# 1

# *Exact and Approximate Dynamic Programming Principles*

**Contents**

In this chapter, we provide some background on exact dynamic programming (DP), with a view towards the suboptimal solution methods, which are based on approximation in value space and are the main subject of this book. We first discuss finite horizon problems, which involve a finite sequence of successive decisions, and are thus conceptually and analytically simpler. We then consider somewhat briefly the more intricate infinite horizon problems, but defer a more detailed treatment for Chapter 5.

We will discuss separately deterministic and stochastic problems (Sections 1.2 and 1.3, respectively). The reason is that deterministic problems are simpler and have some favorable characteristics, which allow the application of a broader variety of methods. Significantly they include challenging discrete and combinatorial optimization problems, which can be fruitfully addressed with some of the rollout and approximate policy iteration methods that are the main focus of this book.

In subsequent chapters, we will discuss selectively some major algorithmic topics in approximate DP and reinforcement learning (RL), including rollout and policy iteration, multiagent problems, and distributed algorithms. A broader discussion of DP/RL may be found in the author's RL textbook [Ber19a], and the DP textbooks [Ber12], [Ber17a], [Ber18a], the neuro-dynamic programming monograph [BeT96], as well as the literature cited in the last section of this chapter.

The DP/RL methods that are the principal subjects of this book, rollout and policy iteration, have a strong connection with the famous AlphaZero, AlphaGo, and other related programs. As an introduction to our technical development, we take a look at this connection in the next section.

## 1.1 AlphaZero, Off-Line Training, and On-Line Play

One of the most exciting recent success stories in RL is the development of the AlphaGo and AlphaZero programs by DeepMind Inc; see [SHM16], [SHS17], [SSS17]. AlphaZero plays Chess, Go, and other games, and is an improvement in terms of performance and generality over AlphaGo, which plays the game of Go only. Both programs play better than all competitor computer programs available in 2020, and much better than all humans. These programs are remarkable in several other ways. In particular, they have learned how to play without human instruction, just data generated by playing against themselves. Moreover, they learned how to play very quickly. In fact, AlphaZero learned how to play chess better than all humans and computer programs within hours (with the help of awesome parallel computation power, it must be said).

Perhaps the most impressive aspect of AlphaZero/chess is that its play is not just better, but it is also very different than human play in terms of long term strategic vision. Remarkably, AlphaZero has discovered

new ways to play chess, a game that has been studied intensively by humans for hundreds of years.

Still, for all of its impressive success and brilliant implementation, AlphaZero is couched on well established methodology, which is the subject of the present book, and is portable to far broader realms of engineering, economics, and other fields. This is the methodology of DP, policy iteration, limited lookahead, rollout, and approximation in value space.†

To understand the overall structure of AlphaZero and related programs, and their connections to our DP/RL methodology, it is useful to divide their design into two parts:

(1) *Off-line training*, which is an algorithm that learns how to evaluate chess positions, and how to steer itself towards good positions with a default/base chess player.

(2) *On-line play*, which is an algorithm that generates good moves in real time against a human or computer opponent, using the training it went through off-line.

We will next briefly describe these algorithms, and relate them to DP concepts and principles.

### Off-Line Training and Policy Iteration

An off-line training algorithm like the one used in AlphaZero is the part of the program that learns how to play through self-training that takes place before real-time play against any opponent. It is illustrated in Fig. 1.1.1, and it generates a sequence of *chess players* and *position evaluators*. A chess player assigns "probabilities" to all possible moves at any given chess position (these are the probabilities with which the player selects the possible moves at the given position). A position evaluator assigns a numerical score to any given chess position (akin to a "probability" of winning the game from that position), and thus predicts quantitatively the performance of a player starting from any position. The chess player and the position evaluator are represented by two neural networks, a *policy*

† It is also worth noting that the principles of the AlphaZero design have much in common with the work of Tesauro [Tes94], [Tes95], [TeG96] on computer backgammon. Tesauro's programs stimulated much interest in RL in the middle 1990s, and exhibit similarly different and better play than human backgammon players. A related impressive program for the (one-player) game of Tetris, also based on the method of policy iteration, is described by Scherrer et al. [SGG15], together with several antecedents. Also the AlphaZero ideas have been replicated by the publicly available program Leela Chess Zero, with similar success. For a better understanding of the connections of AlphaZero, Tesauro's programs (TD-Gammon [Tes94], and its rollout version [TeG96]), and the concepts developed here, the reader may consult the "Methods" section of the paper [SSS17].

*network* and a *value network*, which accept a chess position and generate a set of move probabilities and a position evaluation, respectively.†

In the more conventional DP-oriented terms of this book, a position is the state of the game, a position evaluator is a cost function that gives the cost-to-go at a given state, and the chess player is a randomized policy for selecting actions/controls at a given state.‡

The overall training algorithm is a form of *policy iteration*, a DP algorithm that will be of primary interest to us in this book. Starting from a given player, it repeatedly generates (approximately) improved players, and settles on a final player that is judged empirically to be "best" out of all the players generated.†† Policy iteration may be separated conceptually into two stages (see Fig. 1.1.1).

(1) *Policy evaluation*: Given the current player and a chess position, the outcome of a game played out from the position provides a single data point. Many data points are thus collected, and are used to train a value network, whose output serves as the position evaluator for that player.

---

† Here the neural networks play the role of *function approximators*. By viewing a player as a function that assigns move probabilities to a position, and a position evaluator as a function that assigns a numerical score to a position, the policy and value networks provide approximations to these functions based on training with data (training algorithms for neural networks and other approximation architectures will be discussed in Chapter 4). Actually, AlphaZero uses the same neural network for training both value and policy. Thus there are two outputs of the neural net: value and policy. This is pretty much equivalent to having two separate neural nets and for the purpose of the book, we prefer to explain the structure as two separate networks. AlphaGo uses two separate value and policy networks. Tesauro's backgammon programs use a single value network, and generate moves when needed by one-step or two-step lookahead minimization, using the value network as terminal position evaluator.

‡ One more complication is that chess and Go are two-player games, while most of our development will involve single-player optimization. However, DP theory and algorithms extend to two-player games, although we will not discuss these extensions, except briefly in Chapters 3 and 5 (see Sections 3.6 and 5.5).

