Operations Research Computer Science Interfaces Series

Abhijit Gosavi

Simulation-Based Optimization

Parametric Optimization Techniques and Reinforcement Learning

Second Edition

OR INTERFACES CS

Springer

Abhijit Gosavi

Simulation-Based Optimization

Parametric Optimization Techniques and Reinforcement Learning

Second Edition

Springer

Abhijit Gosavi
Department of Engineering Management
and Systems Engineering
Missouri University of Science
and Technology
Rolla, MO, USA

ISSN 1387-666X
ISBN 978-1-4899-7490-7 ISBN 978-1-4899-7491-4 (eBook)
DOI 10.1007/978-1-4899-7491-4
Springer New York Heidelberg Dordrecht London

Library of Congress Control Number: 2014947352

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

To my parents:
Ashok D. Gosavi
and
Sanjivani A. Gosavi

Preface

This book is written for students and researchers in the field of industrial engineering, computer science, operations research, management science, electrical engineering, and applied mathematics. The aim is to introduce the reader to a *subset* of topics on simulation-based optimization of large-scale and complex stochastic (random) systems. Our goal is *not* to cover all the topics studied under this broad field. Rather, it is to expose the reader to a *selected* set of topics that have recently produced breakthroughs in this area. As such, much of the material focusses on some of the key recent advances.

Those working on problems involving *stochastic* **discrete-event** systems and optimization may find useful material here. Furthermore, the book is self-contained, but only to an extent; a background in elementary college calculus (basics of differential and integral calculus) and linear algebra (matrices and vectors) is expected. Much of the book attempts to cover the topics from an intuitive perspective that appeals to the engineer.

In this book, we have referred to any stochastic optimization problem related to a discrete-event system that can be solved with computer simulation as a **simulation-optimization** problem. Our focus is on those simulation-optimization techniques that do not require any a priori knowledge of the structure of the objective function (loss function), i.e., closed form of the objective function or the underlying probabilistic structure. In this sense, the techniques we cover are **model-free**. The techniques we cover do, however, require all the information typically required to construct a simulation model of the discrete-event system.

Although the science underlying simulation-based optimization has a rigorous mathematical foundation, our development in the initial chapters is based on intuitively appealing explanations of the major concepts. It is only in Chaps. 9–11 that we adopt a somewhat more rigorous approach, but even there our aim is to prove only those results that provide intuitive insights for the reader.

Broadly speaking, the book has two parts: (1) parametric (static) optimization and (2) control (dynamic) optimization. While the sections on control optimization are longer and a greater portion of the book is devoted to control optimization, the intent is not in any way to diminish the significance of parametric optimization. The field of control optimization with simulation has benefited from work in numerous communities, e.g., computer science and electrical engineering, which perhaps explains the higher volume of work done in that field. But the field of parametric optimization has also attracting significant interest in recent times and it is likely that it will expand in the coming years.

By *parametric* optimization, we refer to static optimization in which the goal is to find the values of *parameters* that maximize or minimize a function (the objective or loss function), usually a function of those parameters. By *control* optimization, we refer to those dynamic optimization problems in which the goal is to find an optimal *control* (action) in each state visited by a system. The book's goal is to describe these models and the associated optimization techniques in the context of stochastic (random) systems. While the book presents some classical paradigms to develop the background, the focus is on recent research in both parametric and control optimization. For example, exciting, recently developed techniques such as *simultaneous perturbation* (parametric optimization) and *reinforcement learning* (control optimization) are two of the main topics covered. We also note that in the context of control optimization, we focus only on those systems which can be modeled by Markov or semi-Markov processes.

A common thread running through the book is naturally that of simulation. Optimization techniques considered in this book require simulation as opposed to explicit mathematical models. Some special features of this book are:

1. An accessible introduction to **reinforcement learning** and parametric optimization techniques.

2. A step-by-step description of several algorithms of simulation-based optimization.

3. A clear and simple introduction to the methodology of **neural networks.**

4. A special account of semi-Markov control via simulation.

5. A **gentle** introduction to convergence analysis of some of techniques.

6. **Computer programs** for many algorithms of simulation-based optimization, which are online.

The background material in discrete-event simulation from Chap. 2 is at a very elementary level and can be skipped without loss of continuity by readers familiar with this topic; we provide it here for those readers (possibly not from operations research) who may not have been exposed to it. Links to computer programs have been provided in the appendix for those who want to apply their own ideas or perhaps test their own algorithms on some test-beds. Convergence-related material in Chaps. 9–11 is for those interested in the mathematical roots of this science; it is intended only as a form of an introduction to mathematically sophisticated material found in other advanced texts.

Rolla, USA Abhijit Gosavi

Acknowledgments

I would like to acknowledge a number of people without whose support this book could never have been written. I am very grateful to Gary Folven and Carolyn Ford at Kluwer Academic for their encouragement and hard work in editing the book. I would also like to express my deep appreciation for the enthusiasm and support extended to me by my former department Chair, Jane M. Fraser, at Colorado State University-Pueblo.

Over the last few years, a number of individuals have helped me either by pointing to useful references, sending their work and helping me understand their work better, or reviewing the manuscript. I thank Jayant Rajgopal (University of Pittsburgh), Ramesh Sharda (Oklahoma State University), James Spall (Johns Hopkins University), Kumapati Narendra (Yale University), Alice Smith (Auburn University), Janet Barnett (Colorado State University-Pueblo), Prasad Tadepalli (Oregon State University), Tito Homem-de-Mello (Ohio State University), Vivek S. Borkar (the Tata Institute of Fundamental Research, Mumbai), Sigrún Andradóttir (Georgia Institute of Technology), and Barry Nelson (Northwestern University).

Parts of this book are based on notes prepared for material needed in graduate-level courses that I taught at Colorado State University-Pueblo. The material was developed for teaching stochastic processes, simulation, classical optimization theory, heuristic optimization, and statistics. Many students patiently struggled through my notes, and many made useful suggestions for which I am thankful. Many graduate students in Colorado State University-Pueblo read parts of the manuscript and made helpful comments: I thank Tolga Tezcan, Lisa Haynes, Emrah Ozkaya, and Tal Wigdor.

I would also like to thank my former teachers at the University of South Florida from whom I have a learned a lot: Tapas K. Das, Sridhar Mahadevan (now at the University of Massachusetts, Amherst), Sudeep Sarkar, Suresh K. Khator, Michael X. Weng, Geoffrey O. Okogbaa, and William A. Miller.

Finally, I wish to thank my parents, siblings, and other members of my family, Anuradha and Rajeev Sondur, Sucheta Gosavi, and Aparna and Milind Bhagavat, for their unconditional support over the years.

Second Edition: I begin by thanking Matthew Amboy of Springer for his support and encouragement. The first edition contained errors (typos and mathematical), which have been corrected to the best of my abilities. Some elementary material on probability theory has been shortened and placed in the Appendix. The computer programs that required random numbers have been rewritten to include random number generators from gcc compilers, allowing me to place them online now.

The book has undergone a major revision. Overall, some material has been expanded in its scope, while some material has been reorganized to increase clarity of presentation. Broadly speaking, the major changes for control optimization are: a refined discussion on semi-Markov decision processes (SMDPs) in dynamic programming (DP) and reinforcement learning (RL), coverage of SARSA and conservative approximate policy iteration (API), expanded coverage of function approximation, and a more rigorous discussion on RL convergence via ODEs (ordinary differential equations), along with a discussion on the stochastic shortest path (SSP) and two time-scale updating. To be more specific, I have now covered in greater depth the convergence theory of Q-Learning (discounted reward MDPs and SMDPs), Q-P-Learning (discounted reward MDPs and average reward SMDPs), conservative API (discounted reward MDPs), and R-SMART (average reward SMDPs). The chapter on learning automata for controls has been expanded to cover the actor critic in greater depth. In the area of parametric optimization, I have expanded the discussion on stochastic adaptive search (SAS), including a discussion on backtracking adaptive search, nested partitions, and the stochastic ruler. The framework of Markov chains underlying SAS has now been expanded in scope to show convergence of some algorithms belonging to the SAS family.

Books on DP by D.P. Bertsekas, MDPs by M.L. Puterman, and NDP by D.P. Bertsekas and J.N. Tsitsiklis have deeply influenced my understanding of this area. Via personal communication, Zelda Zabinsky has helped clarify many of the finer aspects of her research covered

in her excellent book. I am also grateful to Paul J. Werbós at the National Science Foundation, who provided financial and moral support and introduced me to the origins of adaptive critics.

Many readers of this book, including students, colleagues, and some friendly strangers, have found typos and made suggestions: I thank Mandar Purohit, Cai Chengtao, Ganesh Subramaniam, Shantanu Phatakwala, Michael Fu, Matthew Reindorp, Ivan Guardiola, Shuva Ghosh, Marc-André Carbonneau, and Leyuan Shi. Finally, I want to thank my wife Apoorva and my daughter Josna for being the wonderful people they are!

Abhijit A. Gosavi
Rolla, Missouri, USA

I take responsibility for errors in the book, but neither me nor the publisher is responsible/liable for any misuse of material presented here. If you have any suggestions, do not hesitate to contact me at: gosavia@mst.edu

Contents

List of Figures

List of Tables

Chapter 1

BACKGROUND

1. Motivation

This book seeks to introduce the reader to the rapidly evolving subject called simulation-based optimization. This is not a very young topic, because from the time computers started making an impact on scientific research and it became possible to analyze random systems with computer programs that generated random numbers, engineers have always wanted to *optimize* systems using simulation models. However, it is only recently that noteworthy success in realizing this objective has been met in practice.

Path-breaking work in computational operations research in areas such as non-linear programming (simultaneous perturbation) and dynamic programming (reinforcement learning) has now made it possible for us to use simulation in conjunction with optimization techniques. This has given simulation the kind of power that it did not have in the past, when simulation optimization was usually treated as a synonym for the relatively sluggish (although robust) response surface method.

The power of computers has increased dramatically over the years, of course, and it continues to increase. This has helped increase the speed of running computer programs, and has provided an incentive to study simulation-based optimization. But the over-riding factor in favor of simulation optimization in recent times is the remarkable research that has taken place in various areas of computational operations research. We mean research that has either given birth to new

A. Gosavi, *Simulation-Based Optimization*, Operations Research/Computer Science Interfaces Series 55, DOI 10.1007/978-1-4899-7491-4_1

optimization techniques *more compatible with simulation*, or in many cases research that has generated modified versions of old optimization techniques that can be combined more elegantly with simulation. It is important to point out that the success of these new techniques has to be also attributed to their deep mathematical roots in operations research. These roots have helped lead simulation optimization onto a new path, where it has attacked problems previously considered intractable.

Surprisingly, the success stories have been reported in widely different, albeit related, areas of operations research. Not surprisingly, all of these success stories have a natural connection; the connecting thread is an adroit integration of computer simulation with an optimization technique. Hence, we believe that there is a need for a book that presents some of the recent advances in this field. Simulation-based optimization, it is expected, will achieve a prominent status within the field of stochastic optimization in the coming years.

1.1. Main Branches

Optimization problems in large-scale and complex stochastic scenarios broadly speaking belong to two main branches of operations research:

1. **Parametric** optimization (also called **static** optimization).
2. **Control** optimization (also called **dynamic** optimization).

In general, parametric optimization is performed to find values of *a set of parameters* (or *decision variables*) that optimize some performance measure (generally, minimize a cost or maximize a reward). On the other hand, control optimization refers to finding *a set of actions* to be taken in the different states that a system visits, such that the actions selected optimize some performance measure of the system (again, minimize a cost or maximize a reward). Classically, in the field of operations research, parametric optimization is performed using mathematical programming, e.g., linear, non-linear, and integer programming. Control optimization on the other hand is generally performed via dynamic programming, but sometimes via mathematical programming as well.

Parametric optimization is often called *static* optimization, because the solution is a set of "static" parameters for all states. Control optimization is often called *dynamic* optimization because the solution depends on the state, which changes dynamically; for every state, we may have a different solution.

1.2. Difficulties with Classical Optimization

The focus of this book is on *discrete-event stochastic* systems, where the size and complexity of the problem prevents a naive application of classical optimization methods. This needs a more detailed explanation.

Formulating an objective function can oftentimes prove to be difficult in parametric optimization problems encountered in the stochastic setting. Usually, the objective function in this setting is non-linear, with multiple integrals and probabilistic elements. It goes without saying that the larger the number of random variables in a system the more complex the system usually is for deriving closed-form expressions for the objective function. In the control optimization scenario, if one is to use classical dynamic programming, one needs the transition probability function, which is difficult to evaluate. The difficulties with this function coupled with a possibly large number of states lead to the well-known *curses* of dynamic programming: the curse of modeling and the curse of dimensionality. The larger the number of states and input random variables (and greater their inter-dependencies), the more difficult it is to break these curses.

Short of a closed form for the objective function in the parametric setting and the transition probability function in the control setting, in a stochastic problem, one often turns to simulation for performance evaluation. However, performance evaluation via simulation is time-consuming, despite advances in computing power. Hence, the traditional approaches to capture the behavior of the objective function (via response surface methods) or the transition probability function (maximum likelihood estimation) required an enormous amount of computer time with the simulator, either due to long trajectories or multiple replications or both. If the simulation time required is impractically large, obviously, simulation-based methods for optimization are ruled out. What is needed (or rather was needed and is becoming a reality now) are optimization techniques that require a manageable amount of simulation time and produce optimal or near-optimal solutions. We now discuss some recent advances in this area.

1.3. Recent Advances in Simulation Optimization

Our intent in this book is to focus on a subset of topics in simulation optimization that revolve around the recent advances in this field. Here we discuss one technique each from parametric optimization and from control optimization.

Parametric optimization. The fact that function evaluations via simulation can take considerable amounts of time has stimulated interest in non-linear programming techniques that can work without an excessively large number of function evaluations. This is so because, as stated above, even a one-time numerical evaluation of the objective function of a complex stochastic system via simulation is computationally expensive. An example of a method that takes a long time but works with simulation-based function evaluations is the age-old gradient-descent (or ascent) algorithm with regular finite differences. This technique requires a relatively large number of function evaluations in each iteration and takes a long time to generate solutions.

A breakthrough in this field is Spall's *simultaneous perturbation* technique [280] (1992). In comparison to other non-linear programming methods, it requires few function evaluations, and as such has a relatively low computational burden. Several extensions of this technique are now being researched upon.

Control optimization. As stated above, stochastic dynamic programming, the main tool of control optimization, suffers from the twin curses of *modeling* and *dimensionality.* In complex large-scale systems, the transition probabilities are hard to find for complex systems (curse of modeling), and the number of states and the transition probability matrices become too huge (curse of dimensionality). For example, a problem with one thousand states, which is a very small problem in the real-world context, has *one million* transition probabilities just for one action. Since these probabilities are usually difficult to store, it is difficult to process them for generating a solution.

This has inspired research in methods that work within simulators but avoid the generation of transition probabilities. An example of one such algorithm is the *Q-Learning* algorithm [312] (1989). This and other related algorithms have given birth to a field called *reinforcement learning.*

What is different about these methods? Much of the literature in stochastic optimization from the 1960s and 1970s focused on developing exact mathematical models for the problem at hand. This oftentimes required *making simplifying assumptions about the system.* A commonly made assumption in many models is the use of the exponential distribution for random variables in the system. This assumption leads to elegant closed-form mathematics but often ignores real-life considerations. Without closed-forms for objective functions, it has been difficult to optimize. *Traditional* simulation-optimization methods, e.g., response surfaces, were known to work with any given distribution for the random variables. However, as mentioned above, they required a large amount of computational time, and hence closed-form methods were often preferred to them. Some of the recent advances in simulation-based optimization seek to change this perception about simulation-based optimization.

2. Goals and Limitations

While writing the book, we have had to obviously make numerous decisions related to what to include and what not to include. We first describe what our main goals are and then discuss the topics that we do *not* cover here.

2.1. Goals

The main goal of this book is to introduce the reader to a selection of topics within simulation-based optimization of discrete-event systems, concentrating on parametric optimization and control optimization. We have elected to cover only a *subset* of topics in this vast field; some of the key topics that we cover are: response surfaces with neural networks, simultaneous perturbation, meta-heuristics (for simulation optimization), reinforcement learning, and learning automata. Theoretical presentation of some algorithms has been supplemented by engineering case studies. The intent behind their presentation is to demonstrate the use of simulation-based optimization on real-life problems. We also hope that from reading the chapters related to convergence, the reader will gain an understanding of the theoretical methods used to mathematically establish that a given algorithm works. Overwhelming the reader with mathematical convergence arguments (e.g., theorems and proofs) was not our intention, and therefore material of that nature is covered towards the end in separate chapters.

A central theme in this book is the development of optimization models that can be combined easily with simulators, in order to optimize complex systems for which analytical models are not easy to construct. Consequently, the focus is on the optimization model: in the context of parametric (static) optimization, the underlying system is assumed to have no special structural properties, while in the context of control (dynamic) optimization, the underlying system is assumed to be driven by Markov chains.

2.2. Limitations

Firstly, we note that we have restricted our discussion to **discrete-event** systems. Therefore, systems modeled by continuous-event dynamics, e.g., Brownian motion or deterministic systems modeled by differential equations, are outside the scope of this text. Secondly, within control optimization, we concentrate *only* on systems governed by Markov chains, in particular the Markov and semi-Markov decision processes. Hence, systems modeled by partially observable Markov processes and other kinds of jump processes are not studied here. Finally, and very importantly, we have not covered any **model-based** techniques, i.e., techniques that exploit the structure of the problem. Usually, model-based techniques assume some prior knowledge of the problem structure, which could either be the objective function's properties (in parametric optimization) or the transition probability functions (in control optimization). Rather, our focus is on **model-free** techniques that require no prior knowledge of the system (other than, of course, what is needed to construct a simulation model).

It is also important to point out that the algorithms presented in this book are restricted to systems for which (a) the distributions of random variables are known (or can be determined with data collection) and (b) to systems that are known to reach steady state. Non-stationary systems or unstable systems that never reach steady state, perhaps due to the occurrence of rare events, are not considered in this book. Finally, we note that any simulation model continues to be accurate only if the underlying random variables continue to follow the distributions assumed. If the distributions change, our simulation-optimization model will break down and should not be used.

3. Notation

We now define much of the notation used in this book. The discussion is in general terms and refers to some conventions that we have adopted. Vector notation has been avoided *as much as possible*; although it is more compact and elegant in comparison to component notation, we believe that component notation, in which all quantities are scalar, is usually easier to understand.

3.1. Some Basic Conventions

The symbol $\forall i$ will denote: for all possible values of i. The notation

$$\sum_i p(i)$$

will denote a *summation* over all values of i, while

$$\prod_i p(i)$$

will denote a *product* over all values of i.

Also, $\mathsf{E}[.]$ will denote the expectation of the quantity in the square brackets, while $\mathsf{P}[.]$ will denote probability of the event inside the square brackets. $\{\ldots\}$ will denote a set of the elements inside the curly braces.

3.2. Vector Notation

In this book, a vector quantity will always have an arrow ($\rightarrow$) placed above it. This convention should distinguish a scalar from a vector. From our experience, not making this distinction can create a great deal of confusion to the beginner. Hence, for example, $\vec{x}$ will denote a vector whereas x will denote a scalar.

For the most part, when we mean vector, we will mean column vector in this book. The ith component of a vector $\vec{x}$ will be denoted by $x(i)$. A column vector will also be denoted, at several places in the book, with the following notation. For example,

$$(x(1), x(2), x(3)) \text{ or } [x(1)\ x(2)\ x(3)]^T$$

will denote a column vector with three elements, where T denotes a transpose. Some other examples are:

$$(1, 4, 6) \text{ and } (3, 7).$$

Note that (a, b) may also denote the open interval between scalars a and b. Where we mean an interval and not a vector, we will spell out the definition clearly. A closed interval will be denoted by:

$$[a, b].$$

The notation $||\vec{x}||$ will denote a **norm** of the vector $\vec{x}$. We now discuss a number of norms that will be needed in the book.

Max norm. The notation $||\vec{x}||_\infty$ is often used to denote the **max** norm of the vector $\vec{x}$ and is defined as:

$$||\vec{x}||_\infty = \max_i |x(i)|,$$

where $|a|$ denotes the absolute value of a. The max norm will be equivalent to the **sup** norm or the **infinity** norm (for the analysis in this book). Whenever $||.||$ is used without a subscript, it will be assumed to equal the max norm.

Euclidean norm. The notation $||\vec{x}||_2$ will denote the **Euclidean** norm of the vector $\vec{x}$ and is defined as:

$$||\vec{x}||_2 = \sqrt{\sum_i [x(i)]^2}.$$

Manhattan norm. The notation $||\vec{x}||_1$ will denote the **Manhattan** norm of the vector $\vec{x}$ and is defined as:

$$||\vec{x}||_1 = \sum_i |x(i)|.$$

3.3. Notation for Matrices

A matrix will be printed in **boldface**. For example:

$$\mathbf{A} = \begin{bmatrix} 2 & 3 & 7 \\ 4 & 2 & 1 \end{bmatrix}. \tag{1.1}$$

The transpose of a matrix **A** will be denoted by $\mathbf{A}^T$. Thus, using the definition given in Eq. (1.1),

$$\mathbf{A}^T = \begin{bmatrix} 2 & 4 \\ 3 & 2 \\ 7 & 1 \end{bmatrix}.$$

I will denote the identity matrix .

3.4. Notation for *n*-tuples

The notation that we have used for n-tuples is distinct from that used for vectors. For instance $\hat{x}$ will denote an n-tuple. An n-tuple may or may not be a vector. An n-tuple may also be denoted with the following notation:

$$(a_1, a_2, \ldots, a_n).$$

The notation, nth, will denote n^{th}. For instance, if $n = 5$ then nth will denote 5th.

3.5. Notation for Sets

Calligraphic letters, such as $\mathcal{A}, \mathcal{B}, \mathcal{C}, \ldots, \mathcal{X}$, will invariably denote sets in this book. The notation $|\mathcal{A}|$ will denote the **cardinality** of the set $\mathcal{A}$ (that is the number of elements in the set), but $|a|$ will denote the **absolute value** of the scalar a. Sets may also be denoted with curly braces in the following form:

$$\{1, 2, 3, 4\} \text{ and } \{1, 3, 5, 7, \ldots, \}.$$

In the above, both representations are sets.

A special set is the real number line, which will be represented by $\Re$. It denotes the set of all real numbers. $\Re$ is also the specific case of a Euclidean space in one dimension. In general, Euclidean space in the nth dimension will be represented by $\Re^n$.

The notation L^2 may denote L raised to the second power if L is a scalar quantity; it may denote something very different if L is a transformation operator. The actual meaning will become clear from the context. Also, the superscript has been used in a number of places to denote the iteration in a given algorithm. In such cases, also, the superscript does not represent the power. Hence, the superscript should not be assumed to be the power, unless it is explicitly defined to be a power.

3.6. Notation for Sequences

The notation $\{a^n\}_{n=1}^{\infty}$ will represent an infinite sequence whose nth term is a^n. For instance, if a sequence is defined as follows:

$$\{a^n\}_{n=1}^{\infty} = \{10, 20, 30, \ldots\},$$

then $a^3 = 30$.

3.7. Notation for Transformations

A transformation is an operation carried out on a vector. The result of a transformation is a vector or a scalar. An example of a transformation (or transform, as it is sometimes referred to) is:

$$x^{k+1} \leftarrow 3x^k + 2y^k, \qquad y^{k+1} \leftarrow 4x^k + 6y^k. \tag{1.2}$$

Suppose a vector starts out at $\left(x^0, y^0\right) = (5, 10)$. When the transformation defined in (1.2) is applied on this vector, we have:

$$x^1 = 35, \text{ and } y^1 = 80.$$

Equation (1.2) is often abbreviated as:

$$\left(x^{k+1}, y^{k+1}\right) = T\left(\left(x^k, y^k\right)\right),$$

where T denotes the transform. T is understood as T^1. Two applications of T is written as T^2, and so on. For example:

$$\left(x^2, y^2\right) = T^2\left(\left(x^0, y^0\right)\right),$$

and

$$\left(x^5, y^5\right) = T\left(\left(x^4, y^4\right)\right) = T^2\left(\left(x^3, y^3\right)\right).$$

3.8. Max, Min, and Arg Max

The following notation will be encountered frequently.

$$x = \max_{i \in \mathcal{S}} [a(i)].$$

This means that x equals the *maximum* of the values that a can assume. So if set $\mathcal{S}$ is defined as below:

$$\mathcal{S} = \{1, 2, 3\},$$

and

$$a(1) = 1, a(2) = 10, \text{ and } a(3) = -2,$$

then $x = 10$. Similarly, min will be used in the context of the *minimum* value. Now read the following notation carefully.

$$y = \arg\max_{i \in \mathcal{S}} [a(i)].$$

Here, y denotes the *argument* or the *element index* associated with the maximum value. So, if set $\mathcal{S}$ and the values of a are defined as above, then

$$y = 2.$$

so that $a(y)$ is the maximum value for a. It is to be noted that arg min has a similar meaning in the context of the minimum.

In Table 1.1, we present a list of acronyms and abbreviations that we have used in this book.

4. Organization

The rest of this book is organized as follows. Chap. 2 covers basic concepts related to discrete-event simulation. Chap. 3 is meant to present an overview of optimization with simulation. Chap. 4 deals with the response surface methodology, which is used in conjunction with simulation optimization. This chapter also presents the topic of neural networks in some detail. Chap. 5 covers the main techniques for parametric optimization with simulation. Chap. 6 discusses the classical theory of stochastic dynamic programming. Chap. 7 focuses on reinforcement learning. Chap. 8 covers automata theory in the context of solving Markov and Semi-Markov decision problems. Chap. 9 deals with some fundamental concepts from mathematical analysis. These concepts will be needed for understanding the subsequent chapters on convergence. Convergence issues related to parametric optimization methods are presented in Chap. 10, while Chap. 11 discusses

Table 1.1. A list of acronyms and abbreviations used in the book

Acronym/abbreviation	Full name
AGV	Automated (Automatic) Guided Vehicle
Backprop	Backpropagation
BAS	Backtracking Adaptive Search
cdf	cumulative distribution function
CF	Contracting Factor
CTMDP	Continuous Time Markov Decision Problem
DMP	Decision-Making Process
DARE	Displacement Adjusted REvenue
DAVN	Displacement Adjusted Virtual Revenue
DP	Dynamic Programming
DeTSMDP	Deterministic Time Semi-Markov Decision Problem
EMSR	Expected Marginal Seat Revenue
GLIE	Greedy in the Limit with Infinite Exploration
LAST	Learning Automata Search Technique
LP	Linear Program
LSTD	Least Squares Temporal Difference
MCAT	Markov Chain Automata Theory
MDP	Markov Decision Problem or Process
NP	Natural Process
ODE	Ordinary Differential Equation
pdf	probability density function
pmf	probability mass function
Q-Learning	Learning with Q factors (approximate value iteration)
Q-P-Learning	Learning with Q and P factors (approximate policy iteration)
RL	Reinforcement Learning
RSM	Response Surface Method or Methodology

(continued)

Table 1.1. (continued)

Acronym/abbreviation	Full name
RVI	Relative Value Iteration
R-SMART	Relaxed Semi-Markov Average Reward Technique
RTDP	Real Time Dynamic Programming
SAS	Stochastic Adaptive Search
SMDP	Semi-Markov Decision Problem or Process
SSE	Sum of Squared Errors
SSP	Stochastic Shortest-Path Problem
TD	Temporal Difference(s)
TPM	Transition Probability Matrix
TRM	Transition Reward Matrix
TTM	Transition Time Matrix
URS	UnRestricted in Sign
WH	Widrow Hoff

the convergence theory of stochastic dynamic programming and reinforcement learning. The book concludes with Chap. 12 which presents an overview of some case studies of simulation optimization from the existing literature. Some additional material including basics on probability theory and links to computer programs can be found at the following website [121]:

`http://web.mst.edu/~gosavia/bookcodes.html`

Chapter 2

SIMULATION BASICS

1. Chapter Overview

This chapter has been written to introduce the topic of **discrete-event** simulation. To comprehend the material presented in this chapter, some background in the theory of probability is needed, some of which is in the Appendix. Two of the main topics covered in this chapter are random number generation and simulation modeling of random systems. Readers familiar with this material can skip this chapter without loss of continuity.

