|ac) + f(acb)\}\\
f(ba) &= \min\{C(c|ba) + f(bac)\}\\
f(bc) &= \min\{C(a|bc) + f(bca)\}\\
f(ca) &= \min\{C(b|ca) + f(cab)\}\\
f(cb) &= \min\{C(a|cb) + f(cba)\}\\
f(abc) &= f(acb) = f(bac) = f(bca) = f(cab) = f(cba) = 0
\end{aligned}$$

where $C(x|S)$ may be either the weak or strong versions. There are $N!$ individual bases cases, each corresponding to a permutation. However, the base-cases are equivalent to the single condition that $f(S) = 0$ when $|S| = N$.

For this problem, the information regarding the ordering of the decisions incorporated in the definition of the state is not necessary; we need only know the members of the decision sequence S so that the next decision d will be a different one (i.e., so that $d \notin S$). If the state is considered unordered, the complexity of the problem decreases from $O(N!)$ for permutations to $O(2^N)$ for subsets. For some problems, the state must *also* specify the most recent decision if it affects the choice or cost of the next decision. In other problems, the state need specify *only* the most recent decision.

We finally note that the equations of Example 1.5 can also be used to obtain the solution to the problem if we assume that $C(x|S) = 0$ (as would be the case when we cannot determine separable costs) and consequently the base cases must be defined by enumeration (instead of being set to zero), namely, $f(abc) = 2.1$, $f(acb) = 2.3$, $f(bac) = 1.8$, $f(bca) = 1.7$, $f(cab) = 2.2$, and $f(cba) = 1.9$.

1.1.9 Relaxation

The term *relaxation* is used in mathematics to refer to certain iterative methods of solving a problem by successively obtaining better approximations x_i to the solution x^*. (Examples of relaxation methods are the Gauss-Seidel method for solving systems of linear equations, and gradient-based methods for finding the minimum or maximum of a continuous function of n variables.)

In the context of discrete optimization problems, we observe that the minimum of a finite set $x^* = \min\{a_1, a_2, \ldots, a_N\}$ can be evaluated by a sequence of pairwise minimization operations

$$x^* = \min\{\min\{\ldots\{\min\{a_1, a_2\}, a_3\}, \ldots\}, a_N\}.$$

The sequence of partial minima, $x_1 = a_1$, $x_2 = \min\{x_1, a_2\}$, $x_3 = \min\{x_2, a_3\}$, $x_4 = \min\{x_3, a_4\}, \ldots$, is the solution of the recurrence relation $x_i = \min\{x_{i-1}, a_i\}$, for $i > 1$, with initial condition $x_1 = a_1$. (Note that $\min\{x_1, a_2\}$ will be called the "innermost min".) Instead of letting $x_1 = a_1$, we may let $x_1 = \min\{x_0, a_1\}$, where $x_0 = \infty$. We may regard the sequence $x_1, x_2, x_3, \ldots$ as "successive approximations" to the final answer x^*. Alternatively, the recursive equation $x = \min\{x, a_i\}$ can be solved using a successive approximations process that sets a "minimum-so-far" variable x to the minimum of its current value and some next value a_i, where x is initially ∞. Borrowing the terminology used for infinite sequences, we say the finite sequence x_i, or the "minimum-so-far" variable x, "converges" to x^*. {In this book, we restrict ourselves to finite processes for which N is fixed, so "asymptotic" convergence issues do not arise.} We will also borrow the term *relaxation* to characterize such successive approximations techniques.

One way in which the relaxation idea can be applied to the solution of dynamic programming problems is in evaluating the minimization operation of the DPFE

$$\begin{aligned} f(S) &= \min_{x \in S}\{C(x|S) + f(S'_x)\} \\ &= \min\{C(x_1|S) + f(S'_{x_1}), \\ &\qquad C(x_2|S) + f(S'_{x_2}), \ldots, \\ &\qquad C(x_m|S) + f(S'_{x_m})\}, \end{aligned} \tag{1.31}$$

where $S = \{x_1, x_2, \dots x_m\}$ and S'_x is the next-state resulting from choosing x in state S. Rather than computing all the values $C(x|S) + f(S'_x)$, for each $x \in S$, before evaluating the minimum of the set, we may instead compute $f(S)$ by successive approximations as follows:

$$\begin{aligned} f(S) = \min\{\min\{\dots\{\min\{&C(x_1|S) + f(S'_{x_1}), \\ &C(x_2|S) + f(S'_{x_2})\}, \dots\}, \\ &C(x_m|S) + f(S'_{x_m})\}. \end{aligned} \tag{1.32}$$

In using this equation to compute $f(S)$, the values of $f(S'_{x_i})$, as encountered in proceeding in a left-to-right (inner-to-outer) order, should all have been previously computed. To achieve this objective, it is common to order (topologically) the values of S for which $f(S)$ is to be computed, as in Example 1.6 of Sect. 1.1.10. An alternative is to use a staged formulation.

Consider the staged "fixed-time" DPFE of the form

$$f(k, S) = \min_x \{C(x|k, S) + f(k-1, S'_x)\}, \tag{1.33}$$

which, for each S, defines a sequence $f(0, S), f(1, S), f(2, S), f(3, S), \dots$ of successive approximations to $f(S)$. The minimum member of the sequence is the desired answer, i.e., $f(S) = \min_k\{f(k, S)\}$. {Here, we adopt the Java "overloading" convention that f with one argument differs from f with two arguments.} Note that $f(k, S)$ is a function not of $f(k-1, S)$, but of $f(k-1, S'_x)$, where S'_x is the next-state; e.g., $f(1, S)$ depends not on $f(0, S)$ but on $f(0, S')$. Since the sequence of values $f(k, S)$ is not necessarily monotonic, we define a new sequence $F(k, S)$ by the "relaxation" DPFE

$$F(k, S) = \min\{F(k-1, S), \min_x\{C(x|k, S) + F(k-1, S'_x)\}\}. \tag{1.34}$$

In Example 1.9 of Sect. 1.1.10, we will see that this new sequence $F(0, S)$, $F(1, S), F(2, S), F(3, S), \dots$ is monotonic, and converges to $f(S)$.

1.1.10 Shortest Path Problems

In the solution to the linear search problem we gave earlier, we used a state transition graph model and noted that solving the linear search problem was equivalent to finding the shortest path in a graph. There are a myriad of other problems that can be formulated and solved as graph optimization problems, so such problems are of special importance. Some of the problems are more complex, however, such as when the graph is cyclic.