†† Quoting from the paper [SSS17]: "The AlphaGo Zero selfplay algorithm can similarly be understood as an approximate policy iteration scheme in which MCTS is used for both policy improvement and policy evaluation. Policy improvement starts with a neural network policy, executes an MCTS based on that policy's recommendations, and then projects the (much stronger) search policy back into the function space of the neural network. Policy evaluation is applied to the (much stronger) search policy: the outcomes of selfplay games are also projected back into the function space of the neural network. These projection steps are achieved by training the neural network parameters to match the search probabilities and selfplay game outcome respectively."

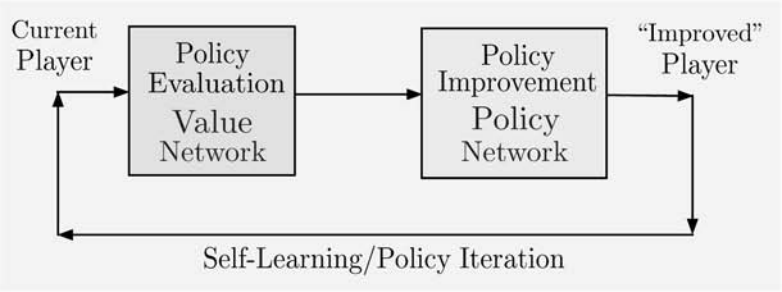


**Figure 1.1.1** Illustration of the AlphaZero off-line training algorithm. It generates a sequence of position evaluators and chess players. The position evaluator and the chess player are represented by two neural networks, a value network and a policy network, which accept a chess position and generate a position evaluation and a set of move probabilities, respectively.

(2) *Policy improvement*: Given the current player and its position evaluator, trial move sequences are selected and evaluated for the remainder of the game starting from many positions. An improved player is then generated by adjusting the move probabilities of the current player towards the trial moves that have yielded the best results. In AlphaZero this is done with a complicated algorithm called *Monte Carlo Tree Search*, which will be described in Chapter 2. However, policy improvement can be done more simply. For example one could try all possible move sequences from a given position, extending forward to a given number of moves, and then evaluate the terminal position with the player's position evaluator. The move evaluations obtained in this way are used to nudge the move probabilities of the current player towards more successful moves, thereby obtaining data that is used to train a policy network that represents the new player.

### On-Line Play and Approximation in Value Space - Rollout

Consider now the "final" player obtained through the AlphaZero off-line training process. It can play against any opponent by generating move probabilities at any position using its off-line trained policy network, and then simply play the move of highest probability. This player would play very fast on-line, but it would not play good enough chess to beat strong human opponents. The extraordinary strength of AlphaZero is attained only after the player obtained from off-line training is embedded into another algorithm, which we refer to as the "on-line player."† In other words *AlphaZero plays on-line much better than the best player it has produced*

† Quoting from the paper [SSS17]: "The MCTS search outputs probabilities of playing each move. These search probabilities usually select much stronger moves than the raw move probabilities of the neural network."

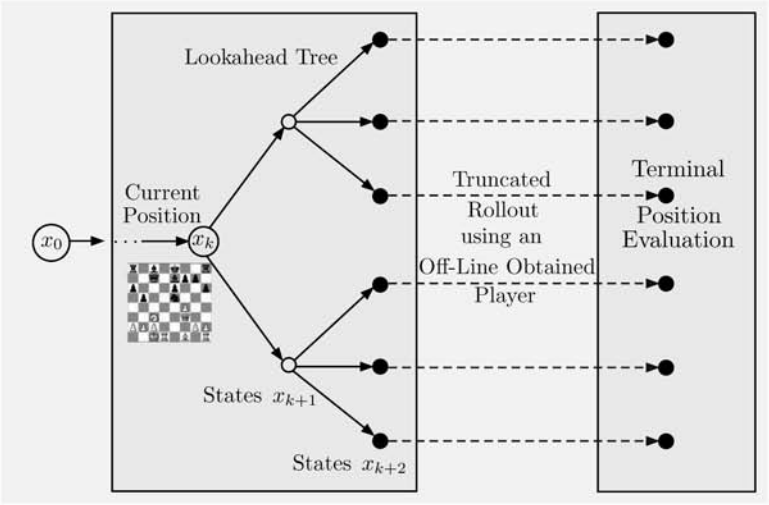


**Figure 1.1.2** Illustration of an on-line player such as the one used in AlphaGo, AlphaZero, and Tesauro's backgammon program [TeG96]. At a given position, it generates a lookahead tree of multiple moves up to a given depth, then runs the off-line obtained player for some more moves, and then evaluates the effect of the remaining moves by using the position evaluator of the off-line obtained player.

*with sophisticated off-line training.* This phenomenon, *policy improvement through on-line play*, is centrally important for our purposes in this book.

Given the policy network/player obtained off-line and its value network/position evaluator, the on-line algorithm plays roughly as follows (see Fig. 1.1.2). At a given position, it generates a lookahead tree of all possible multiple move and countermove sequences, up to a given depth. It then runs the off-line obtained player for some more moves, and then evaluates the effect of the remaining moves by using the position evaluator of the value network. The middle portion, called "truncated rollout," may be viewed as *an economical substitute for longer lookahead*. Actually truncated rollout is not used in the published version of AlphaZero [SHS17]; the first portion (multistep lookahead) is quite long and implemented efficiently, so that the rollout portion is not essential. However, rollout is used in AlphaGo [SHM16]. Moreover, many chess and Go programs use a limited form of rollout, called quiescence search, which aims to resolve imminent threats and highly dynamic positions before invoking the position evaluator. Rollout is instrumental in achieving high performance in Tesauro's 1996 backgammon program [TeG96]. The reason is that backgammon involves stochastic uncertainty, so long lookahead is not possible because of rapid expansion of the lookahead tree with every move.†

† Tesauro's rollout-based backgammon program [TeG96] uses only a value network, called TD-Gammon, which was trained using an approximate policy

We should note that the preceding description of AlphaZero and related games is oversimplified. We will be adding refinements and details as the book progresses. However, DP ideas with cost function approximations, similar to the on-line player illustrated in Fig. 1.1.2, will be central for our purposes. They will be generically referred to as *approximation in value space*. Moreover, the conceptual division between off-line training and on-line policy implementation will be important for our purposes.

Note also that these two processes may be decoupled and may be designed independently. For example the off-line training portion may be very simple, such as using a known heuristic policy for rollout without truncation, or without terminal cost approximation. Conversely, a sophisticated process may be used for off-line training of a terminal cost function approximation, which is used immediately following one-step or multistep lookahead in a value space approximation scheme.