2. Introduction

A system is usually defined as a collection of entities that interact with each other. A simple example is the queue that forms in front of a teller in a bank: see Fig. 2.1. The entities in this system are the people (customers), who arrive to get served, and the teller (server), who provides service.

The *behavior* of a system can be described in terms of the so-called *state* of the system. In our queuing system, one possible definition for the system state is the length of the queue, i.e., the number of people waiting in the queue. Frequently, we are interested in how a system changes over time, i.e., how the state changes as time passes. In a real banking queue, the queue length fluctuates with time, and thus the behavior of the system can also be said to change with time. Systems which change their state with time are called *dynamic* systems. In this book, we will restrict our attention to discrete-event systems. In a

A. Gosavi, *Simulation-Based Optimization*, Operations Research/Computer Science Interfaces Series 55,
DOI 10.1007/978-1-4899-7491-4_2

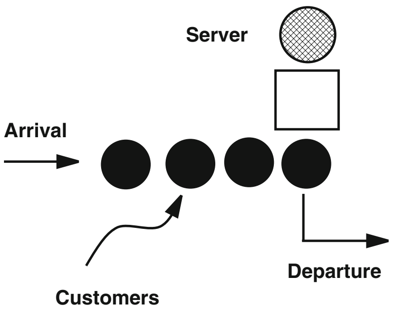

Figure 2.1. A schematic of a queue in a bank

discrete-event system, the time interval between two successive state changes, or alternatively two successive "events," is of a finite duration. Hence, a system will be understood to be a discrete-event system throughout this book.

The behavior of any dynamic system is usually *governed* by some *variables*, often called the **governing variables**. The governing variables in our banking system are the time taken by the teller to provide service to a customer and the time between successive arrivals of customers to the bank. An appropriate question at this point is: Why should these two quantities be considered to be the governing variables? The answer is that using laws from queuing theory, one can show that the behavior of the system (e.g., the queue length) depends on the values assumed by these variables.

When the governing variables are random variables, the system is referred to as a *random* or *stochastic system.* Therefore, the reader should always keep in mind that a random system's behavior is governed by one or more *random* variables. Knowledge of the distributions of these random variables is needed to analyze the system's behavior.

3. Models

To understand, to analyze, and to predict the behavior of systems (both random and deterministic), operations researchers construct **models**. These models are typically *abstract* models, unlike physical models, e.g., a miniature airplane. Abstract models take the form of equations, functions, inequations (inequalities), and computer programs etc. To understand how useful abstract models can be, consider the simple model from Newtonian physics: $v = u + gt$. This equation *predicts* the speed of a freely-falling body that has been in the air for t time units after starting its descent at a speed of u.

The literature in stochastic operations research is full of mathematical models similar to that above which help in analyzing and predicting behavior of stochastic systems. Hence, stochastic operations research is sometimes described as the **physics** of stochastic systems. Queuing theory, renewal theory, and Brownian motion theory have been exploited to construct powerful mathematical models.

Although these mathematical models enjoy an important place in operations research, they are often tied to assumptions made about the system—assumptions that are sometimes necessary to develop a model. These assumptions may be related to the system's structure or to the distributions of the governing random variables. For instance, many queuing-theory models are limited in use to exponentially distributed service and inter-arrival times. Some models related to Brownian motion are restricted to the so-called heavy traffic conditions. Not surprisingly, stochastic operations research has always found models that generalize beyond such narrow assumptions to be very attractive.

One possible avenue for generating powerful models for large and complex systems is via the use of a computer program that mimics the system's behavior. This computer program is called a *simulation model*. The program usually achieves its goal of behavior prediction by generating random numbers for the governing random variables.

For a long time, simulation models did not receive the respect that they deserve. A primary reason was that although they could be used for analyzing random systems, their use in **optimizing** systems was not well understood or as well evolved as it is today. On the other hand, mathematical models, among other things such as being elegant, can usually optimize systems. Fortunately, things have changed, and now simulation models too can be used for optimizing systems. Furthermore, they can be used to optimize *complex, large-scale, and stochastic systems* for which it may be difficult to construct mathematical models.

We note that in this book, we are keenly interested in being able to *optimize* or *control* the stochastic system, so that it operates efficiently and/or the net costs (revenues) of running the system are reduced (increased). And we will use simulation as a tool to measure the system's performance (efficiency, costs, or revenues). However, in order to achieve this goal, it is important for the reader to gain a thorough understanding of the fundamental principles underlying discrete-event simulation. In what follows, we have made a serious attempt to explain the inner workings of a simulation model.

4. Simulation Modeling

Determining the distributions of the governing random variables is the first step towards modeling a stochastic system, regardless of whether mathematical or simulation models are used. In mathematical models, the *pdfs* (or *cdfs*) of the governing random variables are used in the closed forms obtained. In simulation models, the *pdfs* (or *cdfs*) are used to generate random numbers for the variables concerned. These random numbers are then used to imitate, within a computer, the behavior of the system. Imitating the behavior of a system essentially means *re-creating* the events that occur in the real-world system that is being imitated.

How does one determine the distribution of a random variable? For this, usually, one has to actually collect data on the values of the random variables from the real-life system. Then, from this data, it is usually possible to fit a distribution to that data, which is called distribution fitting. For a good discussion on distribution fitting, see e.g., [188].

An important issue in stochastic analysis is related to the *number* of random variables in the system. Generally, the larger the number of governing random variables in a system the more complicated is its analysis. This is especially true of analysis with mathematical models. On the other hand, *simulating* a system with several governing random variables has become a trivial task with modern-day simulation packages.

Our main strategy in simulation is to *re-create*, within a computer program, the events that take place in the real-life system. The re-creation of events is based on using suitable values for the governing random variables. For this, one needs a mechanism for generating values of the governing random variables. We will first discuss how to create random values for these variables and then discuss how to use the random values to re-create the events.

4.1. Random Number Generation

Here, we will discuss some popular random number generation schemes. We begin with a scheme for the uniform distribution between 0 and 1.

4.1.1 Uniform Distribution in $(0, 1)$

We must make it clear at the outset that the random numbers that we will discuss here are *artificial.* “True” random numbers cannot be generated by a computer, but must be generated by a human brain or obtained from a real system. Having said that, for all practical purposes, *artificial* (or pseudo) random numbers generated by computers are usually sufficient in simulations. Artificial random number generation schemes are required to satisfy some statistical tests in order to be acceptable. Needless to add, the schemes that we have at our disposal today do pass these tests. We will discuss one such scheme.

The so-called *linear congruential generator* of random numbers [221] is given by the following equation:

$$I_{j+1} \leftarrow (aI_j \bmod m), \qquad j = 0, 1, 2, \ldots \tag{2.1}$$

where a and m are positive integers. Equation (2.1) should be read as follows: the *remainder* obtained after aI_j is divided by m is denoted by $(aI_j \bmod m)$. The equation above provides an iterative scheme in which if one sets a positive value less than or equal to m for I_0, the sequence of values $I_1, I_2, I_3, \ldots$ will produce integers between 0 and m, where both 0 and m excluded.

To illustrate this idea, consider the following example. Let $a = 2$, $m = 20$ and $I_0 = 12$. (Please note this set of values for a and m are used only for illustration and are not recommended in practice.) The sequence that will be generated is:

$$(12, 4, 8, 16, 12, 4, 8, 16, 12, 4, 8, 16, \ldots) \tag{2.2}$$

This sequence has integers between 1 and $m - 1 = 19$, both 1 and 19 included. It cannot include 20 or an any integer greater than 20, because each integer is obtained after a division by 20. Then, we can conclude that in the sequence defined in (2.2), the terms

$$12, 4, 8, 16$$

form a set that has integers ranging from 0 to 20 (both 0 and 20 excluded).

By a suitable choice of a and m, it is possible to generate a set that contains *all* the different integers in the range from 0 to m and each integer appears only once in the set. The number of elements in such a set will be $m - 1$. If each integer in this set is divided by m, a set of numbers in the interval $(0, 1)$ will be obtained. In general, if the ith integer is denoted by x_i and the ith random number is denoted by y_i, then

$$y_i = x_i/m.$$

Now, each number (y_i) in this set will be equally likely, because each associated integer (x_i), between 0 and m, occurs *once* at some point in the original set. Then, for large values of m, this set of random numbers from 0 to 1 will *approximate* a set of natural random numbers from the uniform distribution. Recall that the *pdf* of the random variable has the same value at every point in the uniform distribution.

The *maximum* number of integers that may be generated in this process before it starts repeating itself is $m - 1$. Also, if I_0, which is known as the **seed**, equals 0, the sequence will only contain zeroes. A suitable choice of a and m yields a set of $m - 1$ numbers such that each integer between 0 and m occurs once at some point.

An important question is: Is it acceptable to use a sequence of random numbers with repeating subsets, e.g., $(12, 4, 8, 16, 12, 4, 8, 16, \ldots)$? The answer is no because the numbers $(12, 4, 8, 16)$ repeat and are therefore deterministic. This is a serious problem. Now, unfortunately, random numbers from the linear congruential generator must repeat after a *finite* period. Therefore, the only way out of this is to generate a sequence that has a *sufficiently long* period, such that we are finished with using the sequence before it gets to repeats itself. Fortunately, if $m = 2^{31} - 1$, then with a suitable choice of a, it is possible to generate a sequence that has a period of $m - 1$. Thus, if the number of random numbers needed is less than $m - 1 = 2{,}147{,}483{,}646$ (for most applications, this is sufficient), we have a set of random numbers with no repetitions.

Schemes with small periods produce erroneous results. Furthermore, repeating numbers are also not *independent.* In fact, repetitions imply that the numbers stray far away from the uniform distribution that they seek to approximate.

If the largest number in a computer's memory is $2^{31} - 1$, then a legal value for a that goes with $m = 2^{31} - 1$ is 16,807. These values of m and a **cannot** be implemented naively in the computer program. The reason is easy to see. In the multiplication of a by I_j, where the latter can be of the order of m, one runs into trouble as the product is often larger than m, the largest number that the computer can store. A clever trick from Schrage [266] helps us circumvent this difficulty. Let $[x/y]$ denote the *integer* part of the quotient obtained after dividing x by y. Using Schrage's approximate factorization, if

$$Q = a(I_j \bmod q) - r[I_j/q],$$

the random number generation scheme is given by

$$I_{j+1} \leftarrow \begin{cases} Q & \text{if } Q \geq 0 \\ Q + m & \text{otherwise.} \end{cases}$$

In the above, q and r are positive numbers. As is clear from this *approximate* factorization, multiplication of a and I_j, which is required in Eq. (2.1), is avoided. For $a = 7^5 = 16{,}807$ and $m = 2^{31} - 1$, values suggested for q and r are: $q = 127{,}773$ and $r = 2{,}836$. Other values can be found in e.g., [221]. The period of this scheme is $m - 1 = 2^{31} - 2$. When the number of calls to this scheme becomes of the order of the period, it starts repeating numbers and is **not** recommended. Two sequences of different periods can be combined to give a sequence of a longer period [189]. The reader is referred to [189] for further reading. See [239] for useful computer programs.

The above-described scheme for generating random numbers from the uniform distribution $(0, 1)$ forms the work horse for random number generation in many commercial packages. It can be used for generating random numbers from any other distribution.

4.1.2 Inverse Function Method

In this subsection, we will discuss how random numbers for some distributions can be generated. In particular, we will discuss the inverse function method.

The inverse function method relies on manipulating the *cdf* of a function. This often requires the *cdf* to be "nice." In other words, the *cdf* should be of a form that can be manipulated. We will show how this method works on the unform distribution (a, b) and the exponential distribution.

For any given value of x, the *cdf* has to assume a value between 0 and 1. Conversely, it may be said that when $F(x)$ assumes a value between 0 and 1, that particular value corresponds to some value of x. Hence, one way to find a random number from a given distribution is to consider a *random* value for $F(x)$, say y, and then determine the value of x that corresponds to y. Since values of $F(x)$ lie between 0 and 1, we can assume that y must come from the uniform distribution $(0, 1)$. Hence, our strategy in the inverse function method is to generate a random number y from the uniform distribution (0,1), equate it to the *cdf*, and then solve for x, which will denote the random number we desire.

The *cdf* of the uniform distribution (a, b) is given by

$$F(x) = \frac{x - a}{b - a}.$$

Hence, using our strategy and solving for x, we have:

$$y = F(x) = \frac{x - a}{b - a}, \text{ i.e., } x = a + y(b - a).$$

The *cdf* of the exponential distribution is given by:

$$F(x) = 1 - e^{-\lambda x}, \text{ where } \lambda > 0.$$

Then using the same strategy, we obtain:

$$F(x) = y, \text{ i.e., } 1 - e^{-\lambda x} = y, \text{ i.e., } e^{-\lambda x} = 1 - y.$$

Taking the natural logarithm of both sides, after some simplification, we have:

$$x = -\frac{ln(1-y)}{\lambda}.$$

Replacing $(1-y)$ by y leads to $x = -\frac{ln(y)}{\lambda}$, which is also an acceptable rule because y is also a number between 0 and 1.

The method described above is called the inverse function method essentially because the *cdf* is manipulable and it is possible to solve for x for a given value of $F(x)$. Sometimes the closed form for the *cdf* may not be manipulable directly, but it may be possible to develop a closed-form approximation of the inverse of the *cdf*. Such approximations are often acceptable in computer simulation, because even computing the logarithm in the scheme described above for the exponential distribution requires an approximation. A remarkable example of this is the closed form approximation of the inverse of the normal distribution's *cdf*, due to Schmeiser [263], which leads to the following formula for generating a random number from the *standard* normal distribution:

$$x = \frac{y^{0.135} - (1-y)^{0.135}}{0.1975}.$$

This provides one decimal-place accuracy for $0.0013499 \leq y \leq 0.998\,6501$ at the very least. From this, one can determine a random number for the normal distribution having a mean of μ and a standard deviation of σ using: $\mu + x\sigma$.

Clearly, the inverse function method breaks down in the absence of a manipulable closed form for the *cdf* or an approximation for the inverse *cdf*. Then, one must use other methods, e.g., the acceptance-rejection method, which is discussed in numerous textbooks [188].

We will now turn to an important mechanism lying at the heart of discrete-event computer simulation. A clear understanding of it is vitally essential.

4.2. Event Generation

The best way to explain how values for random variables can be used to re-create events within a simulator is to use an example. We will use the single-server queuing example that has been discussed above. In the single-server queue, there are two governing variables, both of which are random:

1. The time between successive arrivals to the system: t_a

2. The time taken by the server to give service to one customer: t_s.

We now generate values for the elements of the two sequences. Since we know the distributions, it is possible to generate values for them. E.g.,

$$\text{let the first 7 values for } t_a \text{ be: } 10.1, 2.3, 1, 0.9, 3.5, 1.2, 6.4$$

$$\text{and those for } t_s \text{ be: } 0.1, 3.2, 1.19, 4.9, 1.1, 1.7, 1.5.$$

Now $\{t(n)\}_{n=1}^{\infty}$ will denote the following sequence: $\{t(1), t(2), \ldots, \}$. These values, we will show below, will lead to re-enacting the real-life queue.

If one observes the queue from real life and collects data for any of these sequences from there, the values obtained may not necessarily be identical to those shown above. Then, how will the above lead to a re-enactment of the real-life system within our simulation model? The answer is that the elements of this sequence belong to the distribution of the random variable in the real-life system. In other words, the above sequence could very well be a sequence from the real-life system. We will discuss later why this is *sufficient* for our goals in simulation modeling.

Now, from the two sequences, one can construct a sequence of the events that occur. The events here are of two types:

1. A customer enters the system (**arrival**).

2. A customer is serviced and leaves the system (**departure**).

Our task of re-creating events boils down to the task of finding the time of occurrence of each event. In the single-server queuing system (see Fig. 2.2), when a customer arrives to find that the server is idle, he or she directly goes to the server without waiting. An arriving customer who finds that the server is busy providing service to someone becomes either the first person in the queue or joins the queue's end. The arrivals in this case, we will assume, occur regardless of the number of people waiting in the queue.

To analyze the behavior of our system, the first task is to determine the clock time of each event as it occurs. This task, as stated above, lies at the heart of stochastic simulation. If one can accomplish this task, various aspects of the system can be analyzed.

Before delving into the details of how we can accomplish this task, we need a clear understanding of what the elements of the sequences $\{t_a(n)\}_{n=1}^{\infty}$ and $\{t_s(n)\}_{n=1}^{\infty}$ stand for. The first element in each sequence is associated with the first customer to arrive in the system, the second is associated with the second customer, and so on. Keeping this in mind, note that in the queuing example, there are two types of events: arrivals and departures. Our task in each case can be accomplished as described below:

1. For arrivals: The clock time of any arrival in the system will be equal (in this case) to (1) the clock time of the previous arrival plus (2) the inter-arrival time of the current arrival. (Remember the second quantity is easy to find. The inter-arrival time of the kth arrival is $t_a(k)$.)

2. For departures:

   a. If an arriving customer finds that the server is idle, the next departure will occur at a clock time equal to the arrival clock time of the arriving customer plus the service time of the arriving customer. (Again, the second quantity is easy to find. The service time of the kth arrival is $t_s(k)$.)

   b. If an arriving customer finds that the server is busy, the next departure will occur at a time equal to (1) the clock time of the *previous* departure plus (2) the service time of the customer *currently being served.*

We now illustrate these ideas with the values generated for the sequence. See Fig. 2.2. The first arrival takes place at clock time of 10.1, the second at clock time of $10.1 + 2.3 = 12.4$, the third at clock time of $12.4 + 1 = 13.4$, the fourth at clock time of $13.4 + 0.9 = 14.3$, the fifth at a clock time of $14.3 + 3.5 = 17.8$, the sixth at a clock time of $17.8 + 1.2 = 19$, and the seventh at a clock time of $19 + 6.4 = 25.4$.

When the first arrival occurs, there is nobody in the queue and hence the first departure occurs at a clock time of $10.1 + 0.1$ (service time of the first customer) $= 10.2$. Now, from the clock time of 10.2 till the clock strikes 12.4, when the second arrival occurs, the server is idle. So when the second arrival occurs, there is nobody in the queue and hence the time of the second departure is $12.4+3.2$ (the service time of the second arrival)$= 15.6$. The third arrival takes place at a clock time of 13.4, but the second customer departs much later at a clock time of

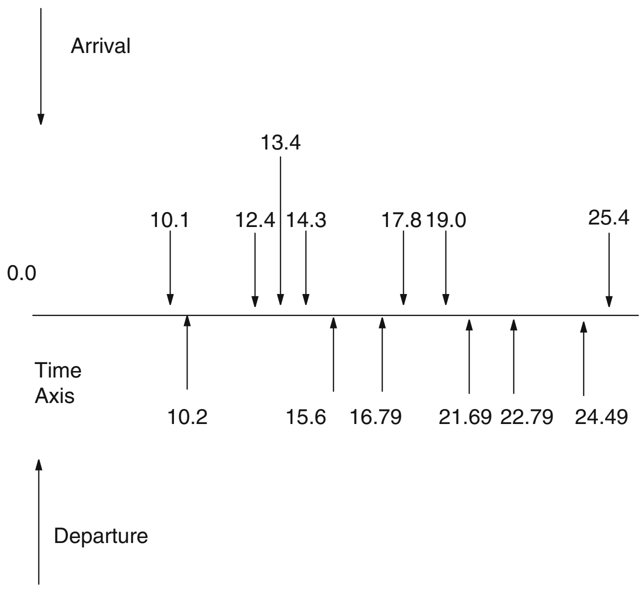

Figure 2.2. The event clock showing arrivals and departures

15.6. Hence the second customer must wait in the queue till 15.6 when he/she joins service. Hence the third departure will occur at a clock time of 15.6 plus the service time of the third customer, which is 1.19. Therefore, the departure of the third customer will occur at a clock time of $15.6 + 1.19 = 16.79$. The fourth customer arrives at a clock time of 14.3 but the third departs only at 16.79. It is clear that the fourth customer will depart at a clock time of $16.79 + 4.9 = 21.69$. In this way, we can find that the fifth departure will occur at a clock time of 22.79 and the sixth at a clock time of 24.49. The seventh arrival occurs at a clock time of 25.4, which is after the sixth customer has departed. Hence when the seventh customer enters the system, there is nobody in the queue, and the seventh customer departs some time after the clock time of 25.4. We will analyze the system until the time when the clock strikes 25.4.

Now, from the sequence of events constructed, we can collect data related to system parameters of interest. First consider server utilization. From the observations above, it is clear that the server was idle from a clock time of 0 until the clock struck 10.1, i.e., for a *time interval* of 10.1. Then again it was idle from the time 10.2 (the clock time of the first departure) until the second arrival at a clock time of 12.4, i.e., for 2.2 time units. Finally, it was idle from the time 24.49

(the clock time of the sixth departure) until the seventh arrival at a clock time of 25.4, i.e., for 0.91 time units. Thus, based on our observations, we can state that the system was idle for a total time of $10.1 + 2.2 + 0.91 = 13.21$ out of the total time of 25.4 time units for which we observed the system. Then, the server utilization (fraction of time for which the server was busy) is clearly: $1 - \frac{13.21}{25.4} = 0.4799$.

If one were to create very *long* sequences for the inter-arrival times and the service times, one could then obtain estimates of the utilization of the server *over a long run.* Of course, this kind of a task should be left to the computer, but the point is that computer programs are thus able to collect estimates of parameters measured over a long run.

It should now be clear to the reader that although the sequences generated may not be *identical* to sequences obtained from actual observation of the original system, what we are really interested in are the estimates of system parameters, e.g., long-run utilization. As long as the sequence of values for the governing random variables are generated from the correct distributions, reliable estimates of these parameters can be obtained. For instance, an estimate like long-run utilization will approach a constant as the simulation horizon (25.4 time units in our example) approaches infinity.

Many other parameters for the queuing system can also be measured similarly. Let $\mathsf{E}[W]$ denote the average customer waiting time in the queue. Intuitively, it follows that the average waiting time can be found by summing the waiting times of a large number (read infinity) of customers and dividing the sum by the number of customers. Thus if w_i denotes the queue waiting time of the ith customer, the long-run average waiting time should be

$$\mathsf{E}[W] = \lim_{n\to\infty} \frac{\sum_{i=1}^{n} w_i}{n}. \tag{2.3}$$

Similarly, the *long-run* average number in the queue can be defined as:

$$\mathsf{E}[Q] = \lim_{T\to\infty} \frac{\int_0^T Q(t)dt}{T}, \tag{2.4}$$

where $Q(t)$ denotes the number in the queue at time t. Now, to use these definitions, in practice, one must treat ∞ as a large number.

For *estimating* the long-run average waiting time, we can use the following formula:

$$\tilde{W} = \frac{\sum_{i=1}^{n} w_i}{n}, \tag{2.5}$$

where $\tilde{W}$ is essentially the sample mean from n samples. In the example of Fig. 2.2,

$$w_1 = w_2 = 0, w_3 = 15.6 - 13.4, w_4 = 16.79 - 14.3, w_5 = 21.69 - 17.8, \text{ and}$$

$$w_6 = 22.79 - 19.0, \text{ i.e., } \tilde{W} = \frac{0 + 0 + 2.2 + 2.49 + 3.89 + 3.79}{6} = 2.06$$

For estimating the average number in the queue, one uses a similar mechanism. The formula for the estimate is:

$$\tilde{Q} = \frac{\sum_{i=1}^{n} t_i Q_i}{T}, \tag{2.6}$$

where t_i is the time duration for which there were Q_i customers in the queue, and $T = \sum_{i=1}^{n} t_i$ is the amount of time for which simulation is conducted. Again, $\tilde{Q}$ is a sample mean.

4.3. Independence of Samples Collected

Often, in simulation analysis, one runs the simulation a number of times, using a *different* set of random numbers in each run. By changing the seed, defined above, one can generate a new set of random numbers. Each run of the simulation with a given set of random numbers is called a **replication**. One replication yields only one *independent* sample for the quantity (or quantities) we are trying to estimate. The reason for this independence is that each replication runs with a *unique* set of random numbers, and hence, an estimate from one replication is independent of that from another. On the other hand, samples from within the same replication may depend on each other, and thus they are not independent. Independent estimates provide us with a mechanism to estimate the true mean (or variance) of the parameter of interest. Independent estimates also allow us to construct confidence intervals on the estimates of means obtained from simulations. The need for independence follows from the strong law of large numbers that we discuss below.

In practice, one estimates sample means from numerous replications, and the means from all the replications are averaged to obtain a *reliable* estimate of the true mean. This process of estimating the mean from several replications is called the *independent replications* method. We can generalize the concept as follows. Let $\tilde{W}_i$ denote the estimate of the waiting time from n samples from the ith replication. Then, a good estimate of the true mean of the waiting time from k replications can be found from the following:

$$\mathsf{E}[W] = \frac{\sum_{i=1}^{k} \tilde{W}_i}{k},$$

provided k is large enough.

Averaging over many independent estimates (obtained from many replications), therefore, provides a good estimate for the true mean of the parameter we are interested in. However, doing multiple replications is not the only way to generate independent samples. There are other methods, e.g., the **batch means** method, which uses one *long* replication. The batch means method is a very intelligent method (see Schmeiser [264]) that divides the output data from a long replication into a small number of large batches, after deletion of some data. The means of these batches, it can be shown, can be treated as independent samples. These samples are then used to estimate the mean. We now present the mathematical result that allows us to perform statistical computations from means.

THEOREM 2.1 *(The Strong Law of Large Numbers) Let X_1, $X_2, \ldots$ be a sequence (a set with infinitely many elements) of independent random variables having a common distribution with mean $\mathsf{E}(X) = \mu$. Then, with probability 1,*

$$\lim_{k \to \infty} \frac{X_1 + X_2 + X_3 + \cdots + X_k}{k} = \mu.$$

In the above theorem, $X_1, X_2, X_3, \ldots$ can be viewed as values of the same random variable from a given distribution. Essentially, what the theorem implies is that if we draw k independent samples of the random variable, then the sample mean, i.e., $\frac{X_1+X_2+X_3+\cdots+X_k}{k}$, will tend to the actual mean of the random variable as k becomes very large.