For acyclic graphs, the shortest path from a source node s to a target node t can be found using a DPFE of a (target-state) form similar to (1.15):

$$f(p) = \min_q\{b(p, q) + f(q)\}, \tag{1.35}$$

where $b(p,q)$ is the distance from p to q, and $f(p)$ is the length of the shortest path from node p to node t. We may either restrict node $q \in \text{succ}(p)$ to be a successor of node p, or let $b(p,q) = \infty$ if $q \notin \text{succ}(p)$. {For acyclic graphs, we may also assume $b(p,p) = \infty$ for all p.} Our goal is to find $f(s)$ with base condition $f(t) = 0$. In this formulation, the state p is defined as the node in the graph at which we make a decision to go some next node q before continuing ultimately to the designated target.

Example 1.6 (SPA). As a numerical example, consider the graph in Fig. 1.1 with nodes $\{s,x,y,t\}$ and branches $\{(s,x),(s,y),(x,y),(x,t),(y,t)\}$ with branch distances $\{3,5,1,8,5\}$, respectively. For illustrative purposes, we add a "dummy" branch (s,t) having distance $b(s,t) = \infty$.

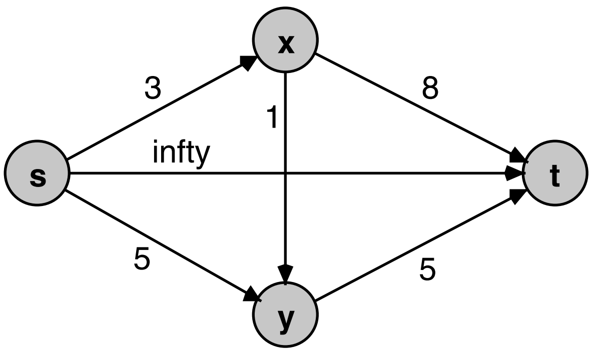

Fig. 1.1. Shortest Path Problem in an Acyclic Graph

The DPFE for the shortest path from s to t yields the following equations:

$$\begin{aligned}
f(s) &= \min\{b(s,x)+f(x), b(s,y)+f(y), b(s,t)+f(t)\},\\
f(x) &= \min\{b(x,y)+f(y), b(x,t)+f(t)\},\\
f(y) &= \min\{b(y,t)+f(t)\},\\
f(t) &= 0.
\end{aligned}$$

Consider evaluating the minimizing value of $f(s)$ by relaxation, i.e.,

$$\begin{aligned}
f(s) &= \min\{b(s,x)+f(x), b(s,y)+f(y), b(s,t)+f(t)\},\\
&= \min\{\min\{b(s,x)+f(x), b(s,y)+f(y)\}, b(s,t)+f(t)\}.
\end{aligned}$$

This is of the same form as (1.32). $f(x)$ and $f(y)$ in the innermost min should be evaluated before $f(t)$, but in fact both $f(x)$ and $f(y)$ depend upon $f(t)$. Thus, $f(t)$ should be evaluated first.

Substituting the above given branch distances into the foregoing equations, we have

$$\begin{aligned} f(s) &= \min\{3 + f(x), 5 + f(y), \infty + f(t)\}, \\ f(x) &= \min\{1 + f(y), 8 + f(t)\}, \\ f(y) &= \min\{5 + f(t)\}, \\ f(t) &= 0. \end{aligned}$$

If these equations are evaluated in "bottom-up" order, then we have $f(t) = 0$, $f(y) = 5$, $f(x) = 6$, and $f(s) = 9$.

If the graph is acyclic, then the graph can be topologically sorted and the DPFE can be evaluated for p in this order such that evaluation of $f(p)$ will always be in terms of previously calculated values of $f(q)$. On the other hand, if the graph is cyclic, so that for example p and q are successors of each other, then $f(p)$ may be defined in terms of $f(q)$, and $f(q)$ may be defined in terms of $f(p)$. This circular definition presents difficulties that require special handling.

For convenience, we will assume that cyclic graphs do not contain *self-loops*, i.e., branches from a node p to itself. For a graph having such a branch, if $b(p, p)$ is positive, that branch cannot be in the shortest path since its omission would lead to a shorter path, hence we may omit the branch. On the other hand, if $b(p, p)$ is negative, the problem is not well-defined since there is no shortest path at all. If $b(p, p) = 0$, the problem is also not well-defined since a shortest path may have an infinite number of branches, hence we may also omit the branch.

One way to handle cyclic graphs (having no self-loops), where $f(q)$ may depend on $f(p)$, is to use relaxation to solve the DPFE

$$f(p) = \min_q \{b(p, q) + f(q)\}, \tag{1.36}$$

where $f(p) = \infty$ for $p \neq t$ initially, and $f(t) = 0$.

Example 1.7 (SPC—successive approximations). To the preceding example, suppose we add a branch (y, x) with distance $b(y, x) = 2$ (see Fig. 1.2).

The graph is then cyclic, and the equations obtained from the DPFE are as follows:

$$\begin{aligned} f(s) &= \min\{b(s, x) + f(x), b(s, y) + f(y), b(s, t) + f(t) \\ &= \min\{3 + f(x), 5 + f(y), \infty + f(t)\}, \\ f(x) &= \min\{b(x, y) + f(y), b(x, t) + f(t)\} = \min\{1 + f(y), 8 + f(t)\}, \\ f(y) &= \min\{b(y, x) + f(x), b(y, t) + f(t)\} = \min\{2 + f(x), 5 + f(t)\}, \\ f(t) &= 0. \end{aligned}$$

We note that $f(x)$ depends on $f(y)$, and $f(y)$ depends on $f(x)$.