## 1.2 Deterministic Dynamic Programming

In all DP problems, the central object is a discrete-time dynamic system that generates a sequence of states under the influence of control. The system may evolve deterministically or randomly (under the additional influence of a random disturbance).

### 1.2.1 Finite Horizon Problem Formulation

In finite horizon problems the system evolves over a finite number $N$ of time steps (also called stages). The state and control at time $k$ of the system will be generally denoted by $x_k$ and $u_k$, respectively. In deterministic systems, $x_{k+1}$ is generated nonrandomly, i.e., it is determined solely by $x_k$ and $u_k$. Thus, a deterministic DP problem involves a system of the form

$$x_{k+1} = f_k(x_k, u_k), \qquad k = 0, 1, \cdots, N-1 \tag{1.1}$$

where $k$ is the time index, and

$x_k$ is the state of the system, an element of some space,

$u_k$ is the control or decision variable, to be selected at time $k$ from some given set $U_k(x_k)$ that depends on $x_k$,

---

iteration scheme developed several years earlier [Tes94]. TD-Gammon is used to generate moves for the truncated rollout via a one-step or two-step lookahead minimization. Thus the value network also serves as a substitute for the policy network during the rollout operation. The terminal position evaluation used at the end of the truncated rollout is also provided by the value network. The middle portion of Tesauro's scheme (truncated rollout) is important for achieving a very high quality of play, as it effectively extends the length of lookahead from the current position.

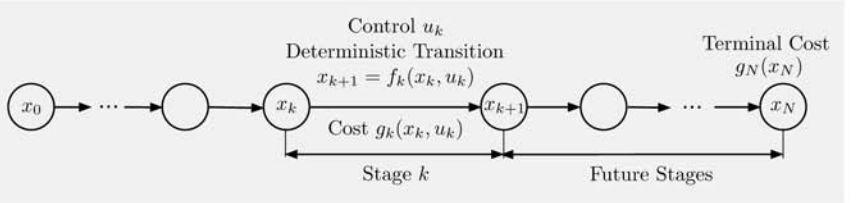


**Figure 1.2.1** Illustration of a deterministic $N$-stage optimal control problem. Starting from state $x_k$, the next state under control $u_k$ is generated nonrandomly, according to

$$x_{k+1} = f_k(x_k, u_k)$$

and a stage cost $g_k(x_k, u_k)$ is incurred.

$f_k$ is a function of $(x_k, u_k)$ that describes the mechanism by which the state is updated from time $k$ to time $k+1$,

$N$ is the horizon, i.e., the number of times control is applied.

The set of all possible $x_k$ is called the *state space* at time $k$. It can be any set and may depend on $k$. Similarly, the set of all possible $u_k$ is called the *control space* at time $k$. Again it can be any set and may depend on $k$. Similarly the system function $f_k$ can be arbitrary and may depend on $k$.†

The problem also involves a cost function that is additive in the sense that the cost incurred at time $k$, denoted by $g_k(x_k, u_k)$, accumulates over time. Formally, $g_k$ is a function of $(x_k, u_k)$ that takes real number values, and may depend on $k$. For a given initial state $x_0$, the total cost of a control sequence $\{u_0, \cdots, u_{N-1}\}$ is

$$J(x_0; u_0, \cdots, u_{N-1}) = g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k) \tag{1.2}$$

† This generality is one of the great strengths of the DP methodology and guides the exposition style of this book, and the author's other DP works. By allowing arbitrary state and control spaces (discrete, continuous, or mixtures thereof), and a $k$-dependent choice of these spaces, we can focus attention on the truly essential algorithmic aspects of the DP approach, exclude extraneous assumptions and constraints from our model, and avoid duplication of analysis.

The generality of our DP model is also partly responsible for our choice of notation. In the artificial intelligence and operations research communities, finite state models, often referred to as Markovian Decision Problems (MDP), are common and use a transition probability notation (see Chapter 5). Unfortunately, this notation is not well suited for deterministic models, and also for continuous spaces models, both of which are important for the purposes of this book. For the latter models, it involves transition probability distributions over continuous spaces, and leads to mathematics that are far more complex as well as less intuitive than those based on the use of the system function (1.1).

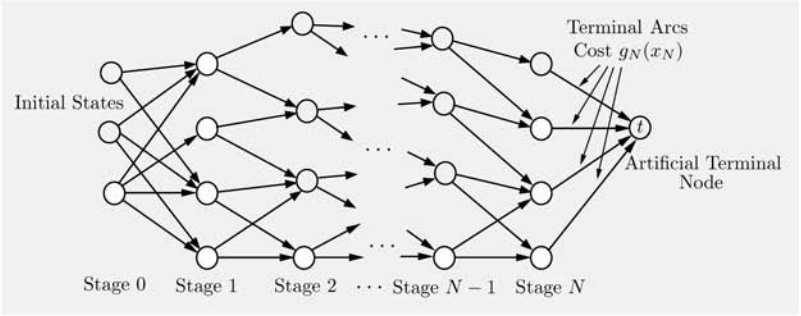


**Figure 1.2.2** Transition graph for a deterministic finite-state system. Nodes correspond to states $x_k$. Arcs correspond to state-control pairs $(x_k, u_k)$. An arc $(x_k, u_k)$ has start and end nodes $x_k$ and $x_{k+1} = f_k(x_k, u_k)$, respectively. The transition cost $g_k(x_k, u_k)$ is viewed as the length of this arc. The problem is equivalent to finding a shortest path from initial nodes of stage 0 to an artificial terminal node $t$.

where $g_N(x_N)$ is a terminal cost incurred at the end of the process. This is a well-defined number, since the control sequence $\{u_0, \cdots, u_{N-1}\}$ together with $x_0$ determines exactly the state sequence $\{x_1, \cdots, x_N\}$ via the system equation (1.1); see Figure 1.2.1. We want to minimize the cost (1.2) over all sequences $\{u_0, \cdots, u_{N-1}\}$ that satisfy the control constraints, thereby obtaining the optimal value as a function of $x_0$:†

$$J^*(x_0) = \min_{\substack{u_k \in U_k(x_k) \\ k=0,\cdots,N-1}} J(x_0; u_0, \cdots, u_{N-1})$$

### Discrete Optimal Control Problems

There are many situations where the state and control spaces are naturally discrete and consist of a finite number of elements. Such problems are often conveniently described with an acyclic graph specifying for each state $x_k$ the possible transitions to next states $x_{k+1}$. The nodes of the graph correspond to states $x_k$ and the arcs of the graph correspond to state-control pairs $(x_k, u_k)$. Each arc with start node $x_k$ corresponds to a choice of a single control $u_k \in U_k(x_k)$ and has as end node the next state $f_k(x_k, u_k)$. The cost of an arc $(x_k, u_k)$ is defined as $g_k(x_k, u_k)$; see Fig. 1.2.2. To handle the final stage, an artificial terminal node $t$ is added. Each state $x_N$ at stage $N$ is connected to the terminal node $t$ with an arc having cost $g_N(x_N)$.