This is an important result used heavily in simulations and in many other settings where samples are obtained from populations to make predictions about the population. While what this law states may be intuitively obvious, what one needs to remember is that the samples drawn have to be independent. Its proof can be found in [252] under some assumptions on the second moment of the random variable.

It should now be clear to the reader why the means obtained from a replication or a batch must be used to estimate the mean in simulations. The independence of the estimates obtained in simulation (either from replications or batches) also allows us to determine confidence intervals (of the mean) and prediction intervals of the parameter of interest. Estimating means, variances, and confidence and prediction intervals is an important topic that the reader should become familiar with.

5. Concluding Remarks

Simulation of dynamic systems using random numbers was the main topic covered in this chapter. However, this book is not about the methodology of simulation, and hence our discussion was not comprehensive. Nevertheless, it is an important topic in the context of simulation-based optimization, and the reader is strongly urged to get a clear understanding of it. A detailed discussion on writing simulation programs in C can be found in [234, 188]. For an in-depth discussion on tests for random numbers, see Knuth [177].

Historical Remarks: The earliest reference to generating random numbers can be traced to George Louis Leclec (later called Buffon) in 1733. It was in the 1940s, however, that "Monte Carlo simulation" first became known, and it was originally used to describe random number generation from a distribution. However, the name is now used loosely to refer to any simulation (including discrete-event) that uses random numbers. The advent of computers during the 1960s led to the birth of *computer* simulation. The power of computers has increased dramatically in the last few decades, enabling it to play a major role in analyzing stochastic systems. The fundamental contributions in simulation were made by pioneers in the industrial engineering community, who worked tirelessly through the decades allowing it to develop into a reliable and sophisticated science that it is today [14].

Chapter 3

SIMULATION-BASED OPTIMIZATION: AN OVERVIEW

1. Chapter Overview

The purpose of this short chapter is to discuss the role played by computer simulation in simulation-based optimization. Simulation-based optimization revolves around methods that require the maximization (or minimization) of the net rewards (or costs) obtained from a *random* system. We will be concerned with two types of optimization problems: (1) **parametric** optimization (also called **static** optimization) and (2) **control** optimization (also called **dynamic** optimization).

2. Parametric Optimization

Parametric optimization is the problem of finding the values of decision variables (**parameters**) that maximize or minimize some function of the decision variables. In general, we can express this problem as:

Maximize or Minimize $f(x(1), x(2), \ldots, x(N))$, where the N decision variables are: $x(1)$, $x(2), \ldots$, and $x(N)$. It is also possible that there are some constraints on the values of the decision variables.

In the above, $f(.)$ denotes a function of the decision variables. It is generally referred to as the **objective function**. It also goes by other names, e.g., the performance metric (measure), the cost function, the loss function, and the penalty function. Now, consider the following example.

A. Gosavi, *Simulation-Based Optimization*, Operations Research/Computer Science Interfaces Series 55,
DOI 10.1007/978-1-4899-7491-4_3

Example 1. Minimize $f(x,y) = (x-2)^2 + (y-4)^2$,
where x and y take values in the interval $(-\infty, \infty)$.

Using calculus, it is not hard to show that the optimal point (x^*, y^*) is $(2,4)$. One calculates the partial derivative of the function with respect to x, and then sets it to 0:

$$\frac{\partial f(x,y)}{\partial x} = 0, \text{ i.e., } x = 2.$$

Similarly, $y = 4$. Finding the optimal point in this case is straightforward, because the objective function's closed form is known. Although the optimization process may not always be this straightforward, the availability of the closed form often simplifies the optimization process.

We now turn our attention to an objective function with *stochastic elements*, i.e., the objective function involves either the probability mass or density function (*pmf* or *pdf*) or the cumulative distribution function (*cdf*) of one or more random variables.

Example 2.

a. Minimize

$$f(x(1), x(2)) = 0.2[x(1)-3]^2 + 0.8[x(2)-5]^2.$$

b. Minimize

$$f(x) = \int_{-\infty}^{\infty} 8[x-5]^{-0.3} g(x)dx$$

Assume the objective function in each case to be the expected value of some random variable, which is associated with a random system. Let us further assume that 0.2 and 0.8 are the elements of the *pmf* (case a) of the random variable and that g is the *pdf* (case b) of the random variable.

Example 2 is a non-linear program in which the *closed* form of the objective function is known. Hence, it is likely that it can be solved with standard non-linear programming techniques. Now, consider the following scenario in which *it is difficult* to obtain the elements of the *pmf* or *pdf* in the objective function given above, but its value can be *estimated* via simulation. In other words, the following holds: (i) The *closed* form of $f(.)$ is *not* known. (ii) It may be difficult or too time-consuming to obtain the closed form. It is in this scenario that simulation may be very helpful in optimization.

Why avoid the closed form? The main reason is that in many real-world stochastic problems, the objective function is too complex to be obtained in its closed form. Then, it may be possible to employ a theoretical model (e.g., renewal theory, Brownian motion, and exact or approximate Markov chains) to obtain an *approximate* closed form. In the process of generating a theoretical model, however, it oftentimes becomes necessary to make simplifying assumptions about the system, e.g., some system random variable is exponentially distributed, to keep the model tractable. The model generated may then turn out to be too simplistic for use in the real world. Clearly, then, if optimization could be performed *without obtaining the closed form*, it would become possible to make realistic assumptions about the system and still optimize it.

Fortunately, optimization methods exist that only need the *numeric* value of the function at any given point, i.e., they do not require the closed form. They are referred to as numerical methods because they depend only on the numeric value of the function at any given point. This is in contrast to analytic methods that need the closed form. Examples of numerical methods are the simplex method of Nelder and Mead [216], finite difference gradient descent, and the simultaneous perturbation method [280].

The advantages of numerical methods lie in their ability to perform optimization without the closed form. Thus, they can optimize the function even when the closed form is unknown, but some mechanism to estimate the function value is available. Consequently, numerical methods form the natural choice for solving complex stochastic optimization problems, where the objective function's closed form is unknown, but the function can be evaluated numerically.

Simulation's role: It is often the case that a stochastic system can be easily *simulated*, whereas a closed-form mathematical model for the related objective function is hard to find. From the simulator, with some effort, the objective function's value can also be estimated. Hence simulation in combination with numerical optimization methods may prove to be an effective tool for attacking difficult optimization problems. The first part of this book focusses on numerical techniques that can be combined with simulation.

The role played by simulation can be explained as follows. In many problems, the objective function is an expected (mean) value of a random variable X. Then, simulation can be used to generate samples of this random variable, $X_1, X_2, \ldots, X_n$, at any given point in the solution space. These samples can be used to find an *estimate* of the objective function at the given point using

$$E(X) \simeq \frac{X_1 + X_2 + \cdots, X_n}{n}.$$

The above follows from the strong law of large numbers (see Theorem 2.1), provided the samples are independent and n is sufficiently large. This "estimate" plays the role of the objective function value.

Combining simulation with numerical parametric optimization methods is easier said than done. There are many reasons for this. First, the estimate of the objective function is not perfect and contains "noise." Fortunately, the effect of noise can often be minimized. Second, parametric optimization methods that require a very large number of function evaluations to generate a good solution may not be of much use in practice, since even *one* function evaluation via simulation usually takes a considerable amount of computer time (one function evaluation in turn requires several samples, i.e., replications or batches).

We would like to reiterate that the role simulation can play in parametric optimization is limited to estimating the function value. Simulation on its own is not an optimization technique. But, as stated above, combining simulation with optimization is possible in many cases, and this opens an avenue along which many real-life systems may be optimized. In subsequent chapters, we will deal with a number of techniques that can be combined with simulation to obtain solutions in a reasonable amount of computer time.

3. Control Optimization

The problem of control optimization is different than the problem of parametric optimization in many respects. Hence, considerable work in operations research has occurred in developing *specialized* techniques for control optimization.

A *system* is defined as a collection of entities (such as people and machines) that interact with each other. A *dynamic* system is one in which the system *changes* in some way from time to time. To detect changes in the system, we describe the system using a numerical attribute called state. Then, a change in the value of the attribute can be *interpreted* as a change in the system.

A stochastic system is a dynamic system in which the state changes *randomly*. For example, consider a queue that builds up in front of a counter in a supermarket. Let the state of the system be denoted by the number of people waiting in the queue. Then, clearly, the queue is stochastic system because the number of people in the queue fluctuates randomly. The randomness in the queuing system could be due to random inter-arrival times of customers or the random service times of the servers (the service providers at the counters) or both.

The science of stochastic control optimization deals with methods that can *control* a stochastic system such that desirable behavior, e.g., improved efficiencies and reduced costs, is obtained. The field of stochastic control optimization in discrete-event systems covers several problems in a subset of which the system's behavior is governed by a Markov chain. In this book, we will focus on this subset.

In general, in any control optimization problem, the goal is to move the system along a desirable path, i.e., to move it along a desirable trajectory of states. In most states, one has to select from more than one *action*. The actions in each state, essentially, dictate the trajectory of states followed by the system. The problem of control optimization, hence, revolves around selecting the right actions in all the states visited by the system. The performance metric is a function of the actions selected in all the states. Let $\mathcal{S}$ denote the set of states in the system and $|\mathcal{S}|$ denote the number of states in the system. In general, the control optimization problem can be mathematically described as:

$$\text{Maximize or Minimize } f(\mu(1), \mu(2), \ldots, \mu(|\mathcal{S}|)),$$

where $\mu(i)$ denotes the action selected in state i and $f(.)$ denotes the objective function. In large problems, $|\mathcal{S}|$ may be of the order of thousands or millions.

Dynamic programming is a well-known and efficient technique for solving many control optimization problems encountered in discrete-event systems governed by Markov chains. It requires the computation of a so-called *value function* for every state.

Simulation's role: It turns out that every element of the value function of dynamic programming can be expressed as an *expectation* of a random variable. Also, fortunately, it is the case that simulation can be used to generate samples of this random variable. Let us denote the random variable by X and its ith sample by X_i. Then, using the strong law of large numbers (see Theorem 2.1), the value function at each state can be estimated by using:

$$E(X) \simeq \frac{X_1 + X_2 + \ldots + X_n}{n},$$

provided n is sufficiently large and the samples are independent. As stated above, simulation can be used in conjunction with dynamic programming to generate a large number of independent samples of the value function. It is important to note that, like in parametric optimization, the role of simulation here is limited to generating samples of the value function. For optimization purposes, we need to turn to methods that are specifically derived for control optimization, e.g., dynamic programming.

There are many advantages to using simulation in combination with dynamic programming. A major drawback of dynamic programming is that it requires the so-called *transition probabilities* of the system. These probabilities are often hard to obtain for complex and large-scale systems. Theoretically, these transition probabilities can be generated from the distributions of the governing random variables of the system. However, in many complex problems with numerous random variables, this may prove to be a difficult task. Hence, a computational challenge created by real-world problems is to evaluate the value function without having to compute the transition functions. It is precisely here that simulation can play a useful role.

One way to use simulation is to first estimate the transition probabilities in a simulator, and then employ classical dynamic programming using the transition probabilities. However, this is usually an inefficient approach. In a method called *reinforcement learning*, simulation is used to generate samples of the value function, which are then averaged to obtain the expected value of the value function. This bypasses the need to compute the transition probabilities. Chapters on control optimization in this book focus on this approach.

Compared to finding the transition probabilities of a complex stochastic system, simulating the same system is relatively "easy." As such, a combination of simulation and dynamic programming can help us solve problems whose transition probabilities are hard to find. These problems were considered intractable in the past, before the advent of reinforcement learning.

A significant volume of the literature in operations research is devoted to analytical methods for finding exact expressions of transition probabilities underlying complex random systems. Expressions for the transition probabilities can get mathematically involved with multiple integrals and complicated algebra (see e.g., [74]). Furthermore, to keep the mathematics tractable, sometimes these expressions are obtained only after making simplifying, but restrictive, assumptions about the system. Since these transition probabilities are not needed in reinforcement learning, many of these assumptions can be easily relaxed. This is a main strength of reinforcement learning.

Reinforcement learning can also be elegantly integrated with the so-called function approximation techniques, e.g., regression and neural networks, that allow us to store the value function of hundreds of thousands of states using a few scalars. This addresses the issue of large dimensionality that plagues many real-world systems with large-scale state spaces.

In conclusion, reinforcement learning, which works within simulators, provides us with a mechanism to solve complex and large-scale control problems governed by the Markov property without generating their transition probabilities. Thus, via reinforcement learning, one has the ability to solve complex problems for which it is difficult, if not impossible, to obtain the transition probabilities. Also in conjunction with function approximators, one can solve large-scale problems.

4. Concluding Remarks

The goal of this chapter was to introduce the reader to two types of problems that will be solved in this book, namely parametric optimization and control optimization (of systems with the Markov property). We remind the reader that parametric optimization is also popularly known as **static** optimization, while control optimization is popularly known as **dynamic** optimization.

The book is not written to be a comprehensive source on the topic of "simulation-based optimization." Rather, our treatment is aimed at providing an introduction to this topic with a focus on some important breakthroughs in this area. In particular, our treatment of parametric optimization is devoted to model-free methods that do not require any properties of the objective function's closed form. In case of control optimization, we only cover problems related to Markov chain governed systems whose transition models are hard to obtain.

Chapter 4

PARAMETRIC OPTIMIZATION: RESPONSE SURFACES AND NEURAL NETWORKS

1. Chapter Overview

This chapter will discuss one of the oldest simulation-based methods of parametric optimization, namely, the response surface method (RSM). While RSM is admittedly primitive for the purposes of simulation optimization, it is still a very robust technique that is often used when other methods fail. It hinges on a rather simple idea, which is to obtain an approximate form of the objective function by simulating the system at a finite number of points carefully sampled from the function space. Traditionally, RSM has used regression over the sampled points to find an approximate form of the objective function.

We will also discuss a more powerful alternative to regression, namely, neural networks in this chapter. Our analysis of neural networks will concentrate on exploring its roots, which lie in the principles of steepest gradient descent (or **steepest descent** for short) and least square error minimization, and on its use in simulation optimization.

We will first discuss the theory of regression-based traditional response surfaces. Thereafter, we will present a response surface technique that uses neural networks that call **neuro-response surfaces**.

2. RSM: An Overview

The problem considered in this chapter is the "parametric-optimization problem" discussed in Chap. 3. For the sake of convenience, we reproduce the problem statement here.

A. Gosavi, *Simulation-Based Optimization*, Operations Research/Computer Science Interfaces Series 55,
DOI 10.1007/978-1-4899-7491-4_4

Maximize $f(\vec{x})$, subject to some linear or non-linear constraints involving the vector

$$\vec{x} = (x(1), x(2), \ldots, x(N)).$$

Here $f(.)$ denotes the objective function, which may be a linear or non-linear function. The elements of vector, $\vec{x}$, which are $\{x(1), x(2), \ldots, x(N)\}$, are the decision variables. In this book, our interest lies in an objective function with the following traits:

1. It is difficult to obtain an expression for its closed form.

2. The closed form contains elements of *pdf*s or *cdf*s, and its value can be estimated via simulation.

As discussed previously in Chap. 3, usually, the value of such functions can be estimated at any given point using simulation. Hence, not surprisingly, simulation can prove to be a useful tool for optimizing such functions—via optimization techniques that rely solely on function evaluation. Although, no attempt is made to find the exact closed form in RSM (we will try to do without the exact closed form throughout this entire book), we will make a **guess** of the structure of the closed form. This guess is usually called the **metamodel**. The metamodel is a term that distinguishes the guessed form from the term "closed form."

We explain this idea with an example. If we assume the structure of the objective function to be linear in one independent variable—x, the decision variable, the metamodel assumes the equation of a straight line, which is:

$$y = a + bx.$$

Here a and b are unknowns that define the metamodel. We will try to estimate their values using the available data related to the function. This is essentially what RSM is all about.

The strategy underlying RSM consists of the following three steps:

1. Select a finite number of points, and evaluate the function at a finite number of points.

2. Assume a **metamodel** for the objective function, and use regression (or some other approach) to fit the metamodel equation to the data; the data is comprised of the selected points and their function values.

Using statistical tests, determine whether the assumed metamodel is acceptable.

3. If the assumed metamodel is acceptable, use it to determine the optimal point; otherwise, go back to the second step.

Remark: We hope that once we estimate the values of the unknowns in the metamodel (e.g., a and b in the straight line), what we have in the metamodel is a *good* approximation of the actual closed form (which is unknown).

Let us, next, look at some more examples of metamodels. If the function is linear with two decision variables (x, y), the metamodel assumes the equation of a plane: $z = ax + by + c$.

The function could be non-linear (a very large number of real-world problems tend to have non-linear objective functions) with one decision variable, and the metamodel **could** be: $y = a + bx^2$, or it **could** be: $y = a + bx + cx^2$, and so on. Clearly, there exist an infinite number of metamodels for a given non-linear objective function, and therefore when the closed form is unknown, the structure of the metamodel is also unknown. Fortunately, there are statistical tests that can be used to determine whether an assumed metamodel is acceptable. One should always use a statistical test to determine if the metamodel used is acceptable.

However, notice that the third step in the RSM strategy may reveal that the metamodel assumed in the second step was incorrect. When this happens, one has to guess **another** metamodel, based on the knowledge of the function, and get back to work! This means that the method may need several iterations if the third step keeps showing that the assumed metamodel is, in fact, not acceptable.

Now, this does not sound very exciting, but it turns out that in practice, very often, we can make pretty good guesses that can closely approximate the actual closed form. Moreover, oftentimes, there are multiple metamodels that are acceptable, and it is sufficient to discover one of them. In Sect. 4, we will develop a *neural network based* response surface method that does not need the knowledge of the metamodel.

3. RSM: Details

As stated above, RSM consists of several steps. In what follows, we will discuss each step in some detail.

3.1. Sampling

Sampling of points (data pieces) from the function space is an issue that has been studied by statisticians for several years [213]. Proper sampling of the function space requires a good **design of**

experiment. In simulation optimization, the experiment has to be designed properly. The reason is obtaining the function value, at even one point, can be time consuming. As a consequence, one must make an *economic* choice of the number of points to be sampled.

A rough guideline for sampling is as follows: Divide the solution space into a finite number of zones, and select the corner points (or points very close to the corner points) of each zone as samples. Additional samples may be collected from the central points in each zone. Fishing for other points by sampling uniformly in between the corner points is a frequently used strategy when nothing is known about the function. Tutorials on designing the experiment for an RSM-based experiment in simulation and optimization can be found in [281, 172, 161]. The reader is also referred to an excellent article by Sanchez [261] for further reading.

3.2. Function Fitting

Regression is usually used to fit a function when the coordinates and function values of some points are known. To use standard regression, one must assume the metamodel of the function to be known. We will begin with the simplest possible example, and then move on to more complicated scenarios.

3.2.1 Fitting a Straight Line

The problem of fitting a straight line belongs to $\Re^2$ space. In other words, the data related to the linear function in one variable is available in the form of (x, y) pairs (or **data pieces**), where y is the objective function value and x is the decision variable. The metamodel is hence:

$$y = a + bx, \tag{4.1}$$

with unknown a and b. See Fig. 4.1.

Regression is one of the many ways available to fit a straight line to given (x, y) pairs for obtaining the values of a and b. Some other methods are: Chebyshev fitting, minimax error fitting, absolute mean error fitting, etc. We will not pursue these topics here; we will limit our discussion to regression.

Regression, in comparison to most of the other methods mentioned above, happens to have a low computational burden. It is important to understand the mechanism of regression to appreciate the philosophy underlying RSM and neural networks.

Regression minimizes the *the total squared error between the actual data and the data predictions from the model (metamodel equation) assumed.* For instance, we assumed above that the model is linear.

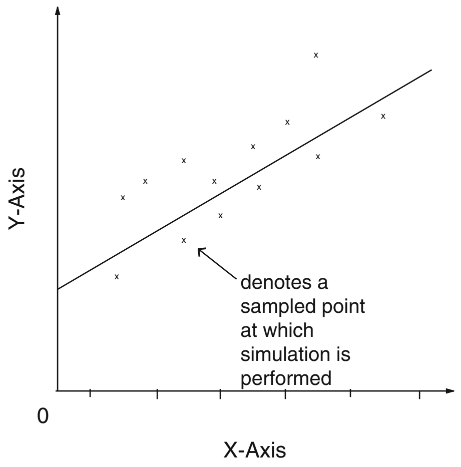

Figure 4.1. Fitting a straight line

Then using regression, we can come up with values for a and b to predict the value of y for any given value of x. Clearly, the predicted value may differ from the actual value unless the data is perfectly linear. In regression, our objective is to find those values of a and b that minimize the sum of the square of these differences. Let us state this in more formal terms.

Let $(x_p, y_p), p = 1, 2, \ldots, n$ represent n data-pairs available to us. The x_p values are selected by the analyst from the optimization space and the corresponding y_p values are the objective function values obtained from simulation. Let us define e_p as the error term for the pth data piece. It is the difference between the actual value of the objective function, which is y_p, and the predicted value, which is $a + bx_p$. Hence

$$e_p = y_p - (a + bx_p). \tag{4.2}$$

As stated above, the goal in regression is to find the values of a and b that minimize the sum of the squares of the e_p terms. We will denote this sum by SSE. In other words, the goal is to

$$\boxed{\textbf{minimize } SSE \equiv \textstyle\sum_{p=1}^{n} (e_p)^2.}$$

Now, using Eq. (4.2), we have that

$$SSE \equiv \sum_{p=1}^{n} (e_p)^2 = \sum_{p=1}^{n} (y_p - a - bx_p)^2.$$

To find a value of a that minimizes SSE, we can find the partial derivative of SSE with respect to a and then equate it to 0, as shown next.

$$\frac{\partial}{\partial a}(SSE) = 0.$$

Calculating the partial derivative, the above becomes:

$$2\sum_{p=1}^{n}(y_p - a - bx_p)(-1) = 0, \text{ which simplifies to:}$$

$$na + b\sum_{p=1}^{n} x_p = \sum_{p=1}^{n} y_p, \text{ noting that } \sum_{p=1}^{n} 1 = n. \tag{4.3}$$

Like in the preceding operations, to find the value of b that minimizes SSE, we can calculate the partial derivative with respect to b and then equate it to 0. Thus

$$\frac{\partial}{\partial b}(SSE) = 0 \text{ which implies that: } 2\sum_{p=1}^{n}(y_p - a - bx_p)(-x_p) = 0,$$

$$\text{which simplifies to } a\sum_{p=1}^{n} x_p + b\sum_{p=1}^{n} x_p^2 = \sum_{p=1}^{n} x_p y_p. \tag{4.4}$$

Equations (4.3) and (4.4) can be solved simultaneously to find the values of a and b. We illustrate the use of these two equations with an example.

Example A. Consider the four pieces of data (x_p, y_p) shown below. The goal is to fit a straight line. The values of x_p have been chosen by the analyst and the values of y_p have been obtained from simulation with decision variable x_p. The values are

$$(50, 12), (70, 15), (100, 21), \text{ and } (120, 25).$$

Then $\sum_{p=1}^{4} x_p = 340, \sum_{p=1}^{4} y_p = 73, \sum_{p=1}^{4} x_p y_p = 6{,}750, \sum_{p=1}^{4} x_p^2 = 31{,}800$. Then using Eqs. (4.3) and (4.4), we have:

$$4a + 340b = 73 \text{ and } 340a + 31{,}800b = 6{,}750$$

which when solved yield $a = 2.2759$, and $b = 0.1879$. Thus the metamodel is: $y = 2.2759 + 0.1879x$.

3.2.2 Fitting Planes and Hyper-Planes

When we have data from $\Re^N$ spaces, where $N \geq 4$, it is not possible to *visualize* what a linear form (or any form for that matter) will look like. When $N = 3$, a linear form is a plane:

$$\phi = a + bx + cy. \tag{4.5}$$

A linear form in a space where $N \geq 4$ is called a **hyper-plane**. When we have three or more decision variables and a linear metamodel to tackle, it is the hyper-plane that needs to be fitted. An example of a hyper-plane with three decision variables is: $\phi = a + bx + cy + dz$.

In a manner analogous to that shown in the previous section, the following four equation can be derived.

$$na + b\sum_{p=1}^{n} x_p + c\sum_{p=1}^{n} y_p + d\sum_{p=1}^{n} z_p = \sum_{p=1}^{n} \phi_p,$$

$$a\sum_{p=1}^{n} x_p + b\sum_{p=1}^{n} x_p^2 + c\sum_{p=1}^{n} x_p y_p + d\sum_{p=1}^{n} x_p z_p = \sum_{p=1}^{n} x_p \phi_p,$$

$$a\sum_{p=1}^{n} y_p + b\sum_{p=1}^{n} y_p x_p + c\sum_{p=1}^{n} y_p^2 + d\sum_{p=1}^{n} y_p z_p = \sum_{p=1}^{n} y_p \phi_p,$$

and

$$a\sum_{p=1}^{n} z_p + b\sum_{p=1}^{n} z_p x_p + c\sum_{p=1}^{n} z_p y_p + d\sum_{p=1}^{n} z_p^2 = \sum_{p=1}^{n} z_p \phi_p.$$

Here, as before, n denotes the number of data-pieces. In general, a hyper-plane with N decision variables needs $(N + 1)$ linear equations since it is defined by $(N + 1)$ unknowns. The plane with $N = 3$ is a special case of the hyper-plane.

3.2.3 Piecewise Regression

Sometimes, the objective function is non-linear, but we wish to approximate it by a piecewise *linear* function. A piecewise linear function is not a continuous function; rather, it is defined by a unique linear function in each domain (piece). See Fig. 4.2. When we have a non-linear function of this kind, we divide the function space into finite areas or volumes or hyper-spaces (depending on the dimension of the space), and then fit, respectively, a straight line, a plane, or a hyper-plane in each. For example, consider a function, in $\Re^2$ space, defined by:

$$y = 6x + 4 \text{ when } 0 \leq x \leq 4,$$

$$y = 2x + 20 \text{ when } 4 < x \leq 7,$$

$$y = -2x + 48 \text{ when } 7 < x \leq 10, \text{ and}$$

$$y = -11x + 138 \text{ when } x > 10.$$

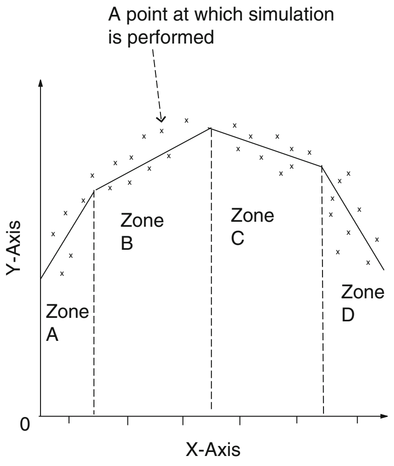

Figure 4.2. Fitting a piecewise linear function

Fitting this function would require the obtaining of a straight-line fit in each of the four zones: $(0, 4], (4, 7], (7, 10]$, and $(10, \infty)$. In piecewise regression, one can also use non-linear pieces such as quadratics or higher-order non-linear forms. (Please note that Fig. 4.2 represents a *similar* function.)

3.2.4 Fitting Non-linear Forms

In this section, we address the issue of how to tackle a non-linear form using regression. Regardless of the form of the objective function, our mechanism is analogous to what we have seen above.