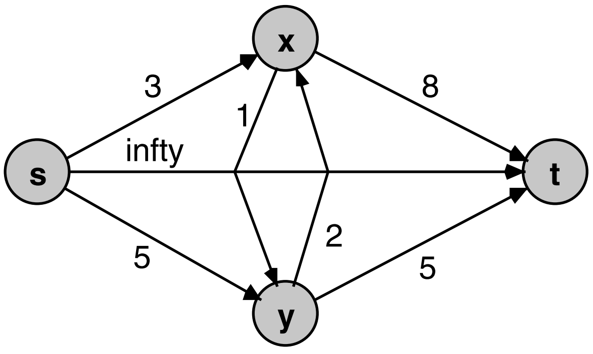

Fig. 1.2. Shortest Path Problem in an Cyclic Graph

Solving these equations by relaxation, assuming $f(s) = f(x) = f(y) = \infty$ and $f(t) = 0$ initially, we have, as the first successive approximation

$$\begin{aligned} f(s) &= \min\{3+\infty, 5+\infty, \infty+0\} = \infty, \\ f(x) &= \min\{1+\infty, 8+0\} = 8, \\ f(y) &= \min\{2+\infty, 5+0\} = 5, \\ f(t) &= 0, \end{aligned}$$

as the second successive approximation

$$\begin{aligned} f(s) &= \min\{3+8, 5+5, \infty+0\} = 10, \\ f(x) &= \min\{1+5, 8+0\} = 6, \\ f(y) &= \min\{2+8, 5+0\} = 5, \\ f(t) &= 0, \end{aligned}$$

and as the third successive approximation

$$\begin{aligned} f(s) &= \min\{3+6, 5+5, \infty+0\} = 9, \\ f(x) &= \min\{1+5, 8+0\} = 6, \\ f(y) &= \min\{2+6, 5+0\} = 5, \\ f(t) &= 0. \end{aligned}$$

Continuing this process to compute additional successive approximations will result in no changes. In this event, we say that the relaxation process has converged to the final solution, which for this example is $f(s) = 9$ (and $f(x) = 6$, $f(y) = 5$, $f(t) = 0$).

Another way to handle cyclic graphs is to introduce staged decisions, using a DPFE of the same form as Eq. (1.33).

$$f(k,p) = \min_q \{b(p,q) + f(k-1,q)\}, \tag{1.37}$$

where $f(k,p)$ is the length of the shortest path from p to t having *exactly* k branches. k, the number of branches in a path, ranges from 0 to $N-1$, where N is the number of nodes in the graph. We may disregard paths having N or more branches that are necessarily cyclic hence cannot be shortest (assuming the graph has no negative or zero-length cycle, since otherwise the shortest path problem is not well-defined). The base conditions are $f(0,t) = 0$, $f(0,p) = \infty$ for $p \neq t$, and $f(k,t) = \infty$ for $k > 0$. We must evaluate $\min_k\{f(k,s)\}$ for $k = 0,\ldots,N-1$.

Example 1.8 (SPC-fixed time). For the same example as above, the staged fixed-time DPFE (1.37) can be solved as follows:

$$\begin{aligned}
&f(k,s) = \min\{b(s,x) + f(k-1,x), b(s,y) + f(k-1,y), b(s,t) + f(k-1,t)\},\\
&f(k,x) = \min\{b(x,y) + f(k-1,y), b(x,t) + f(k-1,t)\},\\
&f(k,y) = \min\{b(y,x) + f(k-1,x), b(y,t) + f(k-1,t)\},\\
&f(k,t) = 0.
\end{aligned}$$

Its solution differs from the foregoing relaxation solution in that $f(0,t) = 0$ initially, but $f(k,t) = \infty$ for $k > 0$. In this case, we have, as the first successive approximation

$$\begin{aligned}
f(1,s) &= \min\{3 + f(0,x), 5 + f(0,y), \infty + f(0,t)\} = \infty,\\
f(1,x) &= \min\{1 + f(0,y), 8 + f(0,t)\} = 8,\\
f(1,y) &= \min\{2 + f(0,x), 5 + f(0,t)\} = 5,\\
f(1,t) &= \infty,
\end{aligned}$$

as the second successive approximation

$$\begin{aligned}
f(2,s) &= \min\{3 + f(1,x), 5 + f(1,y), \infty + f(1,t)\} = 10,\\
f(2,x) &= \min\{1 + f(1,y), 8 + f(1,t)\} = 6,\\
f(2,y) &= \min\{2 + f(1,x), 5 + f(1,t)\} = 10,\\
f(2,t) &= \infty,
\end{aligned}$$

and as the third and final successive approximation

$$\begin{aligned}
f(3,s) &= \min\{3 + f(2,x), 5 + f(2,y), \infty + f(2,t)\} = 9,\\
f(3,x) &= \min\{1 + f(2,y), 8 + f(2,t)\} = 11,\\
f(3,y) &= \min\{2 + f(2,x), 5 + f(2,t)\} = 8,\\
f(3,t) &= \infty.
\end{aligned}$$

Unlike the situation in the preceding example, the values of $f(k,p)$ do not converge. Instead, $f(p) = \min_k\{f(k,p)\}$, i.e.,

$$\begin{aligned}
f(s) &= \min\{\infty,\infty,10,9\} = 9,\\
f(x) &= \min\{\infty,8,6,11\} = 6,\\
f(y) &= \min\{\infty,5,10,8\} = 5,\\
f(t) &= \min\{0,\infty,\infty,\infty\} = 0.
\end{aligned}$$

We emphasize that the matrix $f(k,p)$ must be evaluated rowwise (varying p for a fixed k) rather than columnwise.

Example 1.9 (SPC-relaxation). Suppose we define $F(k,p)$ as the length of the shortest path from p to t having k *or fewer* branches. F satisfies a DPFE of the same form as f,

$$F(k,p) = \min_q \{b(p,q) + F(k-1,q)\}. \tag{1.38}$$

The goal is to compute $F(N-1,s)$. The base conditions are $F(0,t) = 0$, $F(0,p) = \infty$ for $p \neq t$, but where $F(k,t) = 0$ for $k > 0$. In this formulation, the sequence of values $F(k,p)$, for $k = 0,\ldots,N-1$, are successive approximations to the length of the shortest path from p to t having at most $N-1$ branches, hence the goal can be found by finding $F(k,s)$, for $k = 0,\ldots,N-1$. We observe that the values of $F(k,p)$ are equal to the values obtained as the k-th successive approximation for $f(p)$ in Example 1.7.

For a fixed k, if p has m successors $q_1,\ldots,q_m$, then

$$\begin{aligned}
F(k,p) &= \min_q \{b(p,q) + F(k-1,q)\}\\
&= \min\{b(p,q_1) + F(k-1,q_1), b(p,q_2) + F(k-1,q_2),\ldots,\\
&\quad b(p,q_m) + F(k-1,q_m)\},
\end{aligned}$$

which can be evaluated by relaxation (as discussed in Sect. 1.1.9) by computing

$$\begin{aligned}
F(k,p) = \min\{&\min\{\ldots\{\min\{b(p,q_1) + F(k-1,q_1), b(p,q_2) + F(k-1,q_2)\},\\
&\ldots\}, b(p,q_m) + F(k-1,q_m)\}.
\end{aligned} \tag{1.39}$$

For each p, $F(k,p)$ is updated once for each successor $q_i \in \text{succ}(p)$ of p, assuming $F(k-1,q)$ has previously been evaluated for all q. We emphasize that computations are staged, for $k = 1,\ldots,N-1$.