Note that control sequences $\{u_0, \cdots, u_{N-1}\}$ correspond to paths originating at the initial state (a node at stage 0) and terminating at one of the nodes corresponding to the final stage $N$. If we view the cost of an arc as

† Here and later we write "min" (rather than "inf") even if we are not sure that the minimum is attained. Similarly we write "max" (rather than "sup") even if we are not sure that the maximum is attained.

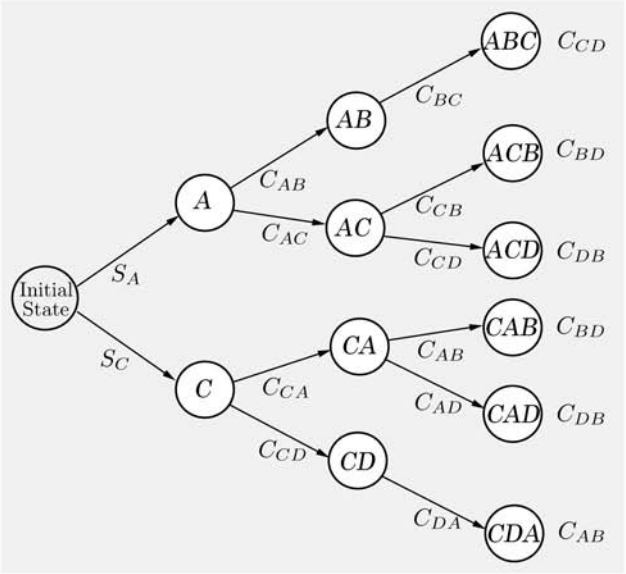


**Figure 1.2.3** The transition graph of the deterministic scheduling problem of Example 1.2.1. Each arc of the graph corresponds to a decision leading from some state (the start node of the arc) to some other state (the end node of the arc). The corresponding cost is shown next to the arc. The cost of the last operation is shown as a terminal cost next to the terminal nodes of the graph.

its length, we see that *a deterministic finite-state finite-horizon problem is equivalent to finding a minimum-length (or shortest) path from the initial nodes of the graph (stage 0) to the terminal node t*. Here, by the length of a path we mean the sum of the lengths of its arcs.†

Generally, combinatorial optimization problems can be formulated as deterministic finite-state finite-horizon optimal control problem, as we will discuss in greater detail in Chapters 2 and 3. The idea is to break down the solution into components, which can be computed sequentially. The following is an illustrative example.

### Example 1.2.1 (A Deterministic Scheduling Problem)

Suppose that to produce a certain product, four operations must be performed on a certain machine. The operations are denoted by A, B, C, and D. We assume that operation B can be performed only after operation A has been performed, and operation D can be performed only after operation C has been performed. (Thus the sequence CDAB is allowable but the sequence CDBA

† It turns out also that any shortest path problem (with a possibly nonacyclic graph) can be reformulated as a finite-state deterministic optimal control problem. See [Ber17a], Section 2.1, and [Ber91], [Ber98] for extensive accounts of shortest path methods, which connect with our discussion here.

is not.) The setup cost $C_{mn}$ for passing from any operation $m$ to any other operation $n$ is given (cf. Fig. 1.2.3). There is also an initial startup cost $S_A$ or $S_C$ for starting with operation A or C, respectively. The cost of a sequence is the sum of the setup costs associated with it; for example, the operation sequence ACDB has cost $S_A + C_{AC} + C_{CD} + C_{DB}$.

We can view this problem as a sequence of three decisions, namely the choice of the first three operations to be performed (the last operation is determined from the preceding three). It is appropriate to consider as state the set of operations already performed, the initial state being an artificial state corresponding to the beginning of the decision process. The possible state transitions corresponding to the possible states and decisions for this problem are shown in Fig. 1.2.3. Here the problem is deterministic, i.e., at a given state, each choice of control leads to a uniquely determined state. For example, at state AC the decision to perform operation D leads to state ACD with certainty, and has cost $C_{CD}$. Thus the problem can be conveniently represented with the transition graph of Fig. 1.2.3. The optimal solution corresponds to the path that starts at the initial state and ends at some state at the terminal time and has minimum sum of arc costs plus the terminal cost.

### 1.2.2 The Dynamic Programming Algorithm

In this section we will state the DP algorithm and formally justify it. The algorithm rests on a simple idea, the *principle of optimality*, which roughly states the following; see Fig. 1.2.4.

**Principle of Optimality**

Let $\{u_0^*, \cdots, u_{N-1}^*\}$ be an optimal control sequence, which together with $x_0$ determines the corresponding state sequence $\{x_1^*, \cdots, x_N^*\}$ via the system equation (1.1). Consider the subproblem whereby we start at $x_k^*$ at time $k$ and wish to minimize the "cost-to-go" from time $k$ to time $N$,

$$g_k(x_k^*, u_k) + \sum_{m=k+1}^{N-1} g_m(x_m, u_m) + g_N(x_N)$$

over $\{u_k, \cdots, u_{N-1}\}$ with $u_m \in U_m(x_m)$, $m = k, \cdots, N-1$. Then the truncated optimal control sequence $\{u_k^*, \cdots, u_{N-1}^*\}$ is optimal for this subproblem.

The subproblem referred to above is called the *tail subproblem* that starts at $x_k^*$. Stated succinctly, the principle of optimality says that *the tail of an optimal sequence is optimal for the tail subproblem.* Its intuitive justification is simple. If the truncated control sequence $\{u_k^*, \cdots, u_{N-1}^*\}$ were not optimal as stated, we would be able to reduce the cost further by switching to an optimal sequence for the subproblem once we reach $x_k^*$

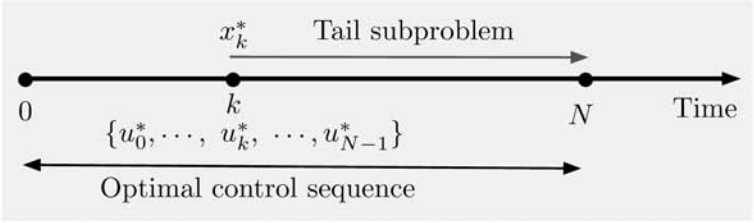


**Figure 1.2.4** Schematic illustration of the principle of optimality. The tail $\{u_k^*, \cdots, u_{N-1}^*\}$ of an optimal sequence $\{u_0^*, \cdots, u_{N-1}^*\}$ is optimal for the tail subproblem that starts at the state $x_k^*$ of the optimal state trajectory.