Consider the function given by:

$$y = a + bx + cx^2. \tag{4.6}$$

This form can be expressed in the form of a plane

$$y = a + bx + cz, \text{ by setting: } z = x^2.$$

With this replacement, the equations of the plane can be used for the metamodel. See Fig. 4.3 for a non-linear form with one independent variable and Fig. 4.4 for a non-linear form with two independent variables. Other non-linear forms can be similarly obtained by using the mechanism of regression explained in the case of a straight line or plane.

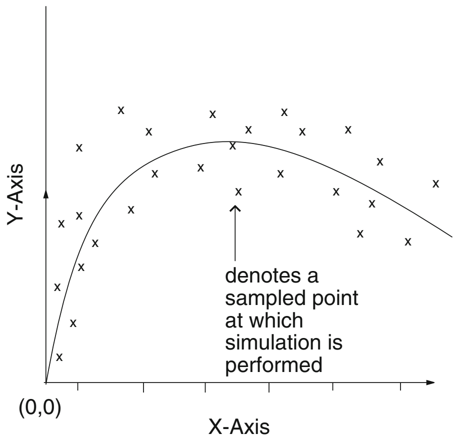

Figure 4.3. Fitting a non-linear equation with one independent variable

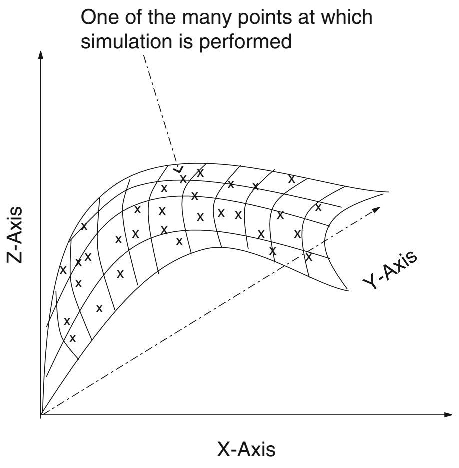

Figure 4.4. Fitting a non-linear equation with two independent variables

3.3. How Good Is the Guessed Metamodel?

An important issue in RSM is to determine whether the guessed metamodel is indeed a good fit. Testing whether the fit is reliable has deep statistical ramifications. We will not discuss these issues in detail here but refer the reader to any standard text on statistics, e.g. [209]. Here, we will restrict ourselves to enumerating some standard concepts.

A parameter that is often used in a **preliminary** test for the goodness of a fit in regression goes by the name: **coefficient of determination**. Its use may be extended to neural network models. It is defined as follows:

$$r^2 = 1 - (SSE/SST),$$

$$\text{where } SST \text{ is given by: } SST = \sum_{p=1}^{n} (y_p - \bar{y})^2,$$

$$\text{and } SSE \text{ is defined as: } SSE = \sum_{p=1}^{n} (y_p - y_p^{predicted})^2.$$

In the above, $\bar{y}$ is the mean of the y_p terms and $y_p^{predicted}$ is the value predicted by the model for the pth data piece. We have defined SSE during the discussion on fitting straight lines and planes. Note that those definitions were special cases of the definition given here.

Now, r^2, denotes the proportion of variation in the data that is explained by the metamodel assumed in calculating SSE. Hence a large value of r^2 (i.e., a value close to 1) usually indicates that the metamodel assumed is a good fit. *In a very rough sense*, the reliability of the r^2 parameter increases with the value of n. However, the r^2 parameter can be misleading if there are several variables and hence should be used cautiously. The reader is also refereed to [173] for further reading on this topic.

3.4. Optimization with a Metamodel

Once a satisfactory metamodel is obtained, it is usually easy to find the optimal point on the metamodel. With most metamodels one can use calculus to find the minima or maxima. When a piecewise linear form is used, usually the endpoints of each piece are candidates for minima or maxima. The following example should serve as an illustration.

Example B. Consider a function that has been fitted with four linear functions. The function is to be maximized. The metamodel is given by:

$$y = 5x + 6 \text{ when } 0 \leq x \leq 5,$$

$$y = 3x + 16 \text{ when } 5 < x \leq 10,$$

$$y = -4x + 86 \text{ when } 10 < x \leq 12, \text{ and}$$

$$y = -3x + 74 \text{ when } x > 12.$$

It is not hard to see that the peak is at $x = 10$ around which the steepest (slope) changes its sign. Thus $x = 10$ is the optimal point.

The approach used above is quite crude. It can be made more sophisticated by adding a few stages to it. The response surface method is often used in a bunch of stages to make it more effective. One first uses a rough metamodel (possibly piecewise linear) to get a general idea of the region in which the optimal point(s) may lie (as shown above via Example B). Then one zeroes in on that region and uses a more non-linear metamodel in that region. In Example B, the optimal point is likely to lie in the region close to $x = 10$. One can now take the next step, which is to use a non-linear metamodel *around* 10. This form can then be used to find a more precise location of the optimum. It makes a lot of sense to use more replications in the second stage than in the first. A multi-stage approach can become quite time-consuming, but more reliable.

Remark: Regression is often referred to as a **model-based** method because it assumes the knowledge of the metamodel for the objective function.

In the next section, we will study a **model-free** mechanism for function-fitting—the neural network. Neural networks are of two types—linear and non-linear. It is the non-linear neural network that is model-free.

4. Neuro-Response Surface Methods

One of the most exciting features of the non-linear neural network is its ability to approximate *any given function.* It is for this reason that neural networks are used in a wide range of areas ranging from cutting force measurement in metal-cutting to cancer diagnosis. No matter where neural networks are used, they are used for function fitting.

Neural networks are used heavily in the area of pattern recognition. In *many* pattern recognition problems, the basic idea is one of function fitting. Once one is able to fit a function to data, a "pattern" is said to have been recognized. This idea can be generally extended to a large number of scenarios. However, just because a problem has a pattern recognition flavor but does not need function fitting, neural networks should not be used on it.

The open literature reports the failure of neural networks on many problems. Usually, the reasons for this can be traced to the misuse of the method in some form. For example, as mentioned above, neural networks should not be used simply because the problem under consideration has a pattern recognition flavor but has nothing to do with function fitting.

As we will see shortly, the theory of neural networks is based on very sound mathematics. It is, in fact, an alternative way of doing regression. It makes clever use of the chain rule of differentiation and a well-known non-linear programming technique called steepest descent. In what follows, we will first discuss **linear neural networks**—also called **neurons** and then **non-linear neural networks**, which use the famous **backpropagation** algorithm.

4.1. Linear Neural Networks

The linear neural network (also called a neuron) is not model-free; it is model-based and assumes a linear model. The neuron is run by an algorithm that performs **linear** regression **without solving** any linear systems of equations. Convergence of this algorithm to an optimal solution can be proved. Also, there is strong empirical backing for this algorithm. Although the algorithm has several names such as delta, adaline, and least mean square, we will call it the Widrow-Hoff algorithm in honor of its inventors [323]. We will, next, derive the Widrow-Hoff (WH) algorithm.

Recall that the goal underlying a regression problem for a hyperplane of the order N is to obtain a fit for the linear equation of the form:

$$y = w(0) + w(1)x(1) + w(2)x(2) + \cdots + w(N)x(N).$$

To obtain a straight line, we would have to set $N = 1$. Recall the definitions of a and b from Eq. (4.1). Then, $w(0)$ would correspond to a and $w(1)$ to b. Similarly, for the plane, $N = 2$, $w(2)$ would correspond to c in Eq. (4.5).

We will now introduce a subscript in the following terms: $x(i)$ and y. The notation $x_p(i)$ will denote the value of $x(i)$ in the pth data piece. Similarly, y_p will denote the function value in the pth data piece. Now, using this notation, the SSE can be written as:

$$SSE \equiv \sum_{p=1}^{n}[y_p - w(0) - w(1)x_p(1) - w(2)x_p(2) - \cdots - w(N)x_p(N)]^2. \tag{4.7}$$

For obtaining neural network algorithms, we minimize $SSE/2$ rather than minimizing SSE. The reason will be clear shortly. It amounts to the same thing since minimization of one clearly ensures the minimization of the other.

Now, the WH algorithm is essentially a non-linear programming algorithm in which the function to be minimized is $SSE/2$ and the decision variables are the $w(i)$ terms, for $i = 0, 1, \ldots, N$. The topic of non-linear programming will be discussed in more detail in Chap. 5. Here, we present an important and popular steepest-based algorithm for solving a non-linear program. If the non-linear programming problem is described as follows:

Minimize $f(\vec{x})$ where $\vec{x} = \{x(1), x(2), \ldots, x(N)\}$ is an N-dimensional vector,

then the main transformation in the steepest-descent algorithm is given by:

$$x(i) \leftarrow x(i) - \mu \frac{\partial f(\vec{x})}{\partial x(i)}, \qquad \text{for each } i, \tag{4.8}$$

where μ is a step size that diminishes to 0.

Thus to derive a steepest-descent algorithm for $SSE/2$, we need to find the partial derivatives of $SSE/2$ with respect to each of the $w(i)$ terms. This is precisely what Widrow and Hoff [323] did. Let us see how it was done.

Now, using Eq. (4.7), we have

$$\begin{aligned}
\frac{\partial E}{\partial w(i)} &= \frac{1}{2}\sum_{p=1}^{n} \frac{\partial}{\partial w(i)} [y_p - w(0)x_p(0) - w(1)x_p(1) - \cdots - w(N)x_p(N)]^2 \\
&= \frac{1}{2}\sum_{p=1}^{n} 2\,(y_p - o_p)\, \frac{\partial}{\partial w(i)} [y_p - w(0)x_p(0) \\
&\quad -w(1)x_p(1) - \cdots - w(N)x_p(N)] \text{ [setting } o_p \equiv w(0)x_p(0) \\
&\quad +w(1)x_p(1) + \cdots + w(N)x_p(N)] \\
&= \sum_{p=1}^{n} (y_p - o_p)\,[-x_p(i)] = -\sum_{p=1}^{n} (y_p - o_p)\,[x_p(i)].
\end{aligned}$$

$$\text{Thus } \frac{\partial E}{\partial w(i)} = -\sum_{p=1}^{n} (y_p - o_p)\, x_p(i). \tag{4.9}$$

Using Eq. (4.9) and transformation (4.8), the WH algorithm becomes:

$$w(i) \leftarrow w(i) + \mu \sum_{p=1}^{n} (y_p - o_p)\, x_p(i), \tag{4.10}$$

where $o_p = w(0)x_p(0) + w(1)x_p(1) + \cdots + w(N)x_p(N)$.

Advantages of the WH Algorithm. The advantages of the WH algorithm over regression are not immediately obvious. First and foremost, one does not have to solve linear equations unlike regression. For hyper-planes with large values of N, this can mean considerable savings in the memory needed for the computer program. For instance, to obtain a regression fit for a problem with 100 variables, one would have to set up a matrix of the size of 101×101 and then use a linear equation solving algorithm, whereas the WH algorithm would need to store only 101 $w(i)$ terms. The WH algorithm is guaranteed to converge as long as μ is sufficiently small.

As we will see later, the WH algorithm can also be used for **incremental** purposes—that is, when the data pieces become available *one by one* and not at the same time. This is seen in reinforcement learning (control optimization). In other words, (4.10) can be used with $n = 1$. We will see this shortly.

We must note here that regression too can be done incrementally and if the order of the hyper-plane is not big, (i.e., for small N), the Widrow-Hoff algorithm does not seem to possess any advantage over regular regression discussed in previous sections.

We next present a step-by-step description of the Widrow-Hoff algorithm.

4.1.1 Steps in the Widrow-Hoff Algorithm

Let us assume that n data pieces are available. Some termination criterion has to be used for stopping the algorithm. One termination criterion assumes that if the absolute value of the difference between the values of SSE computed in successive iterations is "negligible," the algorithm has converged. How small is negligibly small is left to the user. This quantity is usually referred to as *tolerance*. Another possible termination criterion runs the algorithm till the step size becomes negligibly small.

Step 1: Set all the $w(i)$ terms (for $i = 0, 2, \ldots, N$) to small random numbers (preferably between 0 and 1). Set SSE_{old} to a large number. The available data for the pth piece is $(y_p, \vec{x}_p)$ where $\vec{x}_p$ is a vector with N components. Set $x_p(0) = 1$ for all p. Set *tolerance* to a small value. Set $m = 0$.

Step 2: Compute o_p for $p = 1, 2, \ldots, n$ using

$$o_p = \sum_{j=0}^{k} w(j) x_p(j).$$

Step 3: Update each $w(i)$ for $i = 0, 1, \ldots, N$ using:

$$w(i) \leftarrow w(i) + \mu \sum_{p=1}^{n} (y_p - o_p)\, x_p(i).$$

Step 4: Increment m by 1. Calculate SSE_{new} using

$$SSE_{new} = \sum_{p=1}^{n} (y_p - o_p)^2 .$$

Update μ using m (see below for updating rules). If $|SSE_{new} - SSE_{old}| < tolerance$, STOP. Otherwise, set $SSE_{old} = SSE_{new}$, and then go back to Step 2.

Updating rules for step-size: The step-size μ should decay with m. Some popular rules are:

$$\mu = \frac{A}{B + m} \text{ and } \mu = \frac{log(m)}{m},$$

where A and B are scalars, e.g., $A = 500$ and $B = 1{,}000$. Another rule that is quite popular is the Darken-Chang-Moody rule [71]. Chapter 7 discusses step-size rules in more detail.

4.1.2 Incremental Widrow-Hoff

As discussed in Sect. 4.1, an incremental version of the Widrow-Hoff algorithm is sometimes useful. The algorithm presented in the previous subsection is usually called a *batch* algorithm because it uses the entire batch of data simultaneously. The incremental algorithm is obtained from the batch algorithm. This is done by replacing the sum over all values of p by just one quantity. The incremental algorithm is also convergent as long as we use small values for the step size μ. We present a step-by-step description, next.

Step 1: Set all the $w(i)$ terms (for $i = 0, 2, \ldots, N$) to small random numbers (between 0 and 1). The available data for the pth piece is $(\vec{x}_p, y_p)$ where $\vec{x}_p$ is a vector with N components. Set $x_p(0) = 1$ for all values of p. Set $m_{\max}$ to the max number of iterations for which the algorithm is to be run. Set *tolerance* to a small value, and set m to 0.

Step 2: For each value of p from 1 to n, execute the following steps.

Step 2a: Compute o_p using

$$o_p = \sum_{j=0}^{N} w(j)x_p(j).$$

Step 2b: Update each $w(i)$ for $i = 0, 1, \ldots, N$ using:

$$w(i) \leftarrow w(i) + \mu\,(y_p - o_p)\,x_p(i).$$

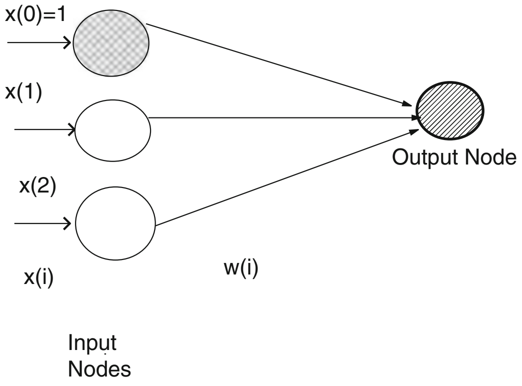

Figure 4.5. A **linear** network—a neuron with three input nodes and one output node: The approximated function is a plane with two independent variables: $x(1)$ and $x(2)$. The node with input $x(0)$ assumes the role of the constant a in regression

Step 3: Increment m by 1. Update μ using m as discussed above. If $m < m_{\max}$, return to Step 2. Otherwise, STOP.

The neuron can be represented pictorially as shown in Fig. 4.5. The circles denote the *nodes*. There is one node for each independent variable and one node for the constant term—the term a in regression (See Eq. (4.1)). The input to the ith node is denoted by $x(i)$.

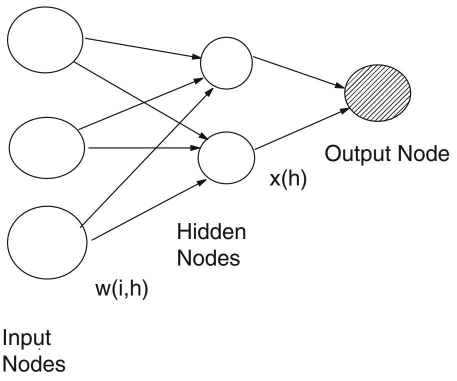

Figure 4.6. A **non-linear** neural network with an input layer, one hidden layer and one output node: The term $w(i, h)$ denotes a weight on the link from the ith input node to the hth hidden node. The term $x(h)$ denotes the weight on the link from the hth hidden node to the output node

4.2. Non-linear Neural Networks

In this section, we will discuss the backpropagation (often abbreviated as **backprop**) algorithm that helps us perform function fitting for non-linear metamodels. To visualize how a non-linear neural network works, consider Fig. 4.6. It shows a neural network with three layers—an *input* layer, a *hidden* layer and an *output* layer. The output layer contains one node. The input nodes are connected to each of the nodes in the hidden layer, and each node in the hidden layer is connected to the output node. All connections are made via unidirectional links.

Neural networks with more than one output node are not needed for regression purposes. In artificial intelligence, we must add here, neural networks with more than one output are used regularly, but their scope is outside of regression. We stick to a regression viewpoint in this book, and therefore avoid multiple outputs. One output in a neural network implies that only one function can be fitted with that neural network.

Neural networks with multiple hidden layers have also been proposed in the literature. They make perfect mathematical sense.

Multiple hidden layers are often needed to model highly non-linear functions. We do not discuss their theory. We would also like to add that multiple hidden layers increase the computational burden of the neural network and make it slower.

4.2.1 The Basic Structure

The input layer consists of a finite number of nodes. One input node is usually associated with each independent variable. Hence, the number of nodes usually equals the number of variables in the function-fitting process.

Usually, we also use one extra node in the input layer. This is called the **bias** node. It takes care of the constant term in a metamodel. It is directly connected to the output node. In our discussion in this section, we will not take this node into consideration. We will deal with it in Sect. 4.2.4.

There is no fixed rule on how to select the number of nodes in the hidden layer. In a rough sense, the more non-linear the function to be fitted, the larger the number of hidden nodes needed. As we will see, the hidden layer is the entity that makes a neural network non-linear.

Before understanding how the neural network implements the backprop algorithm, we must understand how it predicts function values at any given point *after* the backprop algorithm has been used on it. We also need to introduce some notation at this point.

The connecting arrows or links have numbers associated with them. These scalar values are called **weights** (see Fig. 4.6). The neural network backprop algorithm has to derive the right values for each of these weights. Let us denote by $w(i,h)$ the weight on the link from the ith input node to the hth hidden node. Each input node is fed with a value equal to the value of the associated independent variable at that point. Let us denote the value fed to the jth input node by $u(j)$. Then the raw value of the hth hidden node is given by:

$$v^*(h) = \sum_{j=1}^{I} w(j,h)u(j),$$

where I denotes the number of input nodes.

However, this is not the actual value of the hidden node that the backprop algorithm uses. This *raw* value is converted to a so-called "thresholded" value. The thresholded value lies between 0 and 1, and is generated by some function. An example of such a function is:

$$\textbf{Thresholded value} = \frac{1}{1+e^{-\textbf{ Raw value}}}.$$

The above function goes by the name *sigmoid* function. There are other functions that can be used for thresholding. We will understand the role played by functions such as these when we derive the backprop algorithm.

Thus the actual value of the hth hidden node, $v(h)$, using the sigmoid function, is given by:

$$v(h) = \frac{1}{1 + e^{-v^*(h)}}.$$

Let $x(h)$ denote the weight on the link from the hth hidden node to the output node. Then the output node's value is given by:

$$o = \sum_{h=1}^{H} x(h)v(h), \tag{4.11}$$

where $v(h)$ denotes the actual (thresholded) value of the hth hidden node and H denotes the number of hidden nodes. Now we will demonstrate these ideas with a simple example.

Example C: Let us consider a neural network with three input nodes, two hidden nodes, and one output node, as shown in Fig. 4.7. Let the input values be: $u_1 = 0.23, u_2 = 0.43$, and $u_3 = 0.12$. Let the weights from the input node to the hidden nodes be: $w(1,1) = 1.5, w(1,2) = 4.7, w(2,1) = 3.7, w(2,2) = 8.9, w(3,1) = 6.7$ and $w(3,2) = 4.8$. Let the weights from the hidden nodes to the output node be $x(1) = 4.7$ and $x(2) = 8.9$. Then using the formulas given above:

$$v^*(1) = \sum_{i=1}^{3} w(i,j)u(i) = (1.5)(0.23) + (3.7)(0.43) + (0.67)(0.12) = 2.74$$

Similarly, $v^*(2) = 5.484$. Then:

$$v(1) = \frac{1}{1 + e^{-v^*(1)}} = \frac{1}{1 + e^{-2.74}} = 0.9393.$$

Similarly, $v(2) = 0.9959$. Then:

$$o = x(1)v(1) + x(2)v(2) = (4.7)(0.9393) + (8.9)(0.9959) = 13.2782.$$

Remark 1. From the above example, one can infer that large values for inputs and the $w(i,h)$ terms will produce $v(h)$ values that are very close to 1. This implies that for large values of inputs (and weights), the network will lose its discriminatory power. It turns out that even

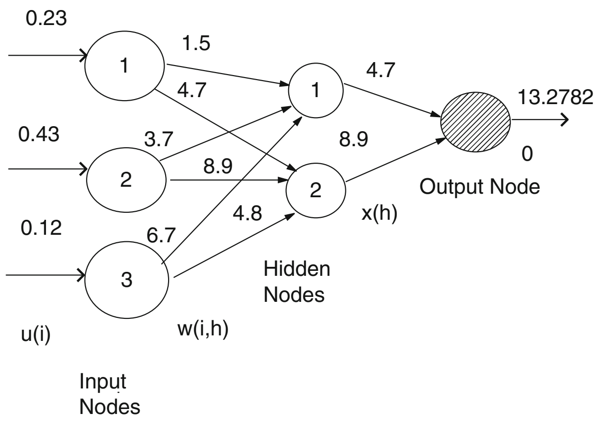

Figure 4.7. Deriving the value of the output for given values of inputs and weights

for values such as 50, we have $\frac{1}{1+e^{-50}} \approx 1$. If all data pieces are in this range, then all of them will produce the same output o for a given set of weights. And this will not work. One way out of this problem is to use the following trick. **Normalize** the raw inputs to values between 0 and 1. Usually the range of values that the input can take on is known. So if the minimum possible value for the ith input is $a(i)$ and the maximum is $b(i)$ then we should first normalize our data using the following principle:

$$u(i) = \frac{u_{raw}(i) - a(i)}{b(i) - a(i)}.$$

So for example, if values are: $u_{raw}(1) = 2, a(1) = 0$, and $b(1) = 17$, then the value of $u(1)$ to be fed into the neural network should be:

$$u(i) = \frac{2-0}{17-0} = 0.117647.$$

Remark 2. An alternative way to work around this difficulty is to modify the sigmoid function as shown below:

$$v = \frac{1}{1+e^{-v^*/M}}, \text{ where } M > 1.$$

This produces a somewhat similar effect but then one must choose M carefully.

$$M = \max_i b(i) - \min_i a(i) \text{ works in practice.}$$

Remark 3. The $w(i, h)$ terms should also remain at small values for retaining the discriminatory power of the neural network. As we will see later, these terms are in the danger of becoming too large. We will discuss this issue later again.

The subscript p will be now used as an index for the data piece. (If the concept of a "data piece" is not clear, we suggest you review Example A in Sect. 3.2.1.) Thus y_p will denote the function value of the pth data piece obtained from simulation. For the same reason, $v_p(h)$ will denote the value of the hth hidden node when the pth data piece is used as an input to the node. So also, $u_p(i)$ will denote the value of the input for the ith input node when the pth data piece is used as input to the neural network. The notations $w(i, h)$ and $x(h)$, however, will never carry this subscript because they do not change with every data piece.

4.2.2 The Backprop Algorithm

Like in regression, the objective of the backprop algorithm is to minimize the sum of the squared differences between actual function values and the predicted values. Recall that in straight line fitting, the regression error was defined as:

$$e_p = y_p - (a + bx_p). \tag{4.12}$$

Now, in the context of backprop, we assume that we do not know the model. So clearly, we do not have access to quantities such as $a + bx_p$ in the equation given above. Notice that this quantity is the *predicted value* of the function. Hence we will replace it by a variable o_p. Hence the error term, regardless of what function is to be fitted, can be expressed as:

$$e_p = y_p - o_p. \tag{4.13}$$

We repeat: the reason for using the term o_p instead of the actual form is: we want to fit the function without assuming any metamodel.

Therefore SSE becomes:

$$SSE = \sum_{p=1}^{n} (y_p - o_p)^2 \tag{4.14}$$

where n is the total number of data pieces available. For the backprop algorithm, instead of minimizing SSE, we will minimize $SSE/2$. (If $SSE/2$ is minimized, SSE will be minimized too.) In the following section, we will discuss the derivation of the backprop algorithm.

The following star-marked section can be skipped without loss of continuity in the first reading.

4.2.3 Deriving Backprop *

The backprop algorithm is essentially a non-linear programming algorithm in which the function to be minimized is $SSE/2$ and the decision variables of the non-linear program are the weights—the $w(i,h)$ and $x(h)$ terms. If the non-linear programming problem is described as follows:

Minimize $f(\vec{x})$ where $\vec{x} = \{x(1), x(2), \ldots, x(N)\}$ is an N-dimensional vector,

then the main transformation in a steepest-descent algorithm is given by:

$$x(i) \leftarrow x(i) - \mu \frac{\partial f(\vec{x})}{\partial x(i)}, \qquad \text{for all } i, \tag{4.15}$$

where μ denotes a step size that diminishes to 0. For the backprop algorithm, $f(\vec{x})$ is $SSE/2$, and $\vec{x}$ is made up of the $w(i,h)$ and $x(h)$ terms. Using the transformation defined in (4.15), for the $w(i,h)$ terms, we have:

$$w(i,h) \leftarrow w(i,h) - \mu \frac{\partial}{\partial w(i,h)}(SSE/2), \tag{4.16}$$

and for the $x(h)$ terms we have:

$$x(h) \leftarrow x(h) - \mu \frac{\partial}{\partial x(h)}(SSE/2). \tag{4.17}$$

We will denote $SSE/2$ by E. We now need to derive expressions for the partial derivatives: $\frac{\partial E}{\partial x(h)}$ and $\frac{\partial E}{\partial w(i,h)}$ in Eqs. (4.17) and (4.16) respectively. This is what we do next.