It should be noted that, given k, in solving (1.39) for $F(k,p)$ as a function of $\{F(k-1,q) | q \in \text{succ}(p)\}$, if $F(k,q)$ has been previously calculated, then it may be used instead of $F(k-1,q)$. This variation is the basis of the Bellman-Ford algorithm [10, 13], for which the sequence $F(k,p)$, for $k = 0,\ldots,N-1$, may converge more rapidly to the desired solution $F(N-1,s)$.

We also may use the path-state approach to find the shortest path in a cyclic graph, using a DPFE of the form

$$f(p_1, \ldots, p_i) = \min_{q \notin \{p_1, \ldots, p_i\}} \{b(p_i, q) + f(p_1, \ldots, p_i, q)\}, \qquad (1.40)$$

where the state is the sequence of nodes $p_1, \ldots, p_i$ in a path S, and q is a successor of p_i that does not appear earlier in the path S. The next-state S' appends q to the path S. The goal is $f(s)$ (where $i = 1$) and the base condition is $f(p_1, \ldots, p_i) = 0$ if $p_i = t$. In Chap. 2, we show how this approach can be used to find longest simple (acyclic) paths (LSP) and to find shortest Hamiltonian paths, the latter also known as the "traveling salesman" problem (TSP).

In the foregoing, we made no assumption on branch distances. If we restrict branch distances to be positive, the shortest path problem can be solved more efficiently using some variations of dynamic programming, such as Dijkstra's algorithm (see [10, 59]). If we allow negative branch distances, but not negative cycles (otherwise the shortest path may be infinitely negative or undefined), Dijkstra's algorithm may no longer find the shortest path. However, other dynamic programming approaches can still be used. For example, for a graph having negative cycles, the path-state approach can be used to find the shortest *acyclic* path.

1.1.11 All-Pairs Shortest Paths

There are applications where we are interested in finding the shortest path from any source node s to any target node t, i.e., where s and t is an arbitrary pair of the N nodes in a graph having no negative or zero-length cycles and, for the reasons given in the preceding section, having no self-loops. Of course, the procedures discussed in Sect. 1.1.10 can be used to solve this "all-pairs" shortest path problem by treating s and t as variable parameters. In practice, we would want to perform the calculations in a fashion so as to avoid recalculations as much as possible.

Relaxation. Using a staged formulation, let $F(k, p, q)$ be defined as the length of the shortest path from p to q having k or fewer branches. Then, applying the relaxation idea of (1.34), the DPFE for the target-state formulation is

$$F(k, p, q) = \min\{F(k-1, p, q), \min_r \{b(p, r) + F(k-1, r, q)\}\}, \qquad (1.41)$$

for $k > 0$, with $F(0, p, q) = 0$ if $p = q$ and $F(0, p, q) = \infty$ if $p \neq q$. The analogous DPFE for the designated-source formulation is

$$F'(k, p, q) = \min\{F'(k-1, p, q), \min_r \{F'(k-1, p, r) + b(r, q)\}\}. \qquad (1.42)$$

If we artificially let $b(p, p) = 0$ for all p (recall that we assumed there are no self-loops), then the former (1.41) reduces to

$$F(k, p, q) = \min_r \{b(p, r) + F(k-1, r, q)\}, \qquad (1.43)$$

where r may now include p, and analogously for the latter (1.42). The Bellman-Ford variation, that uses $F(k,r,q)$ or $F'(k,p,r)$ instead of $F(k-1,r,q)$ or $F'(k-1,p,r)$, respectively, applies to the solution of these equations. If the number of branches in the shortest path from p to q, denoted k^*, is less than $N-1$, then the goal $F(N-1,p,q) = F(k^*,p,q)$ will generally be found without evaluating the sequence $F(k,p,q)$ for all k; when $F(k+1,p,q) = F(k,p,q)$ (for all p and q), the successive approximations process has converged to the desired solution.

Floyd-Warshall. The foregoing DPFE (1.41) is associated with a divide-and-conquer process where a path from p to q (having at most k branches) is divided into subpaths from p to r (having 1 branch) and from r to q (having at most $k-1$ branches). An alternative is to divide a path from p to q into subpaths from p to r and from r to t that are restricted not by the number of branches they contain but by the set of intermediate nodes r that the paths may traverse. Let $F(k,p,q)$ denote the length of the shortest path from p to q that traverses (passes through) intermediate nodes only in the ordered set $\{1,\ldots,k\}$. Then the appropriate DPFE is

$$F(k,p,q) = \min\{F(k-1,p,q), F(k-1,p,k) + F(k-1,k,q)\}, \quad (1.44)$$

for $k > 0$, with $F(0,p,q) = 0$ if $p = q$ and $F(0,p,q) = b(p,q)$ if $p \neq q$. This is known as the *Floyd-Warshall* all-pairs shortest path algorithm. Unlike the former DPFE (1.41), where r may have up to $N-1$ values (if p has every other node as a successor), the minimization operation in the latter DPFE (1.44) is over only two values.

We note that the DPFEs for the above two algorithms may be regarded as matrix equations, which define matrices F^k in terms of matrices F^{k-1}, where p and q are row and column subscripts; since p, q, r, and k are all $O(N)$, the two algorithms are $O(N^4)$ and $O(N^3)$, respectively.