(since the preceding choices of controls, $u_0^*, \cdots, u_{k-1}^*$, do not restrict our future choices).

For an auto travel analogy, suppose that the fastest route from Phoenix to Boston passes through St Louis. The principle of optimality translates to the obvious fact that the St Louis to Boston portion of the route is also the fastest route for a trip that starts from St Louis and ends in Boston.†

The principle of optimality suggests that the optimal cost function can be constructed in piecemeal fashion going backwards: first compute the optimal cost function for the "tail subproblem" involving the last stage, then solve the "tail subproblem" involving the last two stages, and continue in this manner until the optimal cost function for the entire problem is constructed.

The DP algorithm is based on this idea: it proceeds sequentially, by *solving all the tail subproblems of a given time length, using the solution of the tail subproblems of shorter time length*. We illustrate the algorithm with the scheduling problem of Example 1.2.1. The calculations are simple but tedious, and may be skipped without loss of continuity. However, they may be worth going over by a reader that has no prior experience in the use of DP.

### Example 1.2.1 (Scheduling Problem - Continued)

Let us consider the scheduling Example 1.2.1, and let us apply the principle of optimality to calculate the optimal schedule. We have to schedule optimally the four operations $A$, $B$, $C$, and $D$. There is a cost for a transition between two operations, and the numerical values of the transition costs are shown in Fig. 1.2.5 next to the corresponding arcs.

According to the principle of optimality, the "tail" portion of an optimal schedule must be optimal. For example, suppose that the optimal schedule

† In the words of Bellman [Bel57]: "An optimal trajectory has the property that at an intermediate point, no matter how it was reached, the rest of the trajectory must coincide with an optimal trajectory as computed from this intermediate point as the starting point."

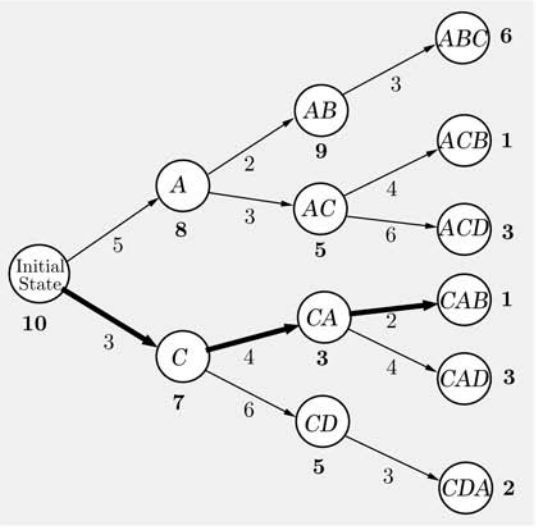


**Figure 1.2.5** Transition graph of the deterministic scheduling problem, with the cost of each decision shown next to the corresponding arc. Next to each node/state we show the cost to optimally complete the schedule starting from that state. This is the optimal cost of the corresponding tail subproblem (cf. the principle of optimality). The optimal cost for the original problem is equal to 10, as shown next to the initial state. The optimal schedule corresponds to the thick-line arcs.

is $CABD$. Then, having scheduled first $C$ and then $A$, it must be optimal to complete the schedule with $BD$ rather than with $DB$. With this in mind, we solve all possible tail subproblems of length two, then all tail subproblems of length three, and finally the original problem that has length four (the subproblems of length one are of course trivial because there is only one operation that is as yet unscheduled). As we will see shortly, the tail subproblems of length $k+1$ are easily solved once we have solved the tail subproblems of length $k$, and this is the essence of the DP technique.

*Tail Subproblems of Length 2*: These subproblems are the ones that involve two unscheduled operations and correspond to the states $AB$, $AC$, $CA$, and $CD$ (see Fig. 1.2.5).

*State $AB$*: Here it is only possible to schedule operation $C$ as the next operation, so the optimal cost of this subproblem is 9 (the cost of scheduling $C$ after $B$, which is 3, plus the cost of scheduling $D$ after $C$, which is 6).

*State $AC$*: Here the possibilities are to (1) schedule operation $B$ and then $D$, which has cost 5, or (2) schedule operation $D$ and then $B$, which has cost 9. The first possibility is optimal, and the corresponding cost of the tail subproblem is 5, as shown next to node $AC$ in Fig. 1.2.5.

*State $CA$*: Here the possibilities are to (1) schedule operation B and then

$D$, which has cost 3, or (2) schedule operation $D$ and then $B$, which has cost 7. The first possibility is optimal, and the corresponding cost of the tail subproblem is 3, as shown next to node $CA$ in Fig. 1.2.5.

*State CD*: Here it is only possible to schedule operation $A$ as the next operation, so the optimal cost of this subproblem is 5.

*Tail Subproblems of Length 3*: These subproblems can now be solved using the optimal costs of the subproblems of length 2.

*State A*: Here the possibilities are to (1) schedule next operation $B$ (cost 2) and then solve optimally the corresponding subproblem of length 2 (cost 9, as computed earlier), a total cost of 11, or (2) schedule next operation $C$ (cost 3) and then solve optimally the corresponding subproblem of length 2 (cost 5, as computed earlier), a total cost of 8. The second possibility is optimal, and the corresponding cost of the tail subproblem is 8, as shown next to node $A$ in Fig. 1.2.5.

*State C*: Here the possibilities are to (1) schedule next operation $A$ (cost 4) and then solve optimally the corresponding subproblem of length 2 (cost 3, as computed earlier), a total cost of 7, or (2) schedule next operation $D$ (cost 6) and then solve optimally the corresponding subproblem of length 2 (cost 5, as computed earlier), a total cost of 11. The first possibility is optimal, and the corresponding cost of the tail subproblem is 7, as shown next to node $C$ in Fig. 1.2.5.