Using Eq. (4.14), we have

$$\begin{aligned}
\frac{\partial E}{\partial x(h)} &= \frac{1}{2}\sum_{p=1}^{n} \frac{\partial}{\partial x(h)} (y_p - o_p)^2 \\
&= \frac{1}{2}\sum_{p=1}^{n} 2\,(y_p - o_p)\,\frac{\partial}{\partial x(h)}\,(y_p - o_p) \\
&= \sum_{p=1}^{n} (y_p - o_p)\left(-\frac{\partial o_p}{\partial x(h)}\right) \\
&= -\sum_{p=1}^{n} (y_p - o_p)\,(v_p(h)).
\end{aligned}$$

The last equation follows from the fact that $o_p = \sum_{i=1}^{H} x(i)v_p(i)$, where H is the number of hidden nodes. Thus we can conclude that:

$$\frac{\partial E}{\partial x(h)} = -\sum_{p=1}^{n} (y_p - o_p)\, v_p(h). \tag{4.18}$$

Similarly, the derivative with respect to $w(i,h)$ can be derived as follows:

$$\begin{aligned}
\frac{\partial E}{\partial w(i,h)} &= \frac{1}{2}\sum_{p=1}^{n} \frac{\partial}{\partial w(i,h)} (y_p - o_p)^2 \\
&= \frac{1}{2}\sum_{p=1}^{n} 2\,(y_p - o_p)\, \frac{\partial\,(y_p - o_p)}{\partial w(i,h)} \\
&= \sum_{p=1}^{n} (y_p - o_p) \left(-\frac{\partial o_p}{\partial w(i,h)}\right) \\
&= \sum_{p=1}^{n} (y_p - o_p) \left(-\frac{\partial o_p}{\partial v_p(h)}.\frac{\partial v_p(h)}{\partial w(i,h)}\right) \\
&= -\sum_{p=1}^{n} (y_p - o_p) \left(x(h).\frac{\partial v_p(h)}{\partial w(i,h)}\right) \text{ since } o_p = \sum_{i=1}^{H} x(i)v_i(p) \\
&= -\sum_{p=1}^{n} (y_p - o_p) \left(x(h).\frac{\partial v_p(h)}{\partial v_p^*(h)}.\frac{\partial v_p^*(h)}{\partial w(i,h)}\right) \\
&= -\sum_{p=1}^{n} (y_p - o_p)\, x(h)v_p(h)\,(1 - v_p(h))\, u_p(i).
\end{aligned}$$

The last equation follows from that facts that

$$v_p^*(h) = \sum_{j=1}^{I} w(j,h)u_p(j) \text{ implies } \frac{\partial v_p^*(h)}{\partial w(i,h)} = u_p(i);$$

$$v_p(h) = \frac{1}{1 + e^{-v_p^*(h)}} \text{ implies } \frac{\partial v_p(h)}{\partial v_p^*(h)} = v_p(h)[1 - v_p(h)].$$

Thus in conclusion,

$$\frac{\partial E}{\partial w(i,h)} = -\sum_{p=1}^{n} [y_p - o_p]x(h)v_p(h)[1 - v_p(h)]u_p(i). \tag{4.19}$$

The backprop algorithm can now be defined by the following two main transformations.

1. Transformation (4.17), which can now be written, using Eq. (4.18), as:
$$x(h) \leftarrow x(h) + \mu \sum_{p=1}^{n} (y_p - o_p)\, v_p(h). \tag{4.20}$$

2. And transformation (4.16), which can now be written, using Eq. (4.19), as:
$$w(i,h) \leftarrow w(i,h) + \mu \sum_{p=1}^{n} (y_p - o_p)\, x(h) v_p(h)\,(1 - v_p(h))\, u_p(i). \tag{4.21}$$

Remark 1: A nice thing about this algorithm is that closed form expressions for the partial derivatives could be derived.

Remark 2: Any steepest-descent algorithm can only be guaranteed to converge to *local optima*. If multiple optima exist, convergence to the global optimum cannot be ensured with a steepest-descent algorithm. Naturally, backprop, being a steepest-descent algorithm, suffers from this drawback. However, in practice, the problem of getting trapped in local optima has not been found to be too menacing. Furthermore, there are ways of working around this problem.

Remark 3: The algorithm derived above uses a sigmoid function for thresholding. Another function that has also been used by researchers is the *tanh* function.

We next discuss how we can deal with a bias node in the backprop algorithm.

4.2.4 Backprop with a Bias Node

The idea underlying the so-called bias node is to assume that the function has a constant term—a term that corresponds to the term, a, in regression theory, which can be found in Eqs. (4.1) or (4.5). (It is always acceptable to do so because if the true function to be fitted does not have such a term, the weight associated with the bias node will converge to 0.) This is taken care of by assuming that there is an extra node—the bias node—that is connected directly to the output node. See Fig. 4.8.

Let us denote the bias weight by b. The input to the bias node is **always 1** or some constant value. Hence the output node o should now be defined as:

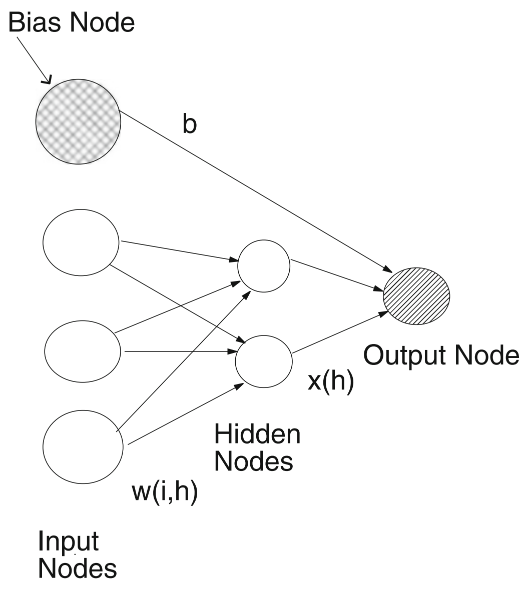

Figure 4.8. A neural network with a bias node: The topmost node is the bias node. The weight on the direct link to the output node is b

$$o = (b)(1) + \sum_{h=1}^{H} x(h)v(h). \tag{4.22}$$

The following star-marked section can be skipped without loss of continuity in the first reading.

4.2.5 Deriving Backprop with a Bias Node *

We will next derive a steepest-descent transformation for the bias weight. Notice the difference between Eqs. (4.22) and (4.11). Change in the definition of o will not alter $\frac{\partial E}{\partial x(h)}$ that we have derived earlier. Of course $\frac{\partial E}{\partial w(i,h)}$ will not change either. Now, $\frac{\partial E}{\partial b}$ can be derived in a manner identical to that of $\frac{\partial E}{\partial x(h)}$. We will show the details, next.

$$\begin{aligned}\frac{\partial E}{\partial b} &= \frac{1}{2}\sum_{p=1}^{n}\frac{\partial}{\partial b}\left(y_p - o_p\right)^2 \\ &= \frac{1}{2}\sum_{p=1}^{n} 2\left(y_p - o_p\right)\frac{\partial}{\partial b}\left(y_p - o_p\right)\end{aligned}$$

$$= \sum_{p=1}^{n} (y_p - o_p) \left(-\frac{\partial o_p}{\partial b} \right)$$

$$= -\sum_{p=1}^{n} (y_p - o_p)\, 1.$$

The last equation follows from the fact that $o_p = b + \sum_{i=1}^{H} x(i) v_p(i)$. Thus,

$$\frac{\partial E}{\partial b} = -\sum_{p=1}^{n} (y_p - o_p)\,. \qquad (4.23)$$

The steepest-descent transformation for the bias weight, which is given by

$$b \leftarrow b - \mu \frac{\partial E}{\partial b},$$

can be written, using (4.23), as:

$$b \leftarrow b + \mu \sum_{p=1}^{n} (y_p - o_p)\,. \qquad (4.24)$$

The backprop algorithm with a bias node is defined by three transformations: (4.20), (4.21), and (4.24). In the next section, we present a step-by-step description of a version of the backprop algorithm that uses the sigmoid function.

4.2.6 Steps in Backprop

The backprop algorithm is like any other iterative steepest-descent algorithm. We will start with arbitrary values for the weights—usually small values close to 1. Then we will use the transformations defined by (4.20), (4.21), and (4.24) repeatedly till the SSE is minimized to an acceptable level. One way to terminate this iterative algorithm is to stop when the difference in the value of SSE obtained in successive iterations (step) becomes negligible.

What we present below is also called the “batch-updating” version of the algorithm. The reason underlying this name is that all the data pieces are simultaneously (batch mode) used in the transformations. When we use a transformation and change a value, we can think of the change as an *update* of the value. The step-by-step description is presented below.

Let us define some quantities and recall some definitions. H will denote the number of hidden nodes, and I will denote the number of input nodes. (Note that I will include the bias node.) The algorithm will be terminated when the absolute value of the difference between the SSE in successive iterations is less than $tolerance$—a pre-specified small number, e.g., 0.001.

Step 1: Set all weights—that is, $w(i,h), x(h)$, and b for $i = 1, 2, \ldots, I$, and $h = 1, 2, \ldots, H$, to small random numbers between 0 and 0.5. Set the value of SSE_{old} to a large value. The available data for the pth piece is $(\vec{u}_p, y_p)$ where $\vec{u}_p$ denotes a vector with I components. Set m to 0.

Step 2: Compute each of the $v_p^*(h)$ terms for $h = 1, 2, \ldots, H$ and $p = 1, 2, \ldots, n$ using

$$v_p^*(h) = \sum_{j=1}^{I} w(j,h) u_p(j).$$

Step 3: Compute each of the $v_p(h)$ terms for $h = 1, 2, \ldots, H$ and $p = 1, 2, \ldots, n$ using

$$v_p(h) = \frac{1}{1 + e^{-v_p^*(h)}}.$$

Step 4: Compute each of the o_p terms for $p = 1, 2, \ldots, n$ where n is the number of data pieces using

$$o_p = b + \sum_{h=1}^{H} x(h) v_p(h).$$

Step 5:

- Update b using: $b \leftarrow b + \mu \sum_{p=1}^{n} (y_p - o_p)$.
- Update each $w(i,h)$ using:

$$w(i,h) \leftarrow w(i,h) + \mu \sum_{p=1}^{n} (y_p - o_p)\, x(h) v_p(h) \left(1 - v_p(h)\right) u_p(i).$$

- Update each $x(h)$ using

$$x(h) \leftarrow x(h) + \mu \sum_{p=1}^{n} (y_p - o_p)\, v_p(h).$$

Step 6: Increment m by 1. Calculate SSE_{new} using $SSE_{new} = \sum_{p=1}^{n}(y_p - o_p)^2$. Update the value of μ using m as discussed above. If $|SSE_{new} - SSE_{old}| < tolerance$, STOP. Otherwise, set $SSE_{old} = SSE_{new}$, and then return to Step 2.

Remark 1: If one wishes to ignore the bias node, b should be set to 0 in Step 4, and the first transformation in Step 5 should be disregarded.

Remark 2: Clearly, the value of *tolerance* should not very low; otherwise, the algorithm may require a very large number of iterations.

Remark 3: Setting a very low value for *tolerance* and running the algorithm for too many iterations can cause *overfitting*, i.e., the fit predicts the function over the training data too closely. This is undesirable as it can lead to a fit that works well for the data over which the function has been fitted, but predicts poorly at points at which the function is unknown.

Remark 4: The parameter r^2 (see Sect. 3.3) used to evaluate the goodness of a fit can be computed in the case of a neural network too.

4.2.7 Incremental Backprop

Like in the case of the linear neural network, we will also discuss the **incremental** variant of the backprop algorithm. The incremental algorithm uses one data piece at a time. We will present the necessary steps in the incremental version. The behavior of the incremental version can be shown to become arbitrarily close to that of the batch version. Incremental backprop is usually not used for response surface optimization. It can be useful in control optimization. We will refer to it in the chapters related to reinforcement learning.

The incremental version is obtained from the batch version, presented in the previous section, by removing the summation over all data pieces. In other words, in the incremental version, the quantity summed in the batch version is calculated for one data piece at a time. The relevant steps are shown next.

Step 1: Set all weights—that is, $w(i,h), x(h)$, and b, for i=$1,2,\ldots,I$, $h = 1,2,\ldots,H$, to small random numbers. The available data for the pth piece is $(\vec{u}_p, y_p)$ where $\vec{u}_p$ denotes a vector with I components. Set m to 0 and $m_{\max}$ to the maximum number of iterations for which the algorithm is to be run.

Step 2: For each value of p, i.e., for $p = 1, 2, \ldots, n$, execute the following steps.

1. Compute $v_p^*(h)$ for $h = 1, 2, \ldots, H$ using

$$v_p^*(h) = \sum_{j=1}^{I} w(j,h)u_p(j).$$

2. Compute $v_p(h)$ for $h = 1, 2, \ldots, H$ using

$$v_p(h) = \frac{1}{1 + e^{-v_p^*(h)}}.$$

3. Compute o_p using $o_p = b + \sum_{h=1}^{H} x(h)v_p(h)$.

4. Update b using: $b \leftarrow b + \mu\,(y_p - o_p)$.
Update each $w(i,h)$ using:

$$w(i,h) \leftarrow w(i,h) + \mu\,(y_p - o_p)\,x(h)v_p(h)\,(1 - v_p(h))\,u_p(i).$$

Update each $x(h)$ using $x(h) \leftarrow x(h) + \mu\,(y_p - o_p)\,v_p(h)$.

Step 3: Increment m by 1, and update the value of μ as discussed above.

Step 4: If $m < m_{\max}$, go back to step 2; otherwise stop.

The advantage of incremental backprop over batch backprop lies in its lower computational burden. Since many quantities are summed over n points in batch backprop, the calculations can take considerable amount of computer time. However, with present-day computers, this is not much of an issue. The gradient in the incremental version is not accurate, as it is not summed over all the n data pieces. However, with **a small enough value for the step size** μ, the incremental version should closely approximate the behavior of the batch version.

Some neural network tricks:

1. In practice, the bias node is placed in the hidden layer and an additional bias node (virtual bias node) is placed in the input layer. It is connected to all the other nodes in the hidden layer. The bias node does not have any connection to the input layer. However, it

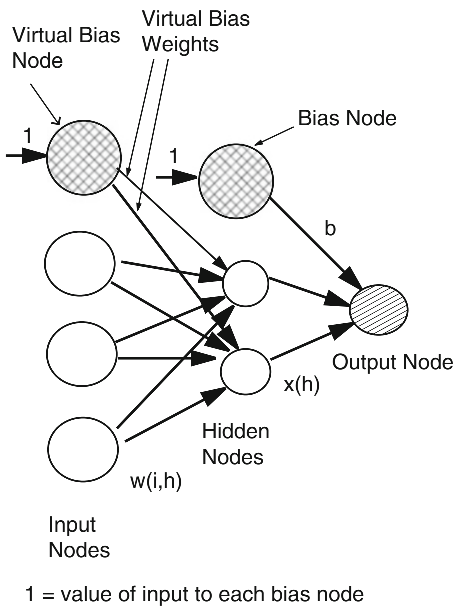

Figure 4.9. Actual implementation with a bias node

is connected to the output node. The input to each bias node is 1. This trick makes the net stable. See Fig. 4.9 for a picture. The online C codes [121] use this idea.

2. Since backprop is a steepest-descent algorithm and the error function, SSE, may contain multiple optima, getting trapped in local optima cannot be ruled out. This is a commonly experienced difficulty with steepest descent. One way around this is to start at a new point when the net does not perform well. This is a well-known trick. In the case of a neural network, the starting point is generated by the random values (between 0 and 1, typically) given to all the weights. Hence one should use a different SEED (see Chap. 2), and generate new values for the weights.

3. Choice of the number of hidden nodes needs some experimentation. Any arbitrary choice may not yield the best results. It is best to start with a small number of hidden nodes.

4. After performing the backprop for a very long time, the weights can become large. This can pose a problem for the net. (It loses its discriminatory power). One way out of this is to multiply each weight in each iteration by $(1-\frac{\mu\gamma}{2})$, where γ is another step size less than 1. This update should be performed after the regular backprop update. This idea has good theoretical backing; see Mitchell [205] for more on this.

Example D: Let us next consider a function the closed form of which is known. We will generate 10 points on this function and then use them to generate a neuro-response surface. Since the function is known, it will help us test the function at any arbitrary point.

Consider the function: $y = 3 + x_1^2 + 5x_1x_2$.

We generate 10 points on this function. The points are tabulated below.

Point	x_1	x_2	y
1	0.000783	0.153779	3.000602
2	0.560532	0.865013	5.738535
3	0.276724	0.895919	4.316186
4	0.704462	0.886472	6.618693
5	0.929641	0.469290	6.045585
6	0.350208	0.941637	4.771489
7	0.096535	0.457211	3.230002
8	0.346164	0.970019	4.798756
9	0.114938	0.769819	3.455619
10	0.341565	0.684224	4.285201

A neural network with backpropagation was trained on this data. Three hidden nodes were used and a bias node along with a virtual bias node (in the input layer) was used. See Fig. 4.9 to see how a virtual bias node is placed in a neural network. We will denote the weight on the link connecting the virtual bias node to the hth hidden node by $vb(h)$. As before, $x(h)$ will denote the weight from the hth hidden node to the output node and $w(i,h)$ will denote the weight from the ith input to the hth hidden node. The bias nodes have inputs of 1 and b will denote the bias weight (the weight on the link from the bias node in the hidden layer to the output node—see Fig. 4.9). The neural network was trained for 500 iterations. The best SSE was found to be 0.785247. The codes are provided at [121]. The $w(i,h)$ values are:

i	h	$w(i,h)$
1	1	2.062851
1	2	2.214936
1	3	2.122674
2	1	0.496078
2	2	0.464138
2	3	0.493996

The $x(h)$ and the $vb(h)$ weights are:

h	$x(h)$	$vb(h)$
1	2.298128	-0.961674
2	2.478416	-1.026590
3	2.383986	-0.993016

The bias weight is 0.820535. Since the function form is known, we can test how well the function has been approximated by the backprop algorithm. Some comparisons are tabulated below.

Point	x_1	x_2	y	$y^{predicted}$
1	0.2	0.5	3.54	3.82
2	0.3	0.9	4.44	4.54
3	0.1	0.9	3.46	3.78

Note that we have tested the function on input values between 0 and 1. As discussed previously, before training, it is best to normalize the input values to the range $(0, 1)$.

4.2.8 Neural Network Validation

Like in the case of regression-based response surface methods, it is important to "validate" the weights generated by the neural network—that is, check if the network predicts well on the data that was *not* used in generating the weights. Unless validation is performed, there is no guarantee of the network's performance. Unlike regression-based models, parameters such as r^2 may not be sufficient, and more sophisticated validation tests are needed.

In a commonly-used strategy, called **data splitting**, a data set of n points at which simulation has been performed is split into 2 sets—$\mathcal{S}_1$ and $\mathcal{S}_2$. $\mathcal{S}_1$ is used to generate the neural network weights and $\mathcal{S}_2$ is used to test the ability of the net to predict well. Then the absolute error, which is reflective of the net's performance, can be calculated as:

$$\text{Absolute Error } \equiv \max_{i \in \mathcal{S}_2} |y_i - y_i^{predicted}|.$$

Several other statistical tests have been described in the literature [302]. Some of these tests are called: re-substitution, cross-validation, and jackknife. (These tests can also be used in testing the validity of regression metamodels.) In many of these tests, the *absolute error* between the value predicted by the net and the actual value (defined above) is used, instead of the *squared error* discussed previously.

4.2.9 Optimization with a Neuro-RSM Model

The neuro-RSM model, once generated and proved to be satisfactory, can be used to evaluate the function at any given point. Then, it may be combined with any numerical non-linear algorithm to find the optimal solution. (Note that the word "optimal" here implies near-optimal since the response surface approach itself introduces considerable approximations.)

5. Concluding Remarks

The chapter was meant to serve as an introduction to the technique of response surfaces in simulation-based optimization. A relatively new topic of combining response surfaces with neural networks was also introduced. The topic of neural networks will surface one more time in this book in the context of control optimization.

Bibliographic Remarks: The technique of response surfaces is now widely used in the industry. The method was developed around the end of the Second World War. For a comprehensive survey of RSM, the reader is referred to [213]. For RSM-based simulation optimization, the reader is referred to [281, 172, 19]. The so-called *kriging* methodology [149], which is based on interpolation rather than function fitting (e.g., least-squares minimization), has also been used in simulation metamodeling: see [174, 9]. Use of neural networks in RSM is a relatively recent development (see e.g., [165, 259, 220, 230, 231, 12, 87, 194, 202]). A more recent development in the field of neural networks include radial basis functions; see [210, 199].

The idea of neural networks, however, is not very new. Widrow and Hoff's work [323] on linear neural networks appeared in 1960. Research in non-linear neural networks was triggered by the pioneering work of Werbös [313]—a Ph.D. dissertation in the year 1974. See also [257] in 1986, which explained the methodology in details. Since then countless papers have been written on the methodology and uses of neural networks. The textbooks [132, 199] also contains excellent discussions. Our account in this chapter follows Law and Kelton [188] and Mitchell [205]. Neural networks remain, even today, a topic of ongoing research. We end with a simple exercise that the reader is urged to carry out.

Exercise: Evaluate the function $f(x)$ at the following 20 points.

$$x = 2 + 0.4i, \text{ where } i = 1, 2, \ldots, 20, \text{ where}$$

$$f(x) = 2x^2 + \frac{ln(x^3)}{x-1}, \text{ where } 1 \leq x \leq 10.$$

Now, using this data, train the batch version of backprop. Then, with the *trained* network, predict the function at the following points.

$$x = 3 + 0.3i \text{ where } i = 1, 2, \ldots, 10.$$

Test the difference between the actual value and the value predicted by the network. (Use codes from [121].)

Chapter 5

PARAMETRIC OPTIMIZATION: STOCHASTIC GRADIENTS AND ADAPTIVE SEARCH

1. Chapter Overview

This chapter focusses on simulation-based techniques for solving stochastic problems of *parametric* optimization, also popularly called *static* optimization problems. Such problems have been defined in Chap. 3.

At the very outset, we would like to state that our discussion will be limited to **model-free** techniques, i.e., techniques that do not require structural properties of the objective function. By structural properties, we mean the availability of the analytical closed form of the objective function, the availability of the distribution (or density) functions of random variables in the objective function, or the ability to manipulate the integrals and derivatives within the analytical form of the objective function. As stated previously, our interest in this book lies in complex stochastic optimization problems with large-scale solution spaces. For such problems, it is usually difficult to obtain the kind of structural properties typically needed by **model-based** techniques, such as likelihood ratios or score functions (which require the distributions of random variables within the objective function) and infinitesimal perturbation analysis. Model-based techniques have been studied widely in the literature; see [91, 253] and Chap. 15 of Spall [281] for an extensive coverage.

Model-free techniques are sometimes also called *black-box* techniques. Essentially, most model-free techniques are numeric, i.e., they rely on the objective function's *value* and not on its *closed form.* Usually, when these techniques are used, one assumes that it

DOI 10.1007/978-1-4899-7491-4_5

is possible to estimate the true value of the objective function at any given point in the solution space by averaging the objective function values obtained from *numerous* simulation runs at the same point. This approach is also called a *sample path* approach.

We will first discuss techniques for *continuous* parametric optimization. The main technique that we will cover is based on stochastic gradient descent. For the *discrete* case, we will cover meta-heuristics and a number of stochastic adaptive search techniques [333]. Convergence of *some* of these techniques is discussed in Chap. 10.

2. Continuous Optimization

The problem of continuous parametric optimization can be described formally as:

$$\text{Minimize } f(x(1), x(2), \ldots, x(N)),$$

where $x(i)$ denotes the ith decision variable, N denotes the number of decision variables, and $f(.)$ denotes the objective function. (Any maximization problem can be converted to a minimization problem by reversing the sign of the objective function $f(.)$, i.e., maximize $f(x) \equiv$ minimize $-f(x)$.)

As discussed in Chap. 3, we are interested in functions with *stochastic* elements, whose analytical expressions are unknown because it is difficult to obtain them. As a result, simulation may have to be used to find estimates of the function value. Now, we will present an approach that uses the gradient of the function for optimization.

2.1. Steepest Descent

The method of steepest descent is often used to solve parametric optimization (non-linear programming) problems. Outside of steepest descent, there are techniques, such as Newton's method and the conjugate gradients method, which use the gradient in the optimization process. In this book, we will, however, focus on classical steepest descent, which requires only the first derivative. Although it is not the most effective technique when the objective function's closed form is known, it requires only the first derivative and hence imposes a minimal computational burden in model-free optimization. The first derivative (gradient or slope) can often be *numerically* estimated even when the closed form is unknown.

The steepest-descent method operates according to the following simple rule when proceeding from one iteration to the next:

$$x(i) \leftarrow x(i) - \mu \frac{\partial f(\vec{x})}{\partial x(i)}, \text{ for } i = 1, 2, \ldots, N,$$

when the function $f(.)$ is to be minimized. In the above:

- $\frac{\partial f(\vec{x})}{\partial x(i)}$ denotes the partial derivative of $f(.)$ with respect to $x(i)$, and its value when evaluated at $\vec{x}$ is used in the rule above.
- μ, a positive scalar, is called the step size.

In case of maximization (also called hill climbing), the rule becomes:

$$x(i) \leftarrow x(i) + \mu \frac{\partial f(\vec{x})}{\partial x(i)} \text{ for } i = 1, 2, \ldots, N.$$

Classical steepest descent is due to Cauchy [59] from 1847. This algorithm forms one of the early applications of step sizes in optimization theory. Although simple, it has given birth to the field of "stochastic gradient descent" in modern times. We have already seen this rule in the context of neural networks.

In what follows, we present a step-by-step description of the steepest-descent method. We will assume for the time being that the analytical form of the function is known. The goal is to **minimize** the function. We will use the following notation frequently:

$$\left.\frac{\partial f(\vec{x})}{\partial x(i)}\right|_{\vec{x}=\vec{a}},$$

which denotes the scalar numeric value of the partial derivative when $\vec{x} = \vec{a}$.

Steps in steepest descent. Set m, the number of iterations in the algorithm, to 1. Let $\vec{x}^m$ denote the solution vector at the mth iteration. Initialize $\vec{x}^1$ to an arbitrary feasible solution (point).

Step 1. For $i = 1, 2, \ldots, N$, obtain the closed-form expression for the partial derivative:

$$\frac{\partial f(\vec{x})}{\partial x(i)}.$$

Step 2. For $i = 1, 2, \ldots, N$, update $x^m(i)$, using the following rule:

$$x^{m+1}(i) \leftarrow x^m(i) - \mu \left.\frac{\partial f(\vec{x})}{\partial x(i)}\right|_{\vec{x}=\vec{x}^m}. \tag{5.1}$$

Step 3. If all the partial derivatives equal zero or are sufficiently close to zero, STOP. Otherwise increment m by 1, and return to Step 2.

In practice, to ensure that the derivatives are sufficiently close to 0, one can test if the Euclidean norm of the partial derivative vector is less than ϵ, a pre-specified value.

When the partial derivative becomes zero, it is clear from Eq. (5.1) that the algorithm fails to update the relevant decision variable any further. Now, all the partial derivatives are zero in what is called a stationary point which could be a *local optimum*. This is why, it is often said that the steepest-descent method gets "trapped" in local optima. The *global optimum* is a point which is not only a local optimum but in addition the function there is at its optimal value. If there is *only one* local optimum for a function, clearly, it is also the global optimum as well. Otherwise, however, we have no way of knowing if the local optimum reached is also a global optimum, and unfortunately, steepest descent, because of its design via Eq. (5.1), cannot escape out of a local optimum.

The next example is provided as a simple reminder (from your first course in calculus) of how the derivative can be used directly in optimization.