1.1.12 State Space Generation

The numerical solution of a DPFE requires that a function $f(S)$ be evaluated for all states in some state space S. This requires that these states be generated systematically. State space generation is discussed in, e.g., [12]. Since not all states may be reachable from the goal S^*, it is generally preferable to generate only those states reachable from the source S^*, and that this be done in a breadth-first fashion. For example, in Example 1.1, the generated states, from the source-state or goal to the target-state or base, are (in the order they are generated):

$$\{a,b,c\}, \{b,c\}, \{a,c\}, \{a,b\}, \{c\}, \{b\}, \{a\}, \emptyset.$$

In Example 1.2, these same states are generated in the same order although, since its DPFE is of the reverse designated-source form, the first state is the base and the last state is the goal. In Example 1.3, the generated states, from the source (goal) to the target (base), are:

$$(1, \{a, b, c\}), (2, \{b, c\}), (2, \{a, c\}), (2, \{a, b\}), (3, \{c\}), (3, \{b\}), (3, \{a\}), (4, \emptyset).$$

In Example 1.4, the generated states, from the source (goal) to the target (base), are:

$$(0, \{a, b, c\}), (2, \{b, c\}), (5, \{a, c\}), (3, \{a, b\}), (8, \{c\}), (5, \{b\}), (7, \{a\}), (10, \emptyset).$$

In Example 1.5, the generated states, from the source (goal) to the targets (bases), are:

$$\emptyset, a, b, c, ab, ac, ba, bc, ca, cb, abc, acb, bac, bca, cab, cba.$$

We note that this state space generation process can be automated for a given DPFE, say, of the form (1.17),

$$f(S) = \mathrm{opt}_{d \in D(S)}\{R(S, d) \circ f(T(S, d))\}, \tag{1.45}$$

using $T(S, d)$ to generate next-states starting from S^*, subject to constraints $D(S)$ on d, and terminating when S is a base state. This is discussed further in Sect. 1.2.2.

1.1.13 Complexity

The complexity of a DP algorithm is very problem-dependent, but in general it depends on the exact nature of the DPFE, which is of the general nonserial form

$$f(S) = \mathrm{opt}_{d \in D(S)}\{R(S, d) \circ p_1.f(T_1(S, d)) \circ p_2.f(T_2(S, d)) \circ \ldots\}. \tag{1.46}$$

A foremost consideration is the size or dimension of the state space, because that is a measure of how many optimization operations are required to solve the DPFE. In addition, we must take into account the number of possible decisions for each state. For example, for the shortest path problem, assuming an acyclic graph, there are only N states, and at most $N - 1$ decisions per state, so the DP solution has polynomial complexity $O(N^2)$. We previously gave examples of problems where the size hence complexity was factorial and exponential. Such problems are said to be *intractable*. The fact that in many cases problem size is not polynomial is known as the *curse of dimensionality* which afflicts dynamic programming. For some problems, such as the traveling salesman problem, this intractability is associated with the problem itself, and any algorithm for solving such problems is likewise intractable.

Regardless of whether a problem is tractable or not, it is also of interest to reduce the complexity of any algorithm for solving the problem. We noted from the start that a given problem, even simple ones like linear search, can be solved by different DPFEs. Thus, in general, we should always consider alternative formulations, with the objective of reducing the dimensionality of the state space as a major focus.

1.1.14 Greedy Algorithms

For a given state space, an approach to reducing the dimensionality of a DP solution is to find some way to reduce the number of states for which $f(S)$ must actually be evaluated. One possibility is to use some means to determine the optimal decision $d \in D(S)$ without evaluating each member of the set $\{R(S,d) \circ f(T(S,d))\}$; if the value $R(S,d) \circ f(T(S,d))$ need not be evaluated, then $f(S')$ for next-state $S' = T(S,d)$ may not need to be evaluated. For example, for some problems, it turns out that

$$\text{opt}_{d \in D(S)}\{R(S,d)\} = \text{opt}_{d \in D(S)}\{R(S,d) \circ f(T(S,d))\}, \qquad (1.47)$$

If this is the case, and we solve $\text{opt}_{d \in D(S)}\{R(S,d)\}$ instead of $\text{opt}_{d \in D(S)}$ $\{R(S,d) \circ f(T(S,d))\}$, then we only need to evaluate $f(T(S,d))$ for N states, where N is the number of decisions. We call this the *canonical* greedy algorithm associated with a given DPFE. A *noncanonical* greedy algorithm would be one in which there exists a function Φ for which

$$\text{opt}_{d \in D(S)}\{\Phi(S,d)\} = \text{opt}_{d \in D(S)}\{R(S,d) \circ f(T(S,d))\}. \qquad (1.48)$$

Algorithms based on optimizing an auxiliary function Φ (instead of $R \circ f$) are also called "heuristic" ones.

For one large class of greedy algorithms, known as "priority" algorithms [7], in essence the decision set D is ordered, and decisions are made in that order.

Regrettably, there is no simple test for whether optimal greedy policies exist for an arbitrary DP problem. See [38] for a further discussion of greedy algorithms and dynamic programming.

1.1.15 Probabilistic DP

Probabilistic elements can be added to a DP problem in several ways. For example, rewards (costs or profits) can be made random, depending for example on some random variable. In this event, it is common to simply define the reward function $R(S,d)$ as an expected value. In addition, next-states can be random. For example, given the current state is S and the decision is d, if $T_1(S,d))$ and $T_2(S,d))$ are two possible next-states having probabilities p_1 and p_2, respectively, then the probabilistic DPFE would typically have the form

$$f(S) = \min_{d \in D(S)}\{R(S,d) + p_1.f(T_1(S,d)) + p_2.f(T_2(S,d))\}. \qquad (1.49)$$

It is common for probabililistic DP problems to be staged. For finite horizon problems, where the number of stages N is a given finite number, such DPFEs can be solved just as any nonserial DPFE of order 2. In Chap. 2, we give several examples. However, for infinite horizon problems, where the state space is not finite, iterative methods (see [13]) are generally necessary. We will not discuss these in this book.

For some problems, rather than minimizing or maximizing some total reward, we may be interested instead in minimizing or maximizing the probability of some event. Certain problems of this type can be handled by defining base conditions appropriately. An example illustrating this will also be given in Chap. 2.

1.1.16 Nonoptimization Problems

Dynamic programming can also be used to solve nonoptimization probems, where the objective is not to determine a sequence of decisions that *optimizes* some numerical function. For example, we may wish to determine *any* sequence of decisions that leads from a given goal state to one or more given target states. The Tower of Hanoi problem (see [57, p.332–337] and [58]) is one such example. The objective of this problem is to move a tower (or stack) of N discs, of increasing size from top to bottom, from one peg to another peg using a third peg as an intermediary, subject to the constraints that on any peg the discs must remain of increasing size and that only “basic” moves of one disc at a time are allowed. We will denote the basic move of a disc from peg x to peg y by $< x, y >$. For this problem, rather than defining $f(S)$ as the minimum or maximum value of an objective function, we define $F(S)$ as a *sequence* of basic moves. Then $F(S)$ is the concatenation of the sequence of moves for certain subproblems, and we have

$$F(N, x, y) = F(N-1, x, z)F(1, x, y)F(N-1, z, y). \tag{1.50}$$

Here, the state $S = (N, x, y)$ is the number N of discs to be moved from peg x to peg y using peg z as an intermediary. This DPFE has no *min* or *max* operation. The value of $F(S)$ is a sequence (or string), not a number. The idea of solving problems in terms of subproblems characterizes DP formulations.