*Original Problem of Length 4*: The possibilities here are (1) start with operation A (cost 5) and then solve optimally the corresponding subproblem of length 3 (cost 8, as computed earlier), a total cost of 13, or (2) start with operation $C$ (cost 3) and then solve optimally the corresponding subproblem of length 3 (cost 7, as computed earlier), a total cost of 10. The second possibility is optimal, and the corresponding optimal cost is 10, as shown next to the initial state node in Fig. 1.2.5.

Note that having computed the optimal cost of the original problem through the solution of all the tail subproblems, we can construct the optimal schedule: we begin at the initial node and proceed forward, each time choosing the optimal operation, i.e., the one that starts the optimal schedule for the corresponding tail subproblem. In this way, by inspection of the graph and the computational results of Fig. 1.2.5, we determine that $CABD$ is the optimal schedule.

### Finding an Optimal Control Sequence by DP

We now state the DP algorithm for deterministic finite horizon problems by translating into mathematical terms the heuristic argument underlying the principle of optimality. The algorithm constructs functions

$$J_N^*(x_N), J_{N-1}^*(x_{N-1}), \cdots, J_0^*(x_0)$$

sequentially, starting from $J_N^*$, and proceeding backwards to $J_{N-1}^*, J_{N-2}^*$, etc. *The value $J_k^*(x_k)$ represents the optimal cost of the tail subproblem that starts at state $x_k$ at time $k$.*

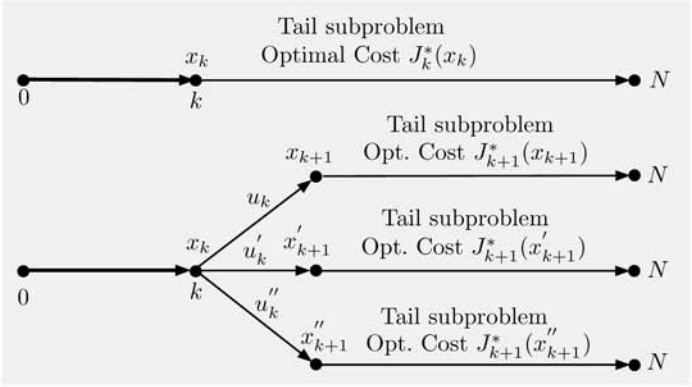


**Figure 1.2.6** Illustration of the DP algorithm. The tail subproblem that starts at $x_k$ at time $k$ minimizes over $\{u_k, \cdots, u_{N-1}\}$ the "cost-to-go" from $k$ to $N$,

$$g_k(x_k, u_k) + \sum_{m=k+1}^{N-1} g_m(x_m, u_m) + g_N(x_N)$$

To solve it, we choose $u_k$ to minimize the (1st stage cost + Optimal tail problem cost) or

$$J_k^*(x_k) = \min_{u_k \in U_k(x_k)} \Big[ g_k(x_k, u_k) + J_{k+1}^*\big(f_k(x_k, u_k)\big) \Big]$$

**DP Algorithm for Deterministic Finite Horizon Problems**

Start with

$$J_N^*(x_N) = g_N(x_N), \qquad \text{for all } x_N \tag{1.3}$$

and for $k = 0, \cdots, N-1$, let

$$J_k^*(x_k) = \min_{u_k \in U_k(x_k)} \Big[ g_k(x_k, u_k) + J_{k+1}^*\big(f_k(x_k, u_k)\big) \Big], \qquad \text{for all } x_k \tag{1.4}$$

The DP algorithm together with the construction of the optimal cost-to-go functions $J_k^*(x_k)$ are illustrated in Fig. 1.2.6. Note that at stage $k$, the calculation in Eq. (1.4) must be done for all states $x_k$ before proceeding to stage $k-1$. The key fact about the DP algorithm is that for every initial state $x_0$, the number $J_0^*(x_0)$ obtained at the last step, is equal to the optimal cost $J^*(x_0)$. Indeed, a more general fact can be shown, namely that for all $k = 0, 1, \cdots, N-1$, and all states $x_k$ at time $k$, we have

$$J_k^*(x_k) = \min_{\substack{u_m \in U_m(x_m) \\ m=k,\cdots,N-1}} J(x_k; u_k, \cdots, u_{N-1}) \tag{1.5}$$

where $J(x_k; u_k, \cdots, u_{N-1})$ is the cost generated by starting at $x_k$ and using subsequent controls $u_k, \cdots, u_{N-1}$:

$$J(x_k; u_k, \cdots, u_{N-1}) = g_N(x_N) + \sum_{t=k}^{N-1} g_t(x_t, u_t) \tag{1.6}$$

Thus, $J_k^*(x_k)$ is the optimal cost for an $(N-k)$-stage tail subproblem that starts at state $x_k$ and time $k$, and ends at time $N$.† Based on the interpretation (1.5) of $J_k^*(x_k)$, we call it the *optimal cost-to-go* from state $x_k$ at stage $k$, and refer to $J_k^*$ as the *optimal cost-to-go function* or *optimal cost function* at time $k$. In maximization problems the DP algorithm (1.4) is written with maximization in place of minimization, and then $J_k^*$ is referred to as the *optimal value function* at time $k$.

Once the functions $J_0^*, \cdots, J_N^*$ have been obtained, we can use a forward algorithm to construct an optimal control sequence $\{u_0^*, \cdots, u_{N-1}^*\}$ and state trajectory $\{x_1^*, \cdots, x_N^*\}$ for the given initial state $x_0$.

**Construction of Optimal Control Sequence $\{u_0^*, \cdots, u_{N-1}^*\}$**

Set

$$u_0^* \in \arg \min_{u_0 \in U_0(x_0)} \Big[ g_0(x_0, u_0) + J_1^*\big(f_0(x_0, u_0)\big) \Big]$$

and

$$x_1^* = f_0(x_0, u_0^*)$$

Sequentially, going forward, for $k = 1, 2, \cdots, N-1$, set

† We can prove this by induction. The assertion holds for $k = N$ in view of the initial condition $J_N^*(x_N) = g_N(x_N)$. To show that it holds for all $k$, we use Eqs. (1.5) and (1.6) to write

$$\begin{aligned} J_k^*(x_k) &= \min_{\substack{u_t \in U_t(x_t) \\ t=k,\cdots,N-1}} \left[ g_N(x_N) + \sum_{t=k}^{N-1} g_t(x_t, u_t) \right] \\ &= \min_{u_k \in U_k(x_k)} \left[ g_k(x_k, u_k) + \min_{\substack{u_t \in U_t(x_t) \\ t=k+1,\cdots,N-1}} \left[ g_N(x_N) + \sum_{t=k+1}^{N-1} g_t(x_t, u_t) \right] \right] \\ &= \min_{u_k \in U_k(x_k)} \Big[ g_k(x_k, u_k) + J_{k+1}^*\big(f_k(x_k, u_k)\big) \Big] \end{aligned}$$

where for the last equality we use the induction hypothesis. A subtle mathematical point here is that, through the minimization operation, the functions $J_k^*$ may take the value $-\infty$ for some $x_k$. Still the preceding induction argument is valid even if this is so. The books [BeT96] and [Ber18a] address DP algorithms that allow infinite values in various operations such as minimization.