A simple example. Let the function to be minimized be:

$$f(x, y) = 2x^2 + 4y^2 - x - y - 4.$$

We compute its partial derivatives and set them to zero:

$$\frac{\partial f(x, y)}{\partial x} = 4x - 1 = 0; \qquad \frac{\partial f(x, y)}{\partial y} = 8y - 1 = 0,$$

yielding $x = 1/4$ and $y = 1/8$.

Now that we know a local optimum for this problem, we will show how the same local optimum can be obtained using steepest descent. Let the starting point for the method be $(x = 2, y = 3)$. We will use 0.1 for the value for μ. The step size will not be changed. The expressions for the partial derivative are already known. The results obtained from using steepest descent are summarized below:

Iteration	x	y	function value
1	1.300000	0.700000	−0.660000
2	0.880000	0.240000	−3.340800
3	0.628000	0.148000	−3.899616
.	.	.	.
.	.	.	.
.	.	.	.
30	0.250000	0.125000	−4.187500

The above demonstrates that the steepest-descent method yields the same local optimum, i.e., $x = 1/4$ and $y = 1/8$.

Intuition suggests that increasing the step size should accelerate the rate of convergence of this procedure. But notice what happens with $\mu = 0.9$!

Iteration	x	y	function value
1	−4.3	−17.700000	1,308.14
2	12.08	110.64	49,129.97
3	80.22	4,248.32	72,201,483.88
.	.	.	.
.	.	.	.
.	.	.	.

As is clear, the method diverges, failing to reach the local optimum. This can be attributed to the step size being too large. It turns out that the step size has to be "sufficiently small" for this method to converge (reasons to be discussed in Chap. 10). Thus, arbitrarily large values for the step size may not work.

When the closed form is unknown, an upper limit for μ can be found with some experimentation. In practice, however, μ is often guessed to be a small positive fixed number such as 0.1 or a number that starts at a small positive value and gradually decays with every algorithm iteration to 0.

The steepest-descent method is guaranteed to reach the optimal solution only when (i) the step size is sufficiently small and (ii) the function satisfies certain conditions related to *convexity, continuity,* and *differentiability* of the function. These conditions will be discussed in Chap. 10 and can be found in any standard text on non-linear programming. Unfortunately, in the absence of the closed form, conditions such as these cannot usually be verified.

We now discuss an important issue regarding local and global optima. As defined above, the local optimum is a point where the partial derivatives are zero, while the global optimum is a local optimum at which the function is optimized. The word optimum is replaced by minimum in case of minimization and by maximum in case of maximization. A so-called non-convex (non-concave) function contains *multiple* local minima (maxima)—in any one of which the steepest-descent method can get trapped. Thus, with non-convex functions, the steepest descent method may yield a local minimum which may in fact be far away from the global optimum.

The steepest-descent method starts from an arbitrary point in the solution space. It then moves (i.e., changes the value of $\vec{x}$) from one point to another in its attempt to seek better points where the function has better values. When it reaches an "optimum" (local or not), it stops moving. See Fig. 5.1 for a pictorial representation of a function containing multiple local optima. As stated above, when a function has multiple optima, the steepest-descent algorithm can get stuck in any one optimum *depending on where it started.*

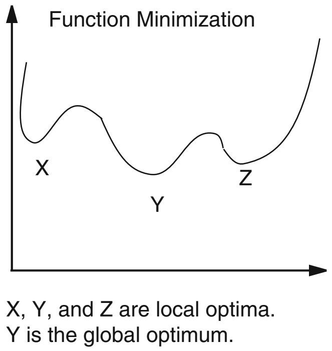

Figure 5.1. A surface with multiple minima

Multi-starts. One way to circumvent the difficulty of getting trapped in local optima is to run the algorithm a number of times, restarting the algorithm at a different point in the solution space each time [335]. The best local optimum obtained from all the starts is declared to be the solution. This approach is called the multi-start approach (or multiple starts) in optimization theory. While it does not guarantee the generation of the optimal solution, it is frequently the best we can do in practice to obtain the global optimum. The approach of multi-starts should be used in conjunction with any algorithm that has the property of getting stuck in the local optima. This property is also called *local convergence* as opposed to *global convergence*, which is the property of reaching the global optimum.

2.1.1 Simulation and Steepest Descent

In this book, we are interested in functions that have (a) stochastic elements and (b) unknown closed forms. With unknown closed forms (closed forms that cannot be expressed in analytical expressions), it is difficult to verify if conditions such as convexity, continuity, and differentiability are satisfied. In simulation optimization, these conditions can be rarely verified. In fact, we use simulation optimization when the closed form is not available. If it were available, we would use other techniques, which are likely to be far more effective and less time consuming. Hence this is an issue that we must live with. Furthermore, since the function form is unknown in simulation optimization, the derivative, if it is to be calculated, must be calculated *numerically*.

Now, a classical definition of the derivative is:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h},$$

which, of course, you know from your first course in calculus. This suggests the following formula for calculating the derivative numerically. Using a "small" value for h,

$$\frac{df(x)}{dx} \approx \frac{f(x+h) - f(x-h)}{2h}. \tag{5.2}$$

The above is called the **central** differences formula. Classically, the derivative can also be defined as:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h},$$

which suggests using the following with a "small" value for h:

$$\frac{df(x)}{dx} \approx \frac{f(x+h) - f(x)}{h}. \tag{5.3}$$

The above is called the **forward** differences formula. Formulas in both (5.2) and (5.3) yield *approximations* of the actual value of the derivative. The true value is obtained in the limit with h tending to zero.

Note that, in the context of simulation, $f(x+h)$, $f(x)$, and $f(x-h)$ will have to be estimated by simulation. Simulation-based estimates themselves have errors, and therefore this approach is approximate; one has to live with this error in simulation-based optimization.

We now state two important facts about derivative evaluations for simulation optimization. (1) There is empirical evidence to suggest (see [8] and references therein) that Eq. (5.2) (central differences) has statistical properties superior to those of Eq. (5.3); i.e., the error produced due to the approximation of h by a positive quantity is less with the central differences formula. (We will prove this in Chap. 10.) (2) The function evaluations at $(x+h)$ and $(x-h)$ must be performed using common random numbers. This means that both function evaluations should use the *same* set of random numbers in the replications. For instance, if a set of random numbers is used in replication 3 of $f(x+h)$, then the same set should be used in replication 3 of $f(x-h)$. Using common random numbers has been proven to be a "good" strategy—through the viewpoint of statistics. See [8] for additional details.

A simple example is now used to illustrate the numerical computing of a derivative. The derivative of the function

$$f(x) = 2x^3 - 1 \text{ with respect to } x \text{ is } 6x^2.$$

Therefore the actual value of the derivative when $x = 1$ is 6. Now from Eq. (5.2), using $h = 0.1$, the derivative is found to be:

$$\frac{[2(6+0.1)^3 - 1] - [2(6-0.1)^3 - 1]}{(2)(0.1)} = 6.02,$$

and using $h = 0.01$, the derivative is found to be:

$$\frac{[2(6+0.01)^3 - 1] - [2(6-0.01)^3 - 1]}{(2)(0.01)} = 6.0002.$$

As h becomes smaller, we approach the value of the derivative. The above demonstrates that the value of the derivative *can* be approximated with small values for h. When the analytic function is unavailable, as is the case with objective functions of complex stochastic systems, one may use numerical approximations such as these for computing derivatives.

The so-called "finite difference" formula for estimating the derivative is the formula in Eq. (5.2) or Eq. (5.3). In problems with *many* decision variables, the finite difference method runs into trouble since its computational burden becomes overwhelming. Here is why. Consider the case with N decision variables:

$$x(1), x(2), \ldots, x(N).$$

In each iteration of the steepest-descent algorithm, one then has to calculate N partial derivatives of the function. Note that the general expression using central differences is:

$$\frac{\partial f(\vec{x})}{\partial x(i)} = \frac{f(x(1),x(2),\ldots,x(i)+h,\ldots,x(N)) - f(x(1),x(2),\ldots,x(i)-h,\ldots,x(N))}{2h}.$$

The ith partial derivative requires two function evaluations:

$$\text{One evaluation is at } (x(1),x(2),\ldots,x(i)+h,\ldots,x(N))$$

$$\text{and the other is at } (x(1),x(2),\ldots,x(i)-h,\ldots,x(N)).$$

This implies that in each iteration of the steepest-descent algorithm, one would require $2k$ function evaluations, i.e., 2 evaluations per decision variable. Since each function evaluation is via simulation, each function evaluation in turn needs several replications. The computational time taken for just one replication can be significant. Clearly, as N increases, the number of simulations needed increases, and consequently just one iteration of steepest descent may take a significant amount of time. This is a major stumbling block. Is there a way out?

The answer is yes, sometimes. A major breakthrough was provided by the work of Spall [280] via what is known as the *simultaneous* perturbation method. Spall showed that regardless of the number of decision variables, an approximate but useful estimate of the derivative could be estimated via only *two* function evaluations. Now, we note that one cannot tamper with the definition of the derivative. The significance of Spall's work hence lies in the fact that although his derivative's estimate strays from that of the classical definition, his estimate *can* be used in a steepest-descent method to find the local optimum! In other words, the derivative itself is inaccurate, but the resulting steepest-descent method, which is really of interest to us, converges. We will discuss this method now.

2.1.2 Simultaneous Perturbation

The word "perturbation" is related to the fact that the function is evaluated at $(x-h)$ and $(x+h)$. In other words, the function is moved slightly (perturbed) from x, which is the point at which the derivative is desired. This, of course, is the central idea underlying numerical evaluation of a derivative, and this stems from the classical definition of a derivative.

In simulation optimization, this idea can be found possibly for the first time in Kiefer and Wolfowitz [164]. The idea is elementary in the sense that it stems from foundational ideas in calculus (from Newton (1642–1727) and Leibniz (1646–1716)). Much of the early work in gradient-based simulation optimization is essentially an application of the fundamental definition of a derivative from the seventeenth century. It is Spall's work [280] that actually, for the first time, breaks away from this idea and paves the way for an efficient method for numerical non-linear optimization.

When used in a steepest-descent algorithm, Spall's definition for a derivative provides us with a mechanism that is not only very efficient in terms of function evaluations needed, but also one that has been shown to converge. As such, it is not surprising that the method is extremely attractive for problems in which the objective function's analytical form is unknown, function estimation can be done via simulation, and the number of decision variables is large. Examples of such problems, as mentioned previously, are optimization problems related to complex stochastic systems. Simulation optimization is clearly a fertile ground for application.

In the finite difference method, when calculating a partial derivative with respect to a variable $x(i)$, it is $x(i)$ that is perturbed—in other words, we evaluate the function at $x(i) + h$ and $x(i) - h$, keeping the other variables unchanged. To illustrate this idea, we present the case with two variables: $x(1)$ and $x(2)$. Then:

$$\frac{\partial f(\vec{x})}{\partial x(1)} = \frac{f(x(1)+h, x(2)) - f(x(1)-h, x(2))}{2h} \text{ and}$$

$$\frac{\partial f(\vec{x})}{\partial x(2)} = \frac{f(x(1), x(2)+h) - f(x(1), x(2)-h)}{2h}.$$

The above should make it clear that each variable is perturbed (with h) separately, and as a result, one needs to evaluate the function 4 times. The four evaluations in the two preceding expressions are:
(i) $f(x(1)+h, x(2))$, (ii) $f(x(1)-h, x(2))$, (iii) $f(x(1), x(2)+h)$, and (iv) $f(x(1), x(2)-h)$.

In the simultaneous perturbation method, we perturb all variables *simultaneously*. So in the above 2-variable example, using simultaneous perturbation, we would need to evaluate the function at only two points. The function evaluations needed would be

$$\text{(i) } f(x(1)+h(1), x(2)+h(2)), \text{ and (ii) } f(x(1)-h(1), x(2)-h(2)).$$

These two evaluations would then be used to find the two partial derivatives. It is perhaps clear now that regardless of the number of variables, we will only need two evaluations. The formula for this estimate of the derivative is provided formally in Step 3 of the algorithm description that follows.

Steps in simultaneous perturbation. Set m, the iteration number, to 1. Let N denote the number of decision variables. Initialize $\vec{x}^1$ to an arbitrary feasible point. We will terminate the algorithm when the step size μ becomes smaller than μ_{min}, a user-specified value. See Remark 1 below for rules for selecting μ.

Step 1. Assume that $H(i)$ for every $i = 1, 2, \ldots, N$ is a Bernoulli-distributed random variable, whose two permissible, *equally likely*, values are 1 and -1. Using this distribution, assign values to $H^m(i)$ for $i = 1, 2, \ldots, N$. Then compute $h^m(i)$ for every value of i using

$$h^m(i) = H^m(i)c^m, \tag{5.4}$$

where c^m is the so-called perturbation coefficient. An example is: $c^m = C/m^t$ where $C = 0.1$ and $t = 1/6$. See Remark 1 below for more on this.

Step 2. Calculate F^+ and F^- using the following formulas:

$$F^+ = f(x^m(1) + h^m(1), x^m(2) + h^m(2), \ldots, x^m(N) + h^m(N));$$

$$F^- = f(x^m(1) - h^m(1), x^m(2) - h^m(2), \ldots, x^m(N) - h^m(N)).$$

Step 3. For $i = 1, 2, \ldots, N$, obtain the value for the partial derivative using

$$\left.\frac{\partial f(\vec{x})}{\partial x(i)}\right|_{\vec{x}=\vec{x}^m} \approx \frac{F^+ - F^-}{2h^m(i)}.$$

Step 4. For each i, update $x^m(i)$ using the following rule.

$$x^{m+1}(i) \leftarrow x^m(i) - \mu \left.\frac{\partial f(\vec{x})}{\partial x(i)}\right|_{\vec{x}=\vec{x}^m}.$$

Notice that, in the above, one needs the values of the derivatives, which were obtained in Step 3.

Step 5. Increment m by 1 and update μ^m using some step size decaying rule (discussed below). If $\mu^m < \mu_{\text{min}}$ then STOP; otherwise, return to Step 1.

Remark 1. Rules for μ^m and c^m: Spall [281] provides the following guidelines for updating the step size and the perturbation coefficient:

$$\mu^m = \frac{A}{(B+m)^l} \text{ and } c^m = \frac{C}{m^t}.$$

Theoretically valid values in the above rules are: $l = 0.602$ and $t = 0.101$. However, the reader is warned that these rules only form guidelines and may not be reliable for every application [281]. Some values that this author has used in his experiments with the rules above are: $l = 1$, $A = 10$, $B = 100$, $C = 0.1$ and $t = 0.5$. The values for both μ^m and c^m should be small in general. If μ^m is too small, however, updating will be discouragingly slow. Trial and error is used in practice to identify the best possible updating rule.

Remark 2. The formula used above for estimating the derivative is evidently different from the classical finite difference formula. The difference should become clearer from the discussion below on the finite difference approach.

Remark 3. The algorithm, as mentioned above, is not guaranteed to find the global optimum. Hence it is best to run the algorithm a few times starting at a different point in each run, i.e., with multiple starts.

Remark 4. Using common random numbers to estimate F^+ and F^- is recommended to reduce the effect of simulation noise.

Remark 5. If the solution space is **constrained**, then one can convert the problem into one of unconstrained minimization by using a so-called *penalty* function. An alternative is to "project" the solution onto the feasible region. The latter implies that when the algorithm suggests a solution outside the feasible region, the solution is adjusted so that it lies just inside the boundary.

A finite difference version. If one were to use a finite difference estimate instead of the simultaneous perturbation estimate, some steps in the description given above would change. In Step 1, $h(i)$ for $i = 1, 2, \ldots, N$ would be assigned to a constant small value. The other changes would be:

Step 2. Calculate $F^+(i)$ and $F^-(i)$ for each $i = 1, 2, \ldots, N$ via:

$$F^+(i)=f(x^m(1)+h(1)I(i=1), x^m(2)+h(2)I(i=2), \ldots, x^m(N)+h(N)I(i=N));$$
$$F^-(i)=f(x^m(1)-h(1)I(i=1), x^m(2)-h(2)I(i=2), \ldots, x^m(N)-h(N)I(i=N)).$$

Here $I(.)$ is an identity function that equals 1 when the condition inside the round brackets is satisfied and equals 0 otherwise.

Step 3. For each $i = 1, 2, \ldots, N$, obtain the value for the partial derivative using:

$$\left.\frac{\partial f(\vec{x})}{\partial x(i)}\right|_{\vec{x}=\vec{x}^m} \approx \frac{F^+(i) - F^-(i)}{2h(i)}.$$

The true values of the derivatives (finite differences and simultaneous perturbation) can be quite different from the estimates produced by these techniques; in other words, there is an error or noise (or bias) in these estimates. As mentioned earlier, we will discuss these issues in Chap. 10 where we present a convergence analysis. Codes for simultaneous perturbation are provided at [121].

Since the finite difference and simultaneous perturbation methods contain noise in their derivatives (gradients), they belong to the family called *stochastic gradient* methods. We have presented only two model-free methods in this family. We will conclude our discussion on continuous optimization with a model-free method that does not require derivatives. An important fact that we would like to underscore here is that both finite difference and simultaneous perturbation are locally convergent. While this does not ensure reaching the global optimum, it is still a rather attractive feature in the world of simulation optimization because it can be combined with multiple starts in practice.

2.2. Non-derivative Methods

The method we discuss here is known as the Nelder-Mead algorithm [216]. It goes by other names such as *downhill simplex* and *flexible polygon search*. This method is **immensely popular** because it has been widely used in the real world with considerable success. Because it does not have satisfactory convergence properties (not even local convergence), it is regarded as a heuristic; see however [186].

It is a natural candidate for simulation optimization because it only needs numeric values of the function [20]. It needs to perform at least $(N+1)$ evaluations of the function per iteration. The philosophy underlying this method is quite simple. One starts with a set of feasible solutions; the set is referred to as a "simplex" or a "polygon." In every iteration, a poor solution in the simplex is dropped in favor of a superior solution. (This method is not to be confused with the simplex method, which is used for solving linear programs.)

We now provide the steps in the algorithm. The algorithm is written in terms of *minimizing* the objective function value. In the description below and in some other algorithms in this chapter, we will use the following notation frequently:

$$\vec{a} \leftarrow \vec{b}.$$

The above implies that if $\vec{a}$ and $\vec{b}$ are N-dimensional,

$$a(j) \leftarrow b(j) \text{ for } j = 1, 2 \ldots, N.$$

Steps in the Nelder-Mead method. Let N denote the number of decision variables. Arbitrarily select $(N+1)$ solutions in the feasible solution space. We denote the ith solution by $\vec{x}(i)$ and the set of these solutions by $\mathcal{P}$. These solutions together form a so-called *polygon* or a *simplex.*

Step 1. From the set $\mathcal{P}$, select the following three solutions: the solution with the maximum objective function value, to be denoted by $\vec{x}_{\max}$, the solution with the second largest objective function value, to be denoted by $\vec{x}_{sl}$, and the solution with the lowest objective function value, to be denoted by $\vec{x}_{\min}$. Now compute the so-called *centroid* as follows:

$$\vec{x}_c \leftarrow \frac{1}{N}\left[-\vec{x}_{\max} + \sum_{i=1}^{N+1} \vec{x}(i)\right].$$

(The above is a centroid of all the points except for $\vec{x}_{\max}$.) Then compute the so-called *reflected* point as follows:

$$\vec{x}_r \leftarrow 2\vec{x}_c - \vec{x}_{\max}.$$

Step 2. Depending on the value of $f(\vec{x}_r)$, we have three choices:

- If $f(\vec{x}_{\min}) > f(\vec{x}_r)$, go to Step 3.
- If $f(\vec{x}_{sl}) > f(\vec{x}_r) \geq f(\vec{x}_{\min})$, go to Step 4.
- If $f(\vec{x}_r) \geq f(\vec{x}_{sl})$, (i.e., either $f(\vec{x}_r) \geq f(\vec{x}_{\max})$ or $f(\vec{x}_{\max}) > f(\vec{x}_r) \geq f(\vec{x}_{sl})$), go to Step 5.

Step 3. We come here if the reflected point is better than the best point in $\mathcal{P}$. The operation performed here is called **expansion**. The idea is to determine if a point even better than the reflected point can be obtained. Compute the expanded solution as follows:

$$\vec{x}_{exp} \leftarrow 2\vec{x}_r - \vec{x}_c.$$

If $f(\vec{x}_{exp}) < f(\vec{x}_r)$, set $\vec{x}_{new} \leftarrow \vec{x}_{exp}$. Otherwise, set $\vec{x}_{new} \leftarrow \vec{x}_r$. Go to Step 6.

Step 4. We come here if the reflected point is better than $\vec{x}_{sl}$. Set $\vec{x}_{new} \leftarrow \vec{x}_r$, and go to Step 6.

Step 5. We come here if the reflected point is worse than $\vec{x}_{sl}$. The operation performed here is called **contraction.**

- If $f(\vec{x}_{\max}) > f(\vec{x}_r) \geq f(\vec{x}_{sl})$, set $\vec{x}_{new} \leftarrow 0.5(\vec{x}_r + \vec{x}_c)$, and go to Step 6.

- Otherwise, compute: $\vec{x}_{new} \leftarrow 0.5(\vec{x}_{\max} + \vec{x}_c)$, and go to Step 6.

Step 6. Remove the old $\vec{x}_{\max}$ from the polygon, i.e., set $\vec{x}_{\max} \leftarrow \vec{x}_{new}$ and return to Step 1.

The algorithm can be run for a user-specified number of iterations. See [239] for a description that differs from the above. The algorithm works well only on problems with up to 10 decision variables [29]. Another non-derivative method, which we do not discuss, is that of Hooke and Jeeves [143].

3. Discrete Optimization

Discrete parametric optimization is actually harder than continuous parametric optimization since the function may have gaps, and hence derivatives may be of little use. Even when the function can be evaluated exactly, discrete parametric optimization leads to a difficult problem, unless the problem has a special structure. Without the closed form, structure is hard to find, and the structure is hence not available in the model-free context.

We will make the following important assumption regarding discrete parametric optimization problems. We will assume that the solution space is finite (although possibly quite large). Like in the continuous case, we assume that it is possible to estimate the function at any given point using simulation, although the estimate may not be exact, i.e., it may contain some noise/error.

Now, if the solution space is manageably small, say composed of 100 points, then the problem can often be solved by an exhaustive search of the solution space. An exhaustive search should be conducted only if it can be performed in a reasonable amount of time. Generally, in an exhaustive search, one evaluates the function with a pre-determined number of replications (samples) at all the points in the solution space. What constitutes a manageably small space may depend on how complex the system is. For an M/M/1 queuing simulation written in C (see [188] for a computer program), testing the function even at 500 points may not take too much time, since M/M/1 is a simple stochastic system defined by just two random variables. However, if the simulation is more complex, the time taken to evaluate the function at even one point can be significant, and hence the size of a "manageable" space may be smaller. With the increasing power of computers, this size is likely to increase.

If the solution space is large, i.e., several thousand or more points, it becomes necessary to use algorithms that can find **good** solutions without having to search exhaustively. Under these circumstances, one can turn to the so-called *meta-heuristic* and *stochastic adaptive search* techniques. These techniques have emerged in the literature to solve combinatorial optimization problems of the discrete nature where the objective function's value can be estimated but no structure is available for the form of the objective function.

In general, meta-heuristics work very well in practice but their convergence properties are unknown, while stochastic adaptive search techniques tend to have well-understood convergence properties, usually global convergence properties, in addition to being practically useful. We will cover these methods in Sects. 3.2 (meta-heuristic) and 3.3 (stochastic adaptive search).

When we have a search space of a manageable size, we have the luxury of using a variable number of replications (samples) in function evaluation during the search for the optimal solution. This can be done via the **ranking** and **selection** methods [106] or the multiple comparison procedure [139]. These methods are likely to be more efficient than an exhaustive search with a pre-determined number of replications.

Ranking and selection methods have strong mathematical backing and serve as robust methods for comparison purposes. As we will see later, they can also be used in combination with other methods. We begin this section with a discussion on ranking and selection.

3.1. Ranking and Selection

Ranking and selection methods are statistical methods designed to select the best solution from among a set of competing candidates. They have a great deal of theoretical (statistical theory) and empirical backing and can be used when one has up to 20 (candidate) solutions. In recent times, these methods have been used effectively on larger (solution space size $\gg$20) problems.

A useful feature of ranking and selection is that if certain conditions are met, one can *guarantee* that the probability of selecting the best solution from the candidate set exceeds a user-specified value. These methods are also useful in a careful statistical comparison of a finite number of solutions.

We will discuss two types of ranking and selection methods, namely, the Rinott method and the Kim-Nelson method. The problem considered here is one of finding the best solution from a set of candidate solutions. We may also want to rank the solutions. We will discuss the comparison problem as one in which the solution with the *greatest* value for the objective function is declared to be the best. We begin with some notation.

r: the total number of solutions to be evaluated and compared.

$X(i, j)$: the jth independent observation from the ith solution. This needs some explanation. Recall from Chap. 2 that to estimate steady-state (or long-run) performance measures, it becomes necessary to obtain several independent observations of the performance measure under consideration. (Usually, these observations are obtained from the independent replications of the system.) In our context, the performance measure is the objective function. If we have r solutions to compare, then clearly, i takes values from the set $\{1, 2, \ldots, r\}$. Similarly, j takes values from the set $\{1, 2, \ldots, m\}$, where m denotes the total number of independent observations.

$\bar{X}(i, m)$: the sample mean obtained from averaging the first m samples from the ith solution. Mathematically:

$$\bar{X}(i, m) = \frac{\sum_{j=1}^{m} X(i, j)}{m}.$$

δ: the so-called "indifference zone" parameter. If the absolute value of the difference in the objective function values of two solutions is less than δ, we will treat the two solutions to be equally good (or poor), i.e., we will not distinguish between those two solutions. Clearly, δ will have to be set by the user.

α: the significance level in the comparison. In defining δ above, we can state that a ranking and selection method will guarantee with a probability of $(1 - \alpha)$ that the solution selected by it as the best *does* have the largest mean, if the *true* mean of the best solution is at least δ better than the second best.

We will assume throughout the discussion on ranking and selection methods that the values of $X(i, j)$ for any given i are normally distributed and that their mean and variance are unknown.

3.1.1 Steps in the Rinott Method

After reviewing the notation defined above, select suitable values for α, δ, and the sampling size, m, where $m \geq 2$. For each $i = 1, 2, \ldots, r$, simulate the system associated with the ith solution. Obtain m independent observations of the objective function value for every system. $X(i, j)$, as defined above, denotes the jth observation (objective function value) of the ith solution.

Step 1. Find the value of Rinott's constant h_{R} using the tables in [325] or from the computer program in [22]. This value depends on m, r,and α. For each $i = 1, 2, \ldots, r$, compute the sample mean using:

$$\bar{X}(i,m) = \frac{\sum_{j=1}^{m} X(i,j)}{m},$$

and the sample variance using:

$$S^2(i) = \frac{1}{m-1}\sum_{j=1}^{m}\left[X(i,j) - \bar{X}(i,m)\right]^2.$$

Step 2. Compute, for each $i = 1, 2, \ldots, r$,

$$\mathsf{N}_i = \max\left(m, \left[\frac{h_{\mathrm{R}}^2 S^2(i)}{\delta^2}\right]^+\right),$$

where $[a]^+$ denotes the *smallest* integer greater than a. If $m \geq \max_i \mathsf{N}_i$, declare the solution with the maximum $\bar{X}(i,m)$ as the best solution. STOP.