The DPFE (1.50) is based on the observation that, to move m discs from peg i to peg j with peg k as an intermediary, we may move $m-1$ discs from i to k with j as an intermediary, then move the last disc from i to j, and finally move the $m-1$ discs on k to j with i as an intermediary. The goal is $F(N, i, j)$, and the base condition is $F(m, i, j) =< i, j >$ when $m = 1$. These base conditions correspond to basic moves. For example, for $N = 3$ and pegs A, B,and C,

$$\begin{aligned} F(3, A, B) &= F(2, A, C)F(1, A, B)F(2, C, B) \\ &= <A, B><A, C><B, C><A, B><C, A><C, B><A, B> . \end{aligned}$$

In this book, our focus is on numerical optimization problems, so we will consider a variation of the Tower of Hanoi problem, where we wish to determine the *number* $f(N)$ of required moves, as a function of the number N of discs to be moved. Then, we have

$$f(N) = 2f(N-1) + 1, \tag{1.51}$$

The base condition for this DPFE is $f(1) = 1$. This recurrence relation and its analytical solution appear in many books, e.g., [53].

It should be emphasized that the foregoing DPFE has no explicit optimization operation, but we can add one as follows:

$$f(N) = \text{opt}_{d\in D}\{2f(N-1) + 1\}, \tag{1.52}$$

where the decision set D has, say, a singleton "dummy" member that is not referenced within the optimand. As another example, consider

$$f(N) = \text{opt}_{d\in D}\{f(N-1) + f(N-2)\}, \tag{1.53}$$

with base conditions $f(1) = f(2) = 1$. Its solution is the N-th Fibonacci number.

In principle, the artifice used above, of having a dummy decision, allows general recurrence relations to be regarded as special cases of DPFEs, and hence to be solvable by DP software. This illustrates the generality of DP and DP tools, although we are not recommending that recurrence relations be solved in this fashion.

1.1.17 Concluding Remarks

The introduction to dynamic programming given here only touches the surface of the subject. There is much research on various other aspects of DP, including formalizations of the class of problems for which DP is applicable, the theoretical conditions under which the Principle of Optimality holds, relationships between DP and other optimization methods, methods for reducing the dimensionality, including approximation methods, especially successive approximation methods in which it is hoped that convergence to the correct answer will result after a reasonable number of iterations, etc.

This book assumes that we can properly formulate a DPFE that solves a given discrete optimization problem. We say a DPFE (with specified base conditions) is *proper*, or properly formulated, if a solution exists and can be found by a finite computational algorithm. Chap. 2 provides many examples that we hope will help readers develop new formulations for their problems of interest. Assuming the DPFE is proper, we then address the problem of numerically solving this DPFE (by describing the design of a software tool for DP. This DP tool has been used for all of our Chap. 2 examples. Furthermore, many of our formulations can be adapted to suit other needs.

1.2 Computational Solution of DPFEs

In this section, we elaborate on how to solve a DPFE. One way in which a DPFE can be solved is by using a "conventional" procedural programming language such as Java. In Sect. 1.2.1, a Java program to solve Example 1.1 is given as an illustration.

1.2.1 Solution by Conventional Programming

A simple Java program to solve Example 1.1 is given here. This program was intentionally written as quickly as possible rather than with great care to reflect what a nonprofessional programmer might produce. A central theme of this book is to show how DP problems can be solved with a minimum amount of programming knowledge or effort. The program as written first solves the DPFE (1.23) [Method S] recursively. This is followed by an iterative procedure to reconstruct the optimal policy. It should be emphasized that this program does not generalize easily to other DPFEs, especially when states are sets rather than integers.

```
class dpfe {

  public static double[][] b= {
      { 999.,   .2,   .5,   .3, 999., 999., 999., 999.},
      { 999., 999., 999., 999.,   1.,   .6, 999., 999.},
      { 999., 999., 999., 999.,   .4, 999.,   .6, 999.},
      { 999., 999., 999., 999., 999.,   .4,   1., 999.},
      { 999., 999., 999., 999., 999., 999., 999.,   .9},
      { 999., 999., 999., 999., 999., 999., 999.,  1.5},
      { 999., 999., 999., 999., 999., 999., 999.,   .6},
      { 999., 999., 999., 999., 999., 999., 999., 999.}
    } ; //branch distance array
  public static int N = b.length;        //number of nodes
  public static int[] ptr = new int[N];  //optimal decisions

  public static double fct(int s) {
    double value=999.; ptr[s]=-1;
    if (s==N-1) {value=0.0; }            // target state
    else
      for (int d=s+1; d<N; d++)            // for s>d
        if (b[s][d]<999.)                    // if d=succ(s)
          if (value>b[s][d]+fct(d))            // if new min
            { value=b[s][d]+fct(d); ptr[s]=d; }  //reset
    return value;
  } //end fct

  public static void main(String[] args) {
    System.out.println("min="+fct(0));     //compute goal
    int i=0; System.out.print("path:"+i);
    while (ptr[i]>0) {                     //reconstruction
      System.out.print("->"+ptr[i]);
      i=ptr[i];
    }
  } // end main
} // end dpfe
```

A *recursive* solution of the DPFE was chosen because the DPFE itself is a recursive equation, and transforming it to obtain an iterative solution is not a natural process. Such a transformation would generally take a significant amount of effort, especially for nonserial problems. On the other hand, a major disadvantage of recursion is the inefficiency associated with recalculating $f(S)$ for states S that are next-states of many other states. This is analogous to the reason it is preferable to solve the Fibonacci recurrence relation iteratively rather than recursively. Although finding an iterative solution for the Fibonacci problem is easy, and it also happens to be easy for the linear search problem, in general we cannot expect this to be the case for DP problems.

1.2.2 The State-Decision-Reward-Transformation Table

This book will describe an alternative to conventional programming, as illustrated above, based on the ability to automatically generate the state space for a given DPFE. Recall that, for Example 1.1, the state space is the set

$$\{\{a,b,c\},\{b,c\},\{a,c\},\{a,b\},\{c\},\{b\},\{a\},\emptyset\}$$

or

$$\{\{0,1,2\},\{1,2\},\{0,2\},\{0,1\},\{2\},\{1\},\{0\},\emptyset\}$$

if we give the decisions a, b, and c the numerical labels 0,1,2 instead.