$$u_k^* \in \arg \min_{u_k \in U_k(x_k^*)} \Big[ g_k(x_k^*, u_k) + J_{k+1}^*\big(f_k(x_k^*, u_k)\big) \Big] \qquad (1.7)$$

and

$$x_{k+1}^* = f_k(x_k^*, u_k^*)$$

Note an interesting conceptual division of the optimal control sequence construction: there is "off-line training" to obtain $J_k^*$ by precomputation [cf. Eqs. (1.3)-(1.4)], which is followed by real-time "on-line play" to obtain $u_k^*$ [cf. Eq. (1.7)]. This is analogous to the two algorithmic processes described in Section 1.1 in connection with chess and backgammon.

Figure 1.2.5 traces the calculations of the DP algorithm for the scheduling Example 1.2.1. The numbers next to the nodes, give the corresponding cost-to-go values, and the thick-line arcs give the construction of the optimal control sequence using the preceding algorithm.

### DP Algorithm for General Discrete Optimization Problems

We have noted earlier that discrete deterministic optimization problems, including challenging combinatorial problems, can be typically formulated as DP problems by breaking down each feasible solution into a sequence of decisions/controls, as illustrated with the scheduling Example 1.2.1. This formulation often leads to an intractable DP computation because of an exponential explosion of the number of states as time progresses. However, a DP formulation brings to bear approximate DP methods, such as rollout and others, to be discussed shortly, which can deal with the exponentially increasing size of the state space. We illustrate the reformulation by an example and then generalize.

**Example 1.2.2 (The Traveling Salesman Problem)**

An important model for scheduling a sequence of operations is the classical traveling salesman problem. Here we are given $N$ cities and the travel time between each pair of cities. We wish to find a minimum time travel that visits each of the cities exactly once and returns to the start city. To convert this problem to a DP problem, we form a graph whose nodes are the sequences of $k$ distinct cities, where $k = 1, \cdots, N$. The $k$-city sequences correspond to the states of the $k$th stage. The initial state $x_0$ consists of some city, taken as the start (city A in the example of Fig. 1.2.7). A $k$-city node/state leads to a $(k+1)$-city node/state by adding a new city at a cost equal to the travel time between the last two of the $k+1$ cities; see Fig. 1.2.7. Each sequence of $N$ cities is connected to an artificial terminal node $t$ with an arc of cost equal to the travel time from the last city of the sequence to the starting city, thus completing the transformation to a DP problem.

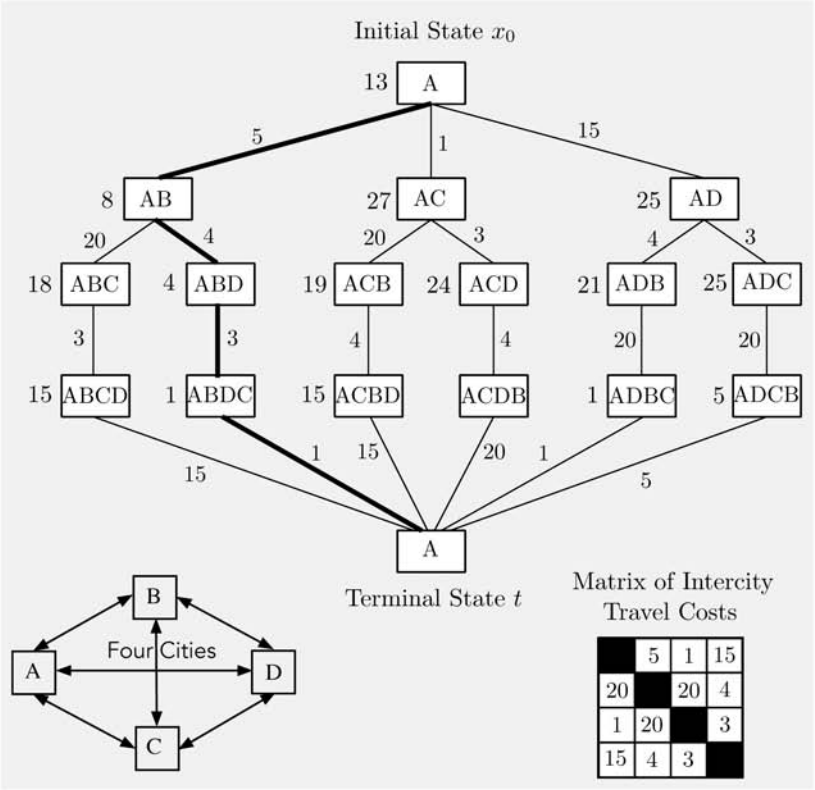


**Figure 1.2.7** Example of a DP formulation of the traveling salesman problem. The travel times between the four cities A, B, C, and D are shown in the matrix at the bottom. We form a graph whose nodes are the $k$-city sequences and correspond to the states of the $k$th stage, assuming that A is the starting city. The transition costs/travel times are shown next to the arcs. The optimal costs-to-go are generated by DP starting from the terminal state and going backwards towards the initial state, and are shown next to the nodes. There is a unique optimal sequence here (ABDCA), and it is marked with thick lines. The optimal sequence can be obtained by forward minimization [cf. Eq. (1.7)], starting from the initial state $x_0$.

The optimal costs-to-go from each node to the terminal state can be obtained by the DP algorithm and are shown next to the nodes. Note, however, that the number of nodes grows exponentially with the number of cities $N$. This makes the DP solution intractable for large $N$. As a result, large traveling salesman and related scheduling problems are typically addressed with approximation methods, some of which are based on DP, and will be discussed in future chapters.