Otherwise, obtain $\max(0, \mathsf{N}_i - m)$ *additional* independent observations of the objective function value for the ith solution, for $i = 1, 2, \ldots, r$. Then declare the solution(s) with the maximum value for $\bar{X}(i, \mathsf{N}_i)$ as the best.

We now discuss the Kim-Nelson method [166], which may require fewer observations in comparison to the Rinott method [246].

3.1.2 Steps in the Kim-Nelson Method

After reviewing the notation defined above, select suitable values for α, δ, and the sampling size, m, where $m \geq 2$. For each $i = 1, 2, \ldots, r$, simulate the system associated with the ith solution. Obtain m independent observations of the objective function value for every system. $X(i,j)$, as defined above, denotes the jth observation (objective function value) of the ith solution.

Step 1. Find the value of the Kim-Nelson constant using:

$$h_{\mathrm{KN}}^2 = \left[[2\{1-(1-\alpha)^{1/(r-1)}\}]^{-2/(m-1)} - 1\right][m-1].$$

Step 2. Let $\mathcal{I} = \{1, 2, \ldots, r\}$ denote the set of candidate solutions. For each $i = 1, 2, \ldots, r$ compute the sample mean as:

$$\bar{X}(i,m) = \frac{\sum_{j=1}^{m} X(i,j)}{m}.$$

For all $i \neq l, i \in \mathcal{I}, l \in \mathcal{I}$, compute:

$$S^2(i,l) = \frac{1}{m-1}\sum_{j=1}^{m}[X(i,j) - X(i,l) + \bar{X}(l,m) - \bar{X}(i,m)]^2.$$

Step 3. Compute:

$$\breve{N}_{il} = \left(\frac{h_{\text{KN}}^2 S^2(i,l)}{\delta^2}\right)^-,$$

where $(a)^-$ denotes the largest integer smaller than a. Let $\mathsf{N}_i = \max_{i \neq l} \breve{N}_{il}$.

If $m \geq (1 + \max_i \mathsf{N}_i)$, declare the solution with the maximum value for $\bar{X}(i,m)$ as the best solution, and STOP.

Otherwise, set $p \leftarrow m$, and go to the next step.

Step 4.

Let $\mathcal{I}_s = \{i : i \in \mathcal{I} \text{ and } \bar{X}(i,p) \geq \bar{X}(l,p) - W_{il}(p) \quad \forall l \in \mathcal{I}, l \neq i\}$, where

$$W_{il}(p) = \max\left(0, \frac{\delta}{2p}\left[\frac{h_{\text{KN}}^2 S(i,l)}{\delta^2} - p\right]\right).$$

Then set: $\mathcal{I} \leftarrow \mathcal{I}_s$.

Step 5. If $|\mathcal{I}| = 1$, declare the solution whose index is still in $\mathcal{I}$ as the best solution, and STOP. Otherwise, go to Step 6.

Step 6. Take one *additional* observation for each system in $\mathcal{I}$ and set $p \leftarrow p + 1$. If $p = 1 + \max_i \mathsf{N}_i$, declare the solution whose index is in $\mathcal{I}$ and has the maximum value for $\bar{X}(i,p)$ as the best solution, and STOP. Otherwise, go to Step 4.

3.2. Meta-heuristics

When we have several hundred or several thousand solutions in the solution space, neither ranking and selection methods nor exhaustive enumeration can be used directly. We may then resort to using *meta-heuristics.* Since it becomes difficult to use a variable number of replications, as needed in ranking and selection, with meta-heuristics, one usually uses a large, but fixed, pre-determined number of replications (samples) in evaluating the function at any point in the solution space. As stated above, meta-heuristics do not have satisfactory convergence properties, but often work well in practice on large-scale discrete-optimization problems. In this subsection, we will cover two meta-heuristic techniques: the *genetic algorithm* and *tabu search*. The genetic algorithm appears to be the oldest meta-heuristic in the literature.

Simulated annealing, another meta-heuristic, has now been analyzed extensively for its convergence properties. Because of this, it is nowadays said to belong to a class of techniques called stochastic adaptive search (SAS). Hence, we will discuss it in the next subsection with other SAS techniques, focusing on the genetic algorithm and tabu search in this subsection. We do note that the line between meta-heuristics and SAS is rather thin because if shown to be convergent a meta-heuristic may be called an SAS technique, and we use this demarcation only to help us in organizing our discussion.

Meta-heuristics rely on numeric function evaluations and as such can be combined with simulation. On large-scale problems, it is difficult to obtain the optimal solution. Therefore, usually, the meta-heuristic's performance (on a large problem) cannot be calibrated with reference to that of an optimal solution. The calibration has to be done with other available heuristic methods or, as we will see later, with pure random search. Also, one must remember that meta-heuristics are **not** guaranteed to produce optimal solutions, but only good solutions; further they produce good solutions in a reasonable amount of time on the computer. In particular, we note that most meta-heuristics do not even guarantee local convergence, as opposed to global convergence guaranteed by many SAS techniques.

Before plunging into the details, we discuss what is meant by a "neighbor" of a solution. We will explain this with an example.

Example. Consider a parametric optimization problem with two decision variables, both of which can assume values from the set:

$$\{1, 2, \ldots, 10\}.$$

Now consider a solution $(3, 7)$. A *neighbor* of this solution is $(4, 6)$, which is obtained by making the following changes in the solution $(3, 7)$.

$$3 \longrightarrow 4 \text{ and } 7 \longrightarrow 6.$$

It is not difficult to see that these changes produced a solution—$(4, 6)$—that lies in the "neighborhood" of a given solution $(3, 7)$. Neighbors can also be produced by more complex changes.

Clearly, the effectiveness of the meta-heuristic algorithm will depend on the effectiveness of the neighbor generation strategy. Almost all the algorithms that we will discuss in the remainder of this chapter will require a neighbor generation strategy. The idea behind the so-called *hit-and-run* strategy originates from [279]. An in-depth discussion of this strategy and its variants which select improving neighbors can be found in Chap. 6 of [333]. We now discuss this strategy in some detail.

Hit-and-run strategy. The underlying idea is similar to that used in simultaneous perturbation. Let $\vec{x} = (x(1), x(2), \ldots, x(N))$ be the current solution with N decision variables. Assume that every $H(i)$, for $i = 1, 2, \ldots, N$, is itself a Bernoulli-distributed random variable, whose two permissible, *equally likely*, values are 1 and -1. Using this distribution, assign values to $H(i)$ for $i = 1, 2, \ldots, N$. Let $c(i)$ denote the step size for the ith decision variable. Then, a new solution, $\vec{y}$, i.e., a neighbor, is generated as follows: For $i = 1, 2, \ldots, N$,

$$y(i) \leftarrow x(i) + H(i)c(i),$$

where the value of $c(i)$ is the least increment permitted for the ith decision variable. Thus, for instance, if the ith decision variable assumes values from an equally spaced set, $\{2, 4, 6, 8, \ldots, 20\}$, then $c(i)$ is clearly 2. Obviously, this definition of $c(.)$ is appropriate for decision variables that have equally spaced values. For variables that do not take values from equally spaced sets, one must select $c(.)$ in a way such that $y(.)$ becomes a feasible solution for every $c(.)$ selected. We also note that $c(.)$ does not have to be the least increment permitted. For variables assuming values from equally spaced sets, $c(.)$ can be any integer multiple of the least increment. We now illustrate the hit-and-run strategy with an example.

Example. Consider a parametric optimization problem with two decision variables, both of which can assume values from the set:

$$\{1, 2, \ldots, 10\}.$$

Now consider a solution $\vec{x} = (1, 7)$. We use the Bernoulli distribution to generate random values for $H(.)$. Assume that our random number generator leads us to: $H(1) = -1$ and $H(2) = 1$. Let $c(i) = 1$ for $i = 1, 2$. Then, a new solution should be:

$$\vec{y} = (1 - 1, 7 + 1) = (0, 8).$$

The above solution is not feasible, since it does not belong to the solution space. Hence, we perform one more attempt to generate a neighbor. Let us assume that on this occasion, the random number generator produces the following values: $H(1) = 1$ and $H(2) = 1$.

Then, the new solution should be:

$$\vec{y} = (1+1, 7+1) = (2, 8),$$

which is a feasible solution. Thus, it is clear that this hit-and-run-based strategy may not yield a feasible neighbor in every attempt, and when that happens, one should keep attempting to generate neighbors until a feasible neighbor is generated. If one gets stuck in a point for which no feasible neighbor exists, one should restart the search at a randomly selected different point.

3.2.1 The Genetic Algorithm

The genetic algorithm is a *very* popular meta-heuristic inspired by evolutionary phenomena that favor reproduction of individuals with certain traits. The algorithm has been applied extensively in the industry with a good deal of success. For expositionary purposes, we will first present a *highly simplified* version of the algorithm, and then discuss some more refined variants.

Steps in the genetic algorithm. Let m denote the iteration number in the algorithm. Let $m_{\max}$ denote the maximum number of iterations to be performed. This number has to be pre-specified, and there is no rule to find an optimal value for it. Usually, $m_{\max}$ is dictated by the permissible amount of computer time.

Step 1. Set $m = 1$. Select r initial solutions, where $r > 1$. The value of r is specified by the user. Ideally, all the r solutions should be relatively "good," although this is not a requirement.

Step 2. Identify the best and the worst among the r solutions (for minimization, the best solution is the one with the minimum objective function value and the worst is the one with the maximum objective function value). Denote the best by $\vec{x}_{best}$ and the worst by $\vec{x}_{worst}$. Randomly select a neighbor of $\vec{x}_{best}$, and call it $\vec{x}_{new}$. Now, replace the worst solution by the new solution. In other words: $\vec{x}_{worst} \leftarrow \vec{x}_{new}$. Do not change any other solution and go to Step 3. Note that the solution set of r solutions has now changed.

Step 3. Increment m by 1. If $m = m_{\max}$, return $\vec{x}_{best}$ as the best solution and STOP. Otherwise, go back to Step 2.

The value of r depends on the size of the problem. Intuition suggests that a large value for r may lead to better performance. A *practical* modification of Step 2 is to check if the new solution is better than the worst. If it is, the new solution should replace the worst; if not, one should return to Step 2 and generate a new neighbor of the best solution.

The format described above for the genetic algorithm is one *out of a very large number of variants* proposed in the literature. Shortly, we will discuss some refined variants of this format.

We now discuss why this algorithm is called a "genetic" algorithm. Survival of the fittest is a widely believed theory in evolution. It is believed that the reproduction process favors the fittest individual; in other words, the fittest individual reproduces more. As such, the fittest individuals get more opportunities—via mating—to pass their genes to the next generation. And an accepted belief is that this, in the next generation, produces individuals who are especially capable of reproduction. In other words those genes are passed that can produce healthy individuals capable of reproduction. Because the algorithm uses a number of solutions within each iteration, it is also called a *population-based* algorithm.

In each generation (iteration) of the algorithm, one has a set of individuals (solutions). The algorithm allows only the fit individuals to reproduce. Fitness, in our context, is judged by how good the objective function is. The algorithm also assumes that a good solution (read a potential *parent* with *strong reproductive features*) is likely to produce a good or a better neighbor (read a *child* that has good or even a better capability of reproducing).

In Step 2, the algorithm selects the best solution (the individual fittest to reproduce), selects its neighbor (allows it to reproduce and produce a child), and then replaces the worst solution (the individual least fit for reproduction dies) by the selected neighbor (child of a fit individual). In the process, in the next iteration (generation), the individuals generated are superior in their objective function values. This continues with every iteration producing a better solution.

Example. We next show the steps in the genetic algorithm via a simple example with two decision variables which assume values from the set $\{1, 2, \ldots, 10\}$. We wish to minimize the function.

Step 1. Set $m = 1$. Let us set r, the population size, to 4. Let us select the 4 solutions to be: $(2, 4)$, $(1, 5)$, $(4, 10)$, and $(3, 2)$, and assume the respective function values to be: 34, 12, 45, and 36.

Step 2. Clearly: $\vec{x}_{best} = (1,5)$ and $\vec{x}_{worst} = (4,10)$. Let the randomly selected neighbor ($\vec{x}_{new}$) of the best solution be $(2,6)$. Replacing the worst solution by the new solution, our new population becomes:

$$(2,4),(1,5),(2,6),\text{ and }(3,2).$$

Then we go back to Step 2, and then perform additional iterations.

Refined variants. The texts [227, 281] discuss a number of variants of the algorithm described above. We discuss a few below.

Ranking: In one variant, in Step 2, the worst and the second worst solutions are replaced by neighbors of the best and the second best solutions respectively. In general, one can rank all the r solutions (possibly using the ranking and selection procedures discussed above). Then, in Step 2, one can replace the kth worst solution by a neighbor of the kth best solution for $\mathsf{k} = 1,2,\ldots,\mathsf{n}$, where

$$\text{the maximum value of } \mathsf{n} = \begin{cases} (\frac{r}{2}+1)^{-} & \text{if } r \text{ is even} \\ (\frac{r}{2})^{-} & \text{if } r \text{ is odd} \end{cases}$$

where $(l)^{-}$ is the largest integer smaller than l. Thus, if $r=6$, $(r/2+1)^{-}=3$, while if $r=5$, $(r/2)^{-}=2$.

Cross-over and mutation: A more refined version of the genetic algorithm uses the notions of *cross-over* and *mutation* to produce neighbors. In a "cross-over," $\vec{x}_{new}$ is generated by combining the best and the second best solutions; the underlying motive is that the two "parents" are the best and the second best while the new solution is the "child" that inherits the best traits. The combining mechanism works as follows. Each solution is assumed to be composed of "genes." So in (a,b), a is the first gene and b the second. An example of a cross-over-generated child, whose parents are $(2,4)$ and $(1,5)$, is $(1,4)$. In this child, the first gene, i.e., 1, comes from the second parent, while the second, i.e., 4, comes from the first. In a so-called mutation, some solution (often the best) is "mutated" by swapping "genes." An example of a mutant of $(1,5)$ is $(5,1)$. In one variant of the genetic algorithm [133], several mutant and cross-over-generated children are produced, and much of the older generation (ninety percent) is replaced by the progeny.

Coding and fitness functions: In yet another strategy, the actual decision space is converted into a string of binary numbers, which are then used to determine which strings lead to superior objective function values. These strings are treated as the genes of the cross-over and mutation strategy, and then combined to produce superior progeny. See [281] for a detailed coverage of this topic.

3.2.2 Tabu Search

The tabu search algorithm, which originated from the work in [100], has emerged as a widely-used meta-heuristic. It has been adapted to solve a large number of combinatorial-optimization problems. It has already appeared in some commercial simulation-optimization packages, e.g., OPTQUEST [103]. A distinctive feature of the tabu search algorithm is the so-called *tabu list*. This is a list of *mutations* that are prohibited in the algorithm. Let us illustrate the idea of mutations *in the context of tabu search* with a simple example.

As is perhaps clear from our discussion of the genetic algorithm, in a meta-heuristic, we *move* from one solution to another. Thus, if a problem with 2 decision variables is considered, in which both decision variables can assume values from the set $\{1, 2, 3\}$, a possible move is:

$$(2,1) \xrightarrow{\text{to}} (3,2).$$

In the above move, for the first decision variable, the "mutation" is $2 \xrightarrow{\text{to}} 3$, and for the second decision variable, the "mutation" is $1 \xrightarrow{\text{to}} 2$.

The tabu list is a finite-sized list of mutations that keeps changing over time. We now present step-by-step details of a tabu search algorithm. The algorithm is presented in terms of *minimization* of the objective function value.

Steps in tabu search. Let m denote the iteration number in the algorithm. Let $m_{\max}$ denote the maximum number of iterations to be performed. Like in the genetic algorithm, $m_{\max}$ has to be pre-specified, and there is no rule to find an optimal value for it. Also, as stated earlier, this number is based on the available computer time to run the algorithm.

Step 1. Set $m = 1$. Select an initial solution $\vec{x}_{current}$ randomly. Set:

$$\vec{x}_{best} \leftarrow \vec{x}_{current},$$

where $\vec{x}_{best}$ is the best solution obtained so far. Create N empty lists. Fix the maximum length of each of these lists to r. Both N and r are pre-specified numbers. *One* list will be associated with *each* decision variable.

Step 2. Select a neighbor of $\vec{x}_{current}$ arbitrarily. Call this neighbor $\vec{x}_{new}$. If all the tabu lists are empty, go to Step 4. Otherwise, go to Step 3.

Step 3. Consider the move $\vec{x}_{current} \xrightarrow{\text{to}} \vec{x}_{new}$. For this move, there is one mutation associated with each decision variable. Check if any of these mutations is present in its respective tabu list. If the answer is no, go to Step 4. Otherwise, the move is considered tabu (illegal); go to Step 5.

Step 4. Enter each mutation associated with $\vec{x}_{current} \xrightarrow{\text{to}} \vec{x}_{new}$ at the top of the respective tabu list. Then push down, by one position, all the entries in each list. If the tabu list has more than r members, as a result of this addition, *delete* the bottommost member. Then set

$$\vec{x}_{current} \leftarrow \vec{x}_{new}.$$

If the new solution is better than the best obtained so far, replace the best obtained so far by the current. That is if

$$f(\vec{x}_{current}) < f(\vec{x}_{best}), \text{ set } \vec{x}_{best} \leftarrow \vec{x}_{current}.$$

Step 5. Increment m by 1. If $m = m_{\max}$, STOP, and return $\vec{x}_{best}$ as the solution. Otherwise, go to Step 2.

The tabu list is thus a list of mutations that have been made recently. Maintaining the list avoids the re-evaluation of solutions that were examined recently. This is perhaps a distinguishing feature of this algorithm. It must be added, however, that in simulation optimization even if a solution is re-examined, it is not necessary to re-simulate the system. All the evaluated solutions can be stored in a so-called binary tree, which is a computer programming construct. Once a solution is simulated, its objective function value can be fetched every time it is needed from the binary tree, making re-simulation unnecessary.

Examples of tabu lists. Consider a problem with two decision variables: s and q, where each decision variable can assume values from:

$$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}.$$

Then the following listing is an example of what gets stored in a tabu list of length 3. Each element in this list is a move.

$$\begin{array}{cc}
s & q \\
2 \xrightarrow{\text{to}} 3 & 11 \xrightarrow{\text{to}} 12 \\
4 \xrightarrow{\text{to}} 3 & 2 \xrightarrow{\text{to}} 3 \\
3 \xrightarrow{\text{to}} 2 & 7 \xrightarrow{\text{to}} 8
\end{array}$$

At this point, if the mutation, $3 \xrightarrow{\text{to}} 2$, is generated for s in Step 3, it will not get stored since it is already in the tabu list. On the other hand, the mutation $3 \xrightarrow{\text{to}} 4$, is not tabu, and hence the new list for s will be:

$$\begin{array}{c}
s \\
3 \xrightarrow{\text{to}} 4 \\
2 \xrightarrow{\text{to}} 3 \\
4 \xrightarrow{\text{to}} 3
\end{array}$$

We need to make a few additional remarks.

Remark 1. An alternative interpretation of what is tabu—an interpretation commonly found in the literature—is to store the entire move as a mutation. In such an implementation, only one tabu list is maintained for the entire problem, and the entire move is treated as a mutation. An example of a tabu list for this implementation is:

$$\begin{array}{c}
(3,4) \xrightarrow{\text{to}} (4,5) \\
(2,6) \xrightarrow{\text{to}} (3,1) \\
(4,2) \xrightarrow{\text{to}} (2,5)
\end{array}$$

Remark 2. In yet another interpretation of what is tabu, whenever a move is selected, the reverse move is entered in the tabu list. For instance, if the algorithm makes the following move $(3,4) \xrightarrow{\text{to}} (4,5)$, then the move $(4,5) \xrightarrow{\text{to}} (3,4)$, is stored in the tabu list. This prevents cycling.

Remark 3. Our strategy for declaring a move to be tabu (in Step 3) may be overly restrictive. One way to work around this is to add a so-called aspiration criterion. A simple aspiration criterion, cited in [133], determines if the selected neighbor is actually better than the best solution so far. If the answer is yes, the tabu list consultation steps are skipped, the newly selected neighbor is treated as $\vec{x}_{current}$, and then one goes to Step 5. This would require a slight modification of Step 2, as shown below.

Step 2. Select a neighbor of $\vec{x}_{current}$ arbitrarily. Call this neighbor $\vec{x}_{new}$.

If $f(\vec{x}_{new}) < f(\vec{x}_{best})$ set:

$$\vec{x}_{best} \leftarrow \vec{x}_{new}, \text{ and } \vec{x}_{current} \leftarrow \vec{x}_{new},$$

and go to Step 5.

Otherwise, check to see if all the tabu lists are empty. If yes, go to Step 4 and if no, go to Step 3.

Of course, the aspiration criterion could be less strong. For instance, an aspiration criterion could determine how many of the new mutations are in their respective tabu lists. If this number is less than N, we could consider the new neighbor as non-tabu and accept it.

Remark 4. The length of the tabu list, r, should be a fraction of the problem size. In other words, for larger problems, larger tabu lists should be maintained. Very small tabu lists can cause cycling, i.e., the same solution may be visited repeatedly. Very large tabu lists can cause the algorithm to wander too much!

As is perhaps obvious from these remarks, there are various ways of implementing tabu search. There is a voluminous literature on tabu search, and the interested reader is referred to [104] for more details.

3.3. Stochastic Adaptive Search

This subsection is devoted to a number of stochastic adaptive search (SAS) techniques that (i) have proven global convergence properties and (ii) tend to "adapt" in the search process [333]. The implication of global convergence, as discussed above, is that the technique is guaranteed in the limit to generate the global optimum. The idea of adapting implies that new solutions are generated on the basis of past experience with previous solutions. As the name suggests, the technique uses a stochastic/probabilistic mechanism of some kind to generate new solutions.

We will begin our discussion with a simple search technique, called pure random search, that uses a stochastic mechanism in its search but does *not* adapt. The idea is to show how SAS techniques differ from a search technique that uses stochastic search but fails to adapt to results from previous iterations. The other motivation for discussing it is that pure random search can be used as the much needed benchmark against existing and newly developed meta-heuristic and SAS techniques. Obviously, if a technique takes more time (or iterations) than pure random search, then its worth is questionable. Unfortunately, both the genetic algorithm [255] and tabu search do not have guarantees of outperforming pure random search. Having said that, both are known to work well in practice and are hence regarded as *meta*-heuristics, probably implying that they defy logic and work in their own mysterious ways!

Pure random search. Pure random search selects every solution in the solution space with the same probability. The algorithm selects a solution randomly in every iteration. If the new solution is better than the best solution selected thus far, i.e., the current best, the new solution replaces the current best. This process is repeated a large number of times (theoretically infinitely many times). We now present a formal description in terms of maximizing the objective function.

Steps in pure random search. Let $(x(1), x(2), \ldots, x(N))$ denote N decision variables, where $x(i)$ assumes values from the finite set $\mathcal{A}(i)$. Thus $\mathcal{A}(i)$ denotes the finite set of values that are permitted for the ith decision variable. Let $p(i, a)$ denote the probability of selecting the value a for the ith decision variable. We define it as:

$$p(i, a) = \frac{1}{|\mathcal{A}(i)|} \text{ for } i = 1, 2, \ldots, N, \text{ and } a \in \mathcal{A}(i), \tag{5.5}$$

where $|\mathcal{A}(i)|$ denotes the number of elements (cardinality) in the set $\mathcal{A}(i)$. Set R_{best} to a small value that is smaller than the lowest possible value for the objective function. Set $m_{\max}$ to the maximum number of iterations permitted and m, the number of iterations, to 1.

Step 1. For $i = 1, 2, \ldots, N$, select a value $x(i)$ from the set $\mathcal{A}(i)$ with probability $p(i, x(i))$. Let the solution be denoted by $\vec{x}$. Evaluate the objective function at that point, i.e., $f(\vec{x})$, via simulation.

Step 2. If $f(\vec{x}) > R_{best}$, set $\vec{x}_{best} \leftarrow \vec{x}$ and $R_{best} \leftarrow f(\vec{x})$. Go to Step 3.

Step 3. Increment m by 1. If $m < m_{\max}$ return to Step 1. Otherwise, STOP.

We illustrate the above with a simple example.

Example. Consider a problem with two decision variables, where the first decision variable ($i = 1$) can assume three values, numbered 1, 2, and 3, while the second ($i = 2$) can assume two values, numbered 1 and 2. Thus, we have a problem with six different solutions. Then, using Eq. (5.5), we have:

$$p(1, a) = \frac{1}{3} \text{ for } a = 1, 2, 3 \text{ and } p(2, a) = \frac{1}{2} \text{ for } a = 1, 2.$$

Step 1. We now generate random numbers, U_1 and U_2, from the uniform distribution $Unif(0, 1)$ to select the values of the decision variables $x(1)$ and $x(2)$. If $U_1 < 1/3$, set $x(1) = 1$; if $1/3 \leq U_1 < 2/3$, set $x(1) = 2$; and otherwise, set $x(1) = 3$. If $U_2 < 1/2$, set $x(2) = 1$; else set $x(2) = 2$. After generating the solution, evaluate its objective function value via simulation.
Perform Steps 2 and 3.

It should be clear from the above that in every iteration a solution is randomly selected and that the probability of selecting any solution in any iteration is the same. The probability of selecting any solution in the above case is $\frac{1}{3} \cdot \frac{1}{2} = 1/6$ for each of the six solutions. In the limit, i.e., as m tends to infinity, the algorithm is guaranteed to visit *every* solution and hence also the global optimum.

Since the probability of selecting a solution does not change in the algorithm, pure random search is essentially *non-adaptive*. In contrast, all the remaining algorithms in this section will be adaptive, i.e., the probability of selecting a solution in any given iteration will change, directly or indirectly depending on what the algorithm has experienced in previous iterations. This perspective will be useful to the reader in viewing the motivation for any SAS technique and for judging its worth.

3.3.1 A Learning Automata Search Technique

We now discuss an SAS technique based on the theory of the so-called "common payoff" automata games in which the probability of selecting a solution is stored like in pure random search but is updated on the basis of the algorithm's experience with objective function values encountered. The technique was developed by [298] in 1987. We refer to it as the **L**earning **A**utomata **S**earch **T**echnique (acronym **LAST**).

LAST begins like pure random search by setting probabilities for selecting values for every decision variable. But with every iteration, it starts adapting to the function surface, eventually zeroing in on the global optimum. The process of adapting is achieved by updating the probabilities of selecting values. The objective function value of the solution selected in the iteration is used to update the probabilities in that iteration.

The scheme underlying the updating mechanism is quite simple. Decision variable values that produce "good" objective function values are rewarded via an increase in their probabilities of getting selected in the future, while those that produce "poor" objective function values are punished by a reduction in their probabilities. We now present a formal description of the underlying mechanism.