The state space for a given DPFE can be generated in the process of producing the State-Decision-Reward-Transformation (SDRT) table. The SDRT table gives, for each state S and for each decision d in the decision space $D(S)$, the reward function $R(S,d)$ and the transformation function(s) $T(S,d)$ for each pair (S,d), starting from the goal state S^*. $T(S,d)$ allows us to generate next-states. For Example 1.1, the SDRT table is given in Table 1.1.

As each next-state S' is generated, if it is not already in the table, it is added to the table and additional rows are added for each of the decisions in $D(S')$. If a base-state is generated, which has no associated decision, no additional rows are added to the table.

Given the SDRT table, for a serial DPFE, we can easily construct a state transition system model whose nodes are the states. For Example 1.1, the (Boolean) adjacency matrix for this state transition model is as follows:

Table 1.1. SDRT Table for Linear Search Example

state	decision	reward	next-states
$\{0,1,2\}$	$d=0$	0.2	$(\{1,2\})$
$\{0,1,2\}$	$d=1$	0.5	$(\{0,2\})$
$\{0,1,2\}$	$d=2$	0.3	$(\{0,1\})$
$\{1,2\}$	$d=1$	1.0	$(\{2\})$
$\{1,2\}$	$d=2$	0.6	$(\{1\})$
$\{0,2\}$	$d=0$	0.4	$(\{2\})$
$\{0,2\}$	$d=2$	0.6	$(\{0\})$
$\{0,1\}$	$d=0$	0.4	$(\{1\})$
$\{0,1\}$	$d=1$	1.0	$(\{0\})$
$\{2\}$	$d=2$	0.9	$(\emptyset)$
$\{1\}$	$d=1$	1.5	$(\emptyset)$
$\{0\}$	$d=0$	0.6	$(\emptyset)$

	1	2	3	4	5	6	7	8
1	0	1	1	1	0	0	0	0
2	0	0	0	0	1	1	0	0
3	0	0	0	0	1	0	1	0
4	0	0	0	0	0	1	1	0
5	0	0	0	0	0	0	0	1
6	0	0	0	0	0	0	0	1
7	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0

The weighted adjacency matrix whose nonzero elements are branch labels is

	1	2	3	4	5	6	7	8
1	0	0.2	0.5	0.3	0	0	0	0
2	0	0	0	0	1.0	0.6	0	0
3	0	0	0	0	0.4	0	0.6	0
4	0	0	0	0	0	0.4	1.0	0
5	0	0	0	0	0	0	0	0.9
6	0	0	0	0	0	0	0	1.5
7	0	0	0	0	0	0	0	0.6
8	0	0	0	0	0	0	0	0

The row and column numbers or indices shown $(1,\ldots,8)$ are not part of the matrix itself; in programming languages, such as Java, it is common to start indexing from zero $(0,\ldots,7)$ instead of one.

Later in this book we show that nonserial DPFEs can be modeled in a similar fashion using a generalization of state transition systems called Petri nets.

1.2.3 Code Generation

The adjacency matrix obtained from the SDRT table associated with a DPFE, as described in Sect. 1.2.2, provides the basis for a DP program generator, i.e., a software tool that automatically generates "solver code", specifically, a sequence of assignment statements for solving a DPFE using a conventional programming language such as Java. We illustrate this solver code generation process in this section.

Given a weighted adjacency matrix, for example, the one given above, we can obtain the numerical solution of the DPFE by defining an assignment statement for each row of the matrix which sets a variable a_i for row i equal to the minimum of terms of the form $c_{i,j} + a_j$, where j is a successor of i.

```
a1=min{.2+a2,.5+a3,.3+a4}
a2=min{1.+a5,.6+a6}
a3=min{.4+a5,.6+a7}
a4=min{.4+a6,1.+a7}
a5=min{.9+a8}
a6=min{1.5+a8}
a7=min{.6+a8}
a8=0
```

These assignment statements can be used in a conventional nonrecursive computer program (in any procedural programming language) to calculate the values a_i. The statements should be compared with the equations of Example 1.1 [Method S]. As in that earlier example, evaluating the values a_i yields the following results: $a_8 = 0, a_7 = 0.6, a_6 = 1.5, a_5 = 0.9, a_4 = \min(1.9, 1.6) = 1.6, a_3 = \min(1.3, 1.2) = 1.2, a_2 = \min(1.9, 2.1) = 1.9, a_1 = \min(2.1, 1.7, 1.9) = 1.7$; note that $a_1 = 1.7$ is the goal. These assignment statements must of course be "topologically" reordered, from last to first, before they are executed.

1.2.4 Spreadsheet Solutions

Above, we showed the basis for a DP program generator that automatically generates a sequence of assignment statements for solving a DPFE using a conventional programming language. We show in this section how a spreadsheet that solves a DPFE can be automatically generated.

The assignment statements given Sect. 1.2.3 for the linear search problem can also be rewritten in the form

```
=min(.2+A2,.5+A3,.3+A4)
=min(1.+A5,.6+A6)
=min(.4+A5,.6+A7)
=min(.4+A6,1.+A7)
=min(.9+A8)
=min(1.5+A8)
```

```
=min(.6+A8)
0
```

which when imported into the first column of a spreadsheet will yield the same results as before; cell A1 of the spreadsheet will have 1.7 as its computed answer. One advantage of this spreadsheet solution is that "topological" sorting is unnecessary.

In this spreadsheet program, only the lengths of the shortest paths are calculated. To reconstruct the optimal policies, i.e. the sequence of decisions that yield the shortest paths, more work must be done. We will not address this reconstruction task further in this Chapter.