Let us now extend the ideas of the preceding example to the general

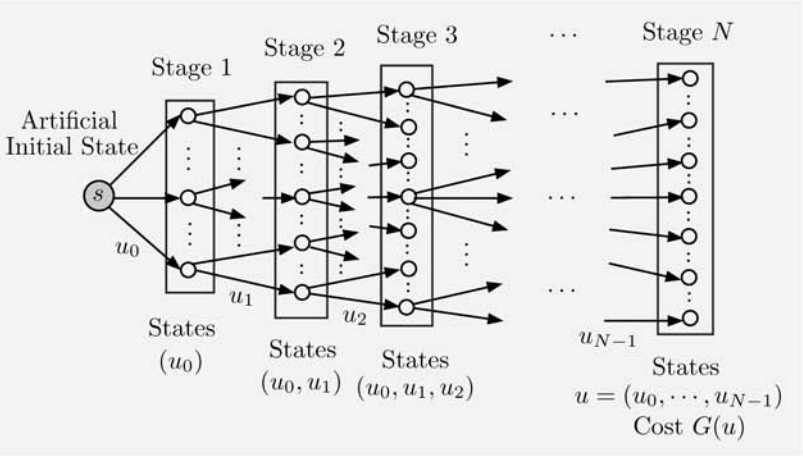


**Figure 1.2.8** Formulation of a discrete optimization problem as a DP problem with $N$ stages. There is a cost $G(u)$ only at the terminal stage on the arc connecting an $N$-solution $u = (u_0, \cdots, u_{N-1})$ upon reaching the terminal state. Note that there is only one incoming arc at each node.

discrete optimization problem:

$$\begin{aligned} &\text{minimize} \quad G(u) \\ &\text{subject to} \quad u \in U \end{aligned}$$

where $U$ is a finite set of feasible solutions and $G(u)$ is a cost function. We assume that each solution $u$ has $N$ components; i.e., it has the form $u = (u_0, \cdots, u_{N-1})$, where $N$ is a positive integer. We can then view the problem as a sequential decision problem, where the components $u_0, \cdots, u_{N-1}$ are selected one-at-a-time. A $k$-tuple $(u_0, \cdots, u_{k-1})$ consisting of the first $k$ components of a solution is called a *$k$-solution*. We associate $k$-solutions with the $k$th stage of the finite horizon DP problem shown in Fig. 1.2.8. In particular, for $k = 1, \cdots, N$, we view as the states of the $k$th stage all the $k$-tuples $(u_0, \cdots, u_{k-1})$. For stage $k = 0, \cdots, N-1$, we view $u_k$ as the control. The initial state is an artificial state denoted $s$. From this state, by applying $u_0$, we may move to any "state" $(u_0)$, with $u_0$ belonging to the set

$$U_0 = \big\{\tilde{u}_0 \mid \text{there exists a solution of the form } (\tilde{u}_0, \tilde{u}_1, \cdots, \tilde{u}_{N-1}) \in U\big\}$$

Thus $U_0$ is the set of choices of $u_0$ that are consistent with feasibility.

More generally, from a state $(u_0, \cdots, u_{k-1})$, we may move to any state of the form $(u_0, \cdots, u_{k-1}, u_k)$, upon choosing a control $u_k$ that belongs to the set

$$\begin{aligned} U_k(u_0, \cdots, u_{k-1}) = \big\{u_k \mid \text{for some } \overline{u}_{k+1}, \cdots, \overline{u}_{N-1} \text{ we have} \\ (u_0, \cdots, u_{k-1}, u_k, \overline{u}_{k+1}, \cdots, \overline{u}_{N-1}) \in U\big\} \end{aligned}$$

These are the choices of $u_k$ that are consistent with the preceding choices $u_0, \cdots, u_{k-1}$, and are also consistent with feasibility. The last stage corresponds to the $N$-solutions $u = (u_0, \cdots, u_{N-1})$, and the terminal cost is $G(u)$; see Fig. 1.2.8. All other transitions in this DP problem formulation have cost 0.

Let

$$J_k^*(u_0, \cdots, u_{k-1})$$

denote the optimal cost starting from the $k$-solution $(u_0, \cdots, u_{k-1})$, i.e., the optimal cost of the problem over solutions whose first $k$ components are constrained to be equal to $u_0, \cdots, u_{k-1}$. The DP algorithm is described by the equation

$$J_k^*(u_0, \cdots, u_{k-1}) = \min_{u_k \in U_k(u_0, \cdots, u_{k-1})} J_{k+1}^*(u_0, \cdots, u_{k-1}, u_k)$$

with the terminal condition

$$J_N^*(u_0, \cdots, u_{N-1}) = G(u_0, \cdots, u_{N-1})$$

This algorithm executes backwards in time: starting with the known function $J_N^* = G$, we compute $J_{N-1}^*$, then $J_{N-2}^*$, and so on up to computing $J_0^*$. An optimal solution $(u_0^*, \cdots, u_{N-1}^*)$ is then constructed by going forward through the algorithm

$$u_k^* \in \arg \min_{u_k \in U_k(u_0^*, \cdots, u_{k-1}^*)} J_{k+1}^*(u_0^*, \cdots, u_{k-1}^*, u_k), \quad k = 0, \cdots, N-1 \quad (1.8)$$

first compute $u_0^*$, then $u_1^*$, and so on up to $u_{N-1}^*$; cf. Eq. (1.7).

Of course here the number of states typically grows exponentially with $N$, but we can use the DP minimization (1.8) as a starting point for the use of approximation methods. For example we may try to use approximation in value space, whereby we replace $J_{k+1}^*$ with some suboptimal $\tilde{J}_{k+1}$ in Eq. (1.8). One possibility is to use as

$$\tilde{J}_{k+1}(u_0^*, \cdots, u_{k-1}^*, u_k)$$

the cost generated by a heuristic method that solves the problem suboptimally with the values of the first $k+1$ decision components fixed at $u_0^*, \cdots, u_{k-1}^*, u_k$. This is the *rollout algorithm*, which is a very simple and effective approach for approximate combinatorial optimization. It will be discussed in the next section, and in Chapters 2 and 3. It will be related to the method of policy iteration and self-learning ideas in Chapter 5.

Let us finally note that while we have used a general cost function $G$ and constraint set $C$ in our discrete optimization model of this section, in many problems $G$ and/or $C$ may have a special structure, which is consistent with a sequential decision making process. The traveling salesman Example 1.2.2 is a case in point, where $G$ consists of $N$ components (the intercity travel costs), one per stage.