The foregoing spreadsheet has formulas that involve both the minimization and addition operations. A simpler "basic" spreadsheet would permit formulas to have only one operation. Suppose we define an intermediary variable a_k for each of the terms $c_{i,j} + a_j$. Then we may rewrite the original sequence of statements as follows:

```
a1=min(a9,a10,a11)
a2=min(a12,a13)
a3=min(a14,a15)
a4=min(a16,a17)
a5=min(a18)
a6=min(a19)
a7=min(a20)
a8=0
a9=.2+a2
a10=.5+a3
a11=.3+a4
a12=1.+a5
a13=.6+a6
a14=.4+a5
a15=.6+a7
a16=.4+a6
a17=1.+a7
a18=.9+a8
a19=1.5+a8
a20=.6+a8
```

As above, we may also rewrite this in spreadsheet form:

```
=min(A9,A10,A11)
=min(A12,A13)
=min(A14,A15)
=min(A16,A17)
=min(A18)
=min(A19)
=min(A20)
```

```
0
=.2+A2
=.5+A3
=.3+A4
=1.+A5
=.6+A6
=.4+A5
=.6+A7
=.4+A6
=1.+A7
=.9+A8
=1.5+A8
=.6+A8
```

This basic spreadsheet is a tabular representation of the original DPFE, and is at the heart of the software system we describe in this book. This software automatically generates the following equivalent spreadsheet from the given DPFE:

```
0
=B1+0.9
=MIN(B2)
=B3+1
=B3+0.4
=B1+1.5
=MIN(B6)
=B7+0.6
=B7+0.4
=MIN(B4,B8)
=B10+0.2
=B1+0.6
=MIN(B12)
=B13+0.6
=B13+1
=MIN(B5,B14)
=B16+0.5
=MIN(B9,B15)
=B18+0.3
=MIN(B11,B17,B19)
```

(Only Column B is shown here.) The different ordering is a consequence of our implementation decisions, but does not affect the results.

1.2.5 Example: SPA

As another illustration, that we will use later in this book since it is a smaller example that can be more easily examined in detail, we consider the shortest path in an acyclic graph (SPA) problem, introduced as Example 1.6 in Sect. 1.1.10. The SDRT table is as follows:

```
StateDecisionRewardTransformationTable
(0) [d=1] 3.0 ((1)) ()
(0) [d=2] 5.0 ((2)) ()
(1) [d=2] 1.0 ((2)) ()
(1) [d=3] 8.0 ((3)) ()
(2) [d=3] 5.0 ((3)) ()
```

From this table, we can generate solver code as a sequence of assignment statements as follows:

```
A1=min(A2+3.0,A3+5.0)
A2=min(A3+1.0,A4+8.0)
A3=min(A4+5.0)
A4=0.0
```

Simplifying the formulas, so that each has only a single (minimization or addition) operation, we may rewrite the foregoing as follows:

```
A1=min(A5,A6)
A2=min(A7,A8)
A3=min(A9)
A4=0.0
A5=A2+3.0
A6=A3+5.0
A7=A3+1.0
A8=A4+8.0
A9=A4+5.0
```

As in the case of the preceding linear search example, these assignment statements must be topologically sorted if they are to be executed as a conventional sequential program. (This sorting is unnecessary if they are imported into a Column A of a spreadsheet.) Rearranging the variables (letting B9=A1, B7=A2, B4=A3, etc.), we have:

```
B1=0.0
B2=B1+8.0
B3=B1+5.0
B4=min(B3)
B5=B4+5.0
B6=B4+1.0
B7=min(B6,B2)
B8=B7+3.0
B9=min(B8,B5)
```

These assignment statements can be executed as a conventional sequential program. Alternatively, importing them into Column B, we arrive at the following spreadsheet solver code:

```
=0.0
=B1+8.0
=B1+5.0
=min(B3)
=B4+5.0
=B4+1.0
=min(B6,B2)
=B7+3.0
=min(B8,B5)
```

1.2.6 Concluding Remarks

It is not easy to modify the above Java or spreadsheet "solver code" to solve DP problems that are dissimilar to linear search or shortest paths. Conventional programming and hand-coding spreadsheets, especially for problems of larger dimension, are error-prone tasks. The desirability of a software tool that automatically generates solver code from a DPFE is clear. That is the focus of this book.

1.3 Overview of Book

In Chap. 2, we discuss numerous applications of DP. Specifically, we formulate a DPFE for each of these applications. For many applications, we provide alternate formulations as well. This compendium of examples shows the generality and flexibility of dynamic programming as an optimization method and of the DP2PN2Solver software tool described in this book for solving dynamic programming problems.

In Chap. 3, we describe gDPS, a text-based specification language for dynamic programming problems. gDPS serves as the input language for the DP2PN2Solver tool. Its syntax is given in BNF form. In effect, a gDPS source program is a transliteration of a DPFE.

In Chap. 4, we show how each of the DPFEs given in Chap. 2 can be expressed in the gDPS language of Chap. 3. The result is a set of computer programs for solving the DP problems given in Chap. 2.

In Chap. 5, we define Bellman nets, a class of Petri nets, which serve as a useful model of DPFEs. Petri nets, hence also Bellman nets, may be regarded as a class of directed graphs.

In Chap. 6, we show how the DPFEs in Chap. 2 or Chap. 4 can be represented as a Bellman net.

In Chap. 7, we describe the overall structure of DP2PN2Solver, a "compiler" tool whose (source code) input is a DPFE and whose (object code) output is "solver code", i.e., a program which when executed solves the DPFE. The first phase of this tool translates a DPFE into its Bellman net representation, and the second phase translates the Bellman net into solver code. In Sect. 7.2, we show the internal Bellman net representations for the DPFEs in Chap. 2. Unlike the graphical representation in Chap. 6, the internal representation is tabular, well suited as an intermediary between Phases 1 and 2.

In Chap. 8, we describe Phase 1 of our DP tool, which parses DPFE source code and outputs its Bellman net representation. This Bellman net is produced indirectly: the parser generates intermediate "builder code" which when executed outputs the Bellman net.

In Chap. 9, we describe Phase 2 of our DP tool, which inputs a Bellman net and produces solver code which when executed outputs a numerical solution of the associated DPFE. This solver code can be Java code, a spreadsheet, or an XML-coded Petri net (using a Petri net simulation language). In the case of Java code, the solver code produced does not directly calculate results, but consists instead of calls to systems routines that perform the calculations.

In Chap. 10, we show the numerical outputs obtained by executing the Java solver codes produced by Phase 2 from the Bellman nets for each of the problems of Chap. 2 and Chap. 4.

In Chap. 11, we show the numerical outputs obtained by executing the spreadsheet and XML solver codes produced by Phase 2 for some sample Bellman nets.

Chapter 12 concludes the book with a brief summary and a discussion of current research into ways to improve and extend our DP software tool.

Appendix A provides supplementary program listings that we include for completeness, detailing key portions of our software tool, for example.

Appendix B is a User's Guide for our tool, including downloading and installation instructions.