Let $(x(1), x(2), \ldots, x(N))$ denote N decision variables (parameters), where $x(i)$ takes values from the finite set $\mathcal{A}(i)$. Thus, $\mathcal{A}(i)$ denotes the finite set of values that are permitted for decision variable i. Let $p^m(i,a)$ denote the probability of selecting the value a for the ith decision variable in the mth iteration of the algorithm. As stated above, the algorithm starts as a pure random search. Mathematically, this implies that: $p^1(i,a) = \frac{1}{|\mathcal{A}(i)|}$ for $i = 1, 2, \ldots, N$, and every $a \in \mathcal{A}(i)$. The updating scheme of the algorithm that we will see below has to ensure that: $\sum_{a \in \mathcal{A}(i)} p^m(i,a) = 1$ for all (i,a)-pairs and every m. Since the probabilities are updated using the objective function values, the objective function value has to be *normalized* to a value between 0 and 1. This is achieved via:

$$F = \frac{R - R_{\max}}{R_{\max} - R_{\min}}, \tag{5.6}$$

where R denotes the *actual* (or raw) objective function value, F denotes the *normalized* objective function value, $R_{\max}$ denotes the maximum value for the actual objective function, and $R_{\min}$ denotes the minimum value for the actual objective function. Knowledge of $R_{\max}$ and $R_{\min}$ is necessary for this algorithm. If these values are not known, one must use guessed estimates.

The best normalized objective function value, obtained thus far in the algorithm, will be denoted by $B(i,a)$ for $i = 1, 2, \ldots, N$ and $a \in \mathcal{A}(i)$. We will need a constant step size, to be denoted by μ, in the updating. In general, $\mu \in (0,1)$; e.g., $\mu = 0.1$. We present the algorithm in terms of *maximizing* the objective function value.

Steps in LAST.

Step 1. Set the number of iterations, m, to 1. Let N denote the number of decision variables. Set $p^m(i,a) = 1/|\mathcal{A}(i)|$ and $B(i,a) = 0$ for $i = 1, 2, \ldots, N$ and every $a \in \mathcal{A}(i)$. The upper limit on the number of iterations to be performed will be denoted by $m_{\max}$. Assign suitable values to μ, $R_{\max}$, $R_{\min}$, and $m_{\max}$. Set $F_{best} = 0$.

Step 2. For $i = 1, 2, \ldots, N$, select a value $x(i)$ from $\mathcal{A}(i)$ with probability $p^m(i, x(i))$. Let the new solution be denoted by $\vec{x}$.

Step 3. Evaluate the objective function value associated with $\vec{x}$. Let the value obtained be denoted by R. Calculate the normalized objective function value, F, using Eq. (5.6). If $F > F_{best}$, set

$$\vec{x}_{best} \leftarrow \vec{x} \text{ and } F_{best} \leftarrow F.$$

Step 4. Set $i = 1$.

Step 5. For $a = 1, 2, \ldots, |\mathcal{A}(i)|$, do:

If $B(i, a) < B(i, x(i))$, set

$$p^{m+1}(i, a) \leftarrow p^m(i, a) - \mu[B(i, x(i)) - B(i, a)]p^m(i, a).$$

If $B(i, a) > B(i, x(i))$, set

$$p^{m+1}(i, a) \leftarrow p^m(i, a) + \mu[B(i, a) - B(i, x(i))]\frac{[1 - p^m(i, a)]p^m(i, x(i))}{|\mathcal{A}(i)| - 1}.$$

Step 6. Set

$$p^{m+1}(i, x(i)) \leftarrow 1 - \sum_{a \neq x(i); a=1}^{a=|\mathcal{A}(i)|} p^{m+1}(i, a).$$

If $i < k$, increment i by 1, and go back to Step 5. Otherwise, increment m by 1, and go to Step 7.

Step 7. If $m < m_{\max}$, go to Step 8; otherwise STOP returning $\vec{x}_{best}$ as the solution.

Step 8. (Updating **B**) For $i = 1, 2, \ldots, N$, do:

$$\text{If } F > B(i, x(i)), \text{ set } B(i, x(i)) \leftarrow F.$$

Step 9. Return to Step 2.

Remark 1. LAST is likely to revisit the same solution a number of times, which may imply repeated simulation of the same solution. This difficulty can be circumvented, like in the case of tabu search, by maintaining a dynamic memory structure, e.g., binary tree, which stores all the solutions that were evaluated in the past. Whenever a solution that was tried previously is re-visited, its objective function's value is fetched from the memory, thereby avoiding re-simulation. When a newly generated solution is identical to the solution in the previous iteration, it is clearly unnecessary to re-simulate the new solution. Hence, even if a dynamic memory structure is not maintained, the optimization problem should always check for this before simulating a new solution.

Remark 2. The size of the **B** matrix and the number of probabilities are very small compared to the solution space. For instance, in a problem with 10 decision variables with 2 values for each variable, the solution space is 2^{10}, but the number of elements in the **B** matrix is only $(10)(2) = 20$, and the number of probabilities is also 20. Further, the matrix **B** can also be stored as a binary tree, thereby reducing the memory requirements. This is possible because as long as a pair (i, a) is not tried, its value is 0, and hence the matrix is sparse.

Remark 3. The problem of requiring the storage of a large number of many probabilities in large-scale problems can sometimes be overcome by using the so-called parameter dependence network [262], which uses Bayesian learning theory. The network can be integrated within the simulator to *adaptively* search the decision variable space.

Remark 4. $R_{\min}$ and $R_{\max}$ should actually be the greatest lower bound (GLB) and the lowest upper bound (LUB), respectively, for the objective function value in case of maximization. If these bounds are not known, any conservative values for the upper and lower bounds can be used. What is essential is that the value of the objective function value is restrained to the interval $(0, 1)$. We note, however, that if as a result of guessing, $R_{\min} \ll GLB$ and $R_{\max} \gg LUB$, updating can become very slow. Hence, these values should be chosen carefully, making sure that updating is not unacceptably slow.

Remark 5. In LAST, a new solution is not necessarily "near" the current solution. Rather, it can be anywhere in the solution space. Thus, the search is not sequential but geared towards searching the entire feasible space, which helps in the quest for the global optimum.

Example. Consider a small problem with two decision variables where each decision variable has three values, numbered 1, 2, and 3. We join the "learning process" after m iterations. As a result, the **B** matrix will not be empty. Let us assume that the **B** matrix after m iterations is:

$$\begin{bmatrix} 0.1 & 0.2 & 0.4 \\ 0.1 & 0.3 & 0.4 \end{bmatrix}.$$

Let the value of $\vec{x}$ selected in the $(m+1)$th iteration be $(2,1)$. In other words, for the first decision variable $a = 2$ was selected and for the second $a = 1$ was selected. Let the objective function value, F, be 0.1. F_{best}, it should be clear from the **B** matrix, is assumed to be 0.4. We now show all the calculations to be performed at the end of this iteration.

Now from Step 5, since $B(1,1) < B(1,2)$, $p(1,1)$ will decrease and will be updated as follows.

$$p^{m+1}(1,1) = p^m(1,1) - \mu[B(1,2) - B(1,1)]p^m(1,1).$$

The probability $p(1,3)$ will increase, since $B(1,3) > B(1,2)$, and the updating will be as follows:

$$p^{m+1}(1,3) = p^m(1,3) + \mu[B(1,3) - B(1,2)]\frac{[1 - p^m(1,3)]p^m(1,2)}{3-1}.$$

And finally from Step 6, $p(1,2)$ will be updated as follows:

$$p^{m+1}(1,2) = 1 - p^{m+1}(1,1) - p^{m+1}(1,3).$$

Similarly, we will update the probabilities associated with the second decision variable. Here both $p(2,2)$ and $p(2,3)$ will increase, since both $B(2,2)$ and $B(2,3)$ are greater than $B(2,1)$. The updating equations are as follows:

$$p^{m+1}(2,2) = p^m(2,2) + \mu[B(2,2) - B(2,1)]\frac{[1 - p^m(2,2)]p^m(2,1)}{3-1} \text{ and}$$

$$p^{m+1}(2,3) = p^m(2,3) + \mu[B(2,3) - B(2,1)]\frac{[1 - p^m(2,3)]p^m(2,1)}{3-1}.$$

The third probability will be normalized as follows:

$$p^{m+1}(2,1) = 1 - p^{m+1}(2,2) - p^{m+1}(2,3).$$

Since both $B(1,2)$ and $B(2,1)$ are greater than F, the new response will not change the **B** matrix. Thus the new **B** matrix will be identical to the old. And then we conduct the $(m+2)$th iteration, and the process continues

3.3.2 Simulated Annealing

This algorithm has been often hailed as a breakthrough in this field. It was first formally presented in 1983 in the work of [169]. The mechanism of simulated annealing is straightforward. An arbitrary solution is selected to be the starting solution. It helps to start at a good solution, but this is not necessary. In each iteration, the algorithm tests one of the neighbors of the current solution for its function value. If the neighbor is better than (or equally good as) the current solution (in terms of its objective function value), the algorithm moves to the neighbor. If the neighbor is worse, the algorithm stays at the current solution with a high probability but moves to it with a "low" probability. Moving to a worse solution is also called *exploration.*

Several iterations of the algorithm are typically needed. As the algorithm progresses, the probability of exploration is reduced. The algorithm terminates when the probability of moving to a worse neighbor approaches zero. At each iteration, the best solution obtained thus far is stored separately. As a result, the best of all the solutions encountered is never lost from the memory and is eventually returned as the "best" solution discovered by the algorithm.

Empirically, simulated annealing has been shown to return an **optimal** solution on small problems in many well-publicized tests; remember on small problems, one can determine the optimal solution by an exhaustive evaluation of the solution space, and hence it is possible to determine whether the algorithm can return the optimal. On large-scale problems, however, this is usually not feasible. In many empirical tests on large-scale problems, the algorithm has been reported to outperform *other heuristics.* More importantly, convergence proofs have been developed which show that the algorithm has the potential to return the optimal solution asymptotically (i.e., as the number of iterations tends to infinity) provided the probability of moving into a worse neighbor is decreased "properly."

Is simulated annealing an SAS technique? It will be evident from the description below that (i) the algorithm's behavior depends on the objective function values from previous iterations and (ii) it uses a probabilistic mechanism to generate new solutions. In addition, as stated above, it has mathematical proofs of convergence. Hence, it is appropriately considered to be an SAS technique.

The simulated annealing algorithm is so named because of its similarity with the "annealing" process in metals. This annealing process in metals requires that the metal's temperature be raised and then *gradually* lowered so that it acquires desirable characteristics related to hardness. The simulated annealing algorithm's behavior is analogous to this metallurgical process because it starts with a relatively high probability (read temperature) of moving to worse neighbors but the probability is gradually reduced. The analogy can be taken a little further. At high temperatures, the atoms in metals have significant vibrations. The vibrations gradually reduce with the temperature. Similarly, at the start, the algorithm has a significant ability to move out to what appear to be poor solutions, but this ability is reduced (or rather should be reduced) as the algorithm progresses.

Although this analogy serves to help our intuition, it will be prudent to caution the reader about such analogies because associated with them is an inherent danger of overlooking the actual reason for the success (or failure) of an algorithm. Ultimately, mathematical arguments have to be used to determine if the method is guaranteed to converge. Fortunately, for simulated annealing, there is increasing evidence of the mathematical kind.

Steps in simulated annealing. Let $f(\vec{x})$ denote the value of the objective function at $\vec{x}$. Choose an initial solution and denote it by $\vec{x}_{current}$. Let $\vec{x}_{best}$ denote the best solution thus far. Set: $\vec{x}_{best} \leftarrow \vec{x}_{current}$. Set T, the "temperature," to a pre-specified value.

The temperature is gradually reduced. But at each temperature, Steps 2 and 3 are performed for a number of iterations. This is called a *phase* in the algorithm. This implies that each phase consists of several iterations. The number of iterations in each phase should generally increase with the number of phases. The steps below are written for *minimizing* the objective function.

Step 1. Set P, the number of phases conducted thus far, to 0.

Step 2. Randomly select a neighbor of the current solution. Selection of neighbors is discussed below in Remark 1. Denote the neighbor by $\vec{x}_{new}$.

Step 3. If $f(\vec{x}_{new}) < f(\vec{x}_{best})$, then set: $\vec{x}_{best} \leftarrow \vec{x}_{new}$.

Let $\Delta = f(\vec{x}_{new}) - f(\vec{x}_{current})$.

- If $\Delta \leq 0$, set:
$$\vec{x}_{current} \leftarrow \vec{x}_{new}.$$
- Otherwise, that is, if $\Delta > 0$, generate a uniformly distributed random number between 0 and 1, and call it U. If
$$U \leq \exp(-\frac{\Delta}{T}), \text{ set: } \vec{x}_{current} \leftarrow \vec{x}_{new}; \text{ else keep } \vec{x}_{current} \text{ unchanged.}$$

Step 4. One execution of Steps 2 and 3 constitutes one iteration of a phase. Repeat Steps 2 and 3 until the maximum number of iterations permitted for the current phase are performed. See Remark 2 for the maximum number permitted. When these iterations are performed, go to Step 5.

Step 5. Increment the phase number P by 1. If $P < P_{\max}$, then reduce T (temperature reduction schemes will be discussed below in Remark 2) and go back to Step 2 for another phase. Otherwise, terminate the algorithm and declare $\vec{x}_{best}$ to be the best solution obtained.

Remark 1. In general, the neighbor selection strategy affects the amount of time taken to produce a good solution. The hit-and-run strategy discussed above is one possible strategy for generating neighbors. Other strategies have been discussed in [154, 55, 43, 291].

Remark 2. The issue of temperature reduction has generated a great deal of debate leading to valuable research. It is well-known that the algorithm can perform poorly when the temperature is not reduced at an appropriate rate. Furthermore, finding the right strategy for temperature reduction may need experimentation. Two issues are relevant here.

i. What should the number of iterations in one phase (remember, in each phase, the temperature is kept at a constant value) be?

ii. How should the temperature be reduced?

The answer to the first question can lead to two different categories of algorithms. We have named them *impatient* and *patient*.

1. *Impatient*: In the impatient category, a fixed number of iterations is associated with each phase, where the number may depend on the phase number. Once the number of iterations associated with a phase are complete, the algorithm moves to the next phase and reduces the temperature. *The number of iterations does not depend on whether the algorithm has actually found a point better than the current point,* i.e., the algorithm does not wait for the objective function to improve, but once its quota of iterations per phase is complete, it moves on to the next phase. For instance, one may choose to perform $m_{\max}$ iterations per phase, where $m_{\max}$ does not depend on P with $m_{\max}$ ranging from 1 to any large number. Another strategy is to use a $m_{\max}$ that increases with P. A simple linear rule that accomplishes this is: $m_{\max}(P) = A + B \cdot P$, where $A \geq 1$ and $B \geq 0$ are user-specified integers. A quadratic rule towards the same goal would be as follows: $m_{\max}(P) = A + B \cdot P^2$.

2. *Patient*: This strategy was proposed in [248, 249], where one does not terminate a phase until a better solution is found. Thus the

number of iterations per phase may be very large or very small depending on how the algorithm performs. In practice, the number of iterations will likely be a random variable that changes with every phase. It is possible that the actual number of iterations required per phase may actually increase in practice with the phase number, P, like in the increasing functions of the impatient variety, but the number will depend on the function, unlike the arbitrary number imposed in the impatient variety.

To answer the second question, a general temperature-reduction strategy is to make the temperature a decreasing function of P. Many rules have been proposed in the literature to this end. We present three categories of rules that can be employed: logarithmic, geometric, and rational.

1. *Logarithmic*: A logarithmic rule (see e.g., [99, 130]) is

$$T(P) = \frac{C}{ln(2+P)}, \tag{5.7}$$

   where $T(P)$ is the temperature for phase P, P starts at 0, and C is a user-specified positive constant. The value of C is usually determined empirically. Small values of C can cause the temperature to decay very quickly (reducing hopes of reaching the global optimum).

2. *Geometric*: A geometric rule (see e.g., [169]) is

$$T(P+1) = \lambda T(P), \tag{5.8}$$

   in which $0 < \lambda < 1$ (e.g., $\lambda = 0.99$) and $T(0) = C$, where $C > 0$ is user-specified. An equivalent rule [333] is $T(P) = C(\lambda)^P$.

3. *Rational*: A rule based on a rational function is (also used in neural networks and reinforcement learning)

$$T(P) = \frac{C}{B+P}, \tag{5.9}$$

   where C and B are user-specified positive constants, e.g., $C = 1$ and $B = 0$ [291]; $C > 1$ and $B > C$ (used typically in reinforcement learning, see e.g., [113]). Another rule is from [193]: $T(P) = T(0)/(1 + B \cdot T(0) \cdot P)$, where $B \ll 1$.

Remark 3. The expression $\exp(-\frac{\Delta}{T})$ with which U (the random number between 0 and 1) is compared needs to be studied carefully.

For small positive values of T, this expression also assumes small positive values. When the expression is small, so is the probability of accepting a worse solution. A general rule of simulated annealing is that T should decrease as the number of phases increases. Thus, for instance if the rules discussed above are used for temperature decay, C (and other relevant constants) should be chosen in a manner such that when P is zero, $\exp(-\frac{\Delta}{T})$ is significantly larger than zero. Otherwise the probability of selecting a worse solution will be very small at the start itself, which can cause the algorithm to get trapped in the nearest local optimum, essentially negating the idea of exploring. Note also that if the temperature starts at a high value and is never reduced, and in addition, the number of iterations per phase keeps increasing, the algorithm essentially becomes a "wanderer," which is equivalent to a pure random search. Thus, one should start with a sufficiently high temperature *and* decay the temperature.

Remark 4. When we use simulated annealing with a simulator, we assume that the estimate produced by the simulator is "close" to the actual function value. In reality, there is some noise/error. Fortunately, as long as the noise is not too "large," the algorithm's behavior is not impacted (see Chap. 10). It may be a good idea, however, to increase the accuracy of the function estimation process, by increasing the number of replications, as the algorithm progresses.

Remark 5. The reason for allowing the algorithm to move to *worse* solutions is to provide it with the opportunity of moving away from a local optimum and finding the global optimum. See Fig. 5.1 (see page 76). A simulated annealing algorithm that finds X in Fig. 5.1 may still escape from it and go on to find the global optimum, Y. Remember that if the algorithm moves out of a local optimum and that local optimum happens to be a global optimum, the global optimum is not lost because the best solution is always retained in the algorithm's memory; such algorithms are called *memory-based.*

Remark 6. Finally, an important question is: how many phases ($P_{\max}$) should be performed? The answer depends on how the temperature is reduced. When the temperature approaches small values at which no exploration occurs, the algorithm should be stopped. The rate at which the temperature is reduced depends on how much time is available to the user. Slower the decay the greater the chance of exploring the entire solution space for finding the global optimum.

Example. We will demonstrate a few steps in the simulated annealing algorithm with an example. The example will be one of minimization. Consider a problem with two decision variables, x and y, each taking values from the set: $\{1, 2, 3, 4, 5, 6\}$. We will assume that one iteration is allowed per phase. The temperature is decayed using the following rule: $T(P) = 100/ln(2 + P)$. Let the current solution be: $\vec{x}_{current} = (3, 4)$. The same solution is also the best solution currently; in other words: $\vec{x}_{best} = (3, 4)$. Let $f(\vec{x}_{current})$ be 1,400.

Step 1. P is equal to 0. Hence, $T = 100/ln(2) = 144.27$.

Step 2. Let the selected neighbor of the current solution be: $\vec{x}_{new} = (2, 5)$, where $f(\vec{x}_{new}) = 1{,}350$.

Step 3. Since $f(\vec{x}_{new}) < f(\vec{x}_{best})$,

$$\vec{x}_{best} = \vec{x}_{new} = (2, 5).$$

$$\text{Now: } \Delta = f(\vec{x}_{new}) - f(\vec{x}_{current}) = 1{,}350 - 1{,}400 = -50.$$

Since $\Delta < 0$, set: $\vec{x}_{current} = \vec{x}_{new} = (2, 5)$.

Step 4. Since only one iteration is to be done per phase, we move to Step 5.

Step 5. Increment P to 1. Since $P < P_{\max}$, re-calculate T to be $2/ln(3) = 91.02$.

Step 2. Let the selected neighbor of the current solution be $\vec{x}_{new} = (1, 6)$, where $f(\vec{x}_{new}) = 1{,}470$.

$$\text{Hence } \Delta = f(\vec{x}_{new}) - f(\vec{x}_{current}) = 1{,}470 - 1{,}350 = 120.$$

Since $\Delta > 0$, generate U, a uniformly distributed random number between 0 and 1. Let U be 0.1. Now:

$$\exp(-\frac{\Delta}{T}) = \exp(-120/91.2) = 0.268.$$

Since $U = 0.1 < 0.268$, we will actually move into the worse solution. Hence, $\vec{x}_{current} = \vec{x}_{new} = (1, 6)$ and so on.

3.3.3 Backtracking Adaptive Search

Backtracking **A**daptive **S**earch, abbreviated as **BAS**, is due to [182]. Its convergence properties (e.g., the ability to generate the global optimum) are better understood than those of simulated annealing. Like simulated annealing, this algorithm moves to worse solutions with some probability, but unlike simulated annealing, this probability does not depend on a decaying temperature. The probability depends only on the values of the objective function of the current point and the point to which a move is being considered by the algorithm. Also, the neighbor-generating strategy will use a stochastic generator matrix that we first discuss.

Stochastic generator matrix. Consider a simple problem where there are three solutions, which are indexed as 1, 2, and 3. Now consider the following matrix:

$$\mathbf{G} = \begin{bmatrix} 0 & 0.2 & 0.8 \\ 0.3 & 0 & 0.7 \\ 0.1 & 0.9 & 0 \end{bmatrix}.$$

The sum of elements in any row of this matrix sum to 1, which means that it is a so-called *stochastic* matrix. Such matrices will be covered extensively from the next chapter onwards in the control optimization setting. Here it is sufficient for the reader to view this matrix as an entity that can randomly *generate* a new solution from a current solution. The generation mechanism works as follows: If the algorithm is currently in a solution indexed by i, then the probability with which it be moved to a solution indexed by j is given by $G(i, j)$. We consider an example next.

Assume that the algorithm is in the solution indexed by 2. Then, the probability that it will move to the solution with index i is given by $G(2, i)$. Thus, it will move to the solution indexed as 1 with a probability of $G(2, 1) = 0.3$ and to the solution indexed as 3 with a probability of $G(2, 3) = 0.7$. In order to achieve the move, one generates a uniformly distributed random number, U, between 0 and 1. If $U \leq 0.3$, the neighbor (new solution) is the solution indexed as 1, while if $U > 0.3$, the solution indexed as 3 becomes the neighbor. Note that we have assumed the diagonal elements in $\mathbf{G}$ to be 0 above, since it will be clearly inefficient in simulation optimization to consider the same point again as a neighbor.

Steps in BAS. Set the number of iterations, m, to 1. Let $f(\vec{x})$ denote the value of the objective function (obtained via simulation) at $\vec{x}$. Select a solution randomly from the feasible space and denote it by $\vec{x}_{current}$. Let $\vec{x}_{best}$ denote the best solution so far. Set $\vec{x}_{best} \leftarrow \vec{x}_{current}$. Initialize values to the stochastic generator matrix $\mathbf{G}$. Also initialize $m_{\max}$, the maximum number of iterations permitted. The algorithm is written for *minimizing* the objective function.

Step 1. Generate a neighbor of the current solution using the stochastic matrix $\mathbf{G}$. Denote the neighbor by $\vec{x}_{new}$.

Step 2. Set $\vec{x}_{current} \leftarrow \vec{x}_{new}$ with a probability of $A(current, new)$, where *current* and *new* denote the indices of the current and the new solutions respectively. See Remark 1 below for more on how the probability $A(current, new)$ is computed. *This probability is 1 if the new solution is equally good or better than the current solution.* With a probability of $1 - A(current, new)$, do not change the solution. If $f(\vec{x}_{new}) < f(\vec{x}_{best})$, set $\vec{x}_{best} \leftarrow \vec{x}_{new}$. Otherwise, if the new solution is not better than the best, do not change the best solution.

Step 3. Increment m by 1. If $m < m_{\max}$, return to Step 1. Otherwise terminate the algorithm and declare $\vec{x}_{best}$ to be the best solution obtained.

Remark 1. The acceptance probability, $A(current, new)$, as stated above, is the probability of accepting a new solution indexed by *new* when the current solution is indexed by *current*. We will always assume, in accordance with the original description [182, 333], that this probability will equal 1 if the new solution is as good as or better than the current solution. When the new solution is worse than the current solution, this probability will depend on the values of the objective function at the current and the new solutions. For instance, we could define the acceptance probability as:

$$A(current, new) = \begin{cases} 1 & \text{if } f(\vec{x}_{new}) \leq f(\vec{x}_{current}) \\ \exp(-\frac{f(\vec{x}_{new}) - f(\vec{x}_{current})}{T}) & \text{otherwise} \end{cases}, \tag{5.10}$$

where $T > 0$ does not depend on the iteration, but may depend on the values of the objective function. Hence here T should *not* necessarily be viewed as the temperature of simulated annealing. In general T is a function of $f(\vec{x}_{new})$ and $f(\vec{x}_{current})$. Other mechanisms for generating the matrix $\mathbf{A}$ can also be used as long as $A(current, new)$ equals 1 when the function finds an improved/equally good point.

Remark 2. Stopping criteria other than m_{max} can also be used. Essentially, the value of $m_{\max}$ depends on the time available to the analyst. In global optimization, unless the algorithm has the opportunity to sample the entire solution space, the chances of finding the global optimum are low. Hence, higher this value, the better the performance is likely to be.

Remark 3. The algorithm follows the format of simulated annealing with an important difference: The exploration/backtracking probability in BAS does not depend on the iteration number but only on the objective function values of the current and the new solution. In simulated annealing, the exploration probability depends, in addition to the objective function values, on the "temperature," which in turn depends on the number of iterations the algorithm has performed thus far. Also, the convergence properties of the two algorithms are markedly different, which we will discuss in Chap. 10.

Remark 4. Note that we did not define stochastic generator matrices in simulated annealing or LAST, because it was not necessary to store them explicitly in the computer's memory. However, intrinsically, such generators exist underlying all SAS techniques. In simulated annealing, we discussed the hit-and-run strategy. Parameters underlying the hit-and-run strategy can in fact be used to compute this matrix. In LAST, we can generate this matrix from the probabilities used in the solution. For instance, in a three-solution problem assume that in a given iteration, the probabilities of selecting the solutions, indexed 1, 2, and 3, are $0.2, 0.3$ and 0.5 respectively. Then, the stochastic generator matrix in LAST for that iteration is:

$$\mathbf{G} = \begin{bmatrix} 0.2 & 0.3 & 0.5 \\ 0.2 & 0.3 & 0.5 \\ 0.2 & 0.3 & 0.5 \end{bmatrix},$$

where every row is identical. Also, note that in every iteration of LAST, this matrix changes.

The remaining two techniques that we discuss have features designed for simulation optimization. The techniques we discussed above were designed for global optimization of deterministic problems.

3.3.4 Stochastic Ruler

The stochastic ruler was probably one of the first discrete optimization techniques designed for simulation optimization. The original stochastic ruler is from [329]. Here, we present a modified version from [6], which is often called the *modified stochastic ruler*. The algorithm relies on a stochastic generator matrix (discussed above), which has to be generated in a specific manner that we discuss below. In general, the generator matrix is defined as follows in [6]:

$$G(i,j) = G'(i,j)/W(i),$$

where i and j are solution indices, $W(i) > 0$ for all i, and $G'(i,j) \geq 0$ for all i, j. Further $\sum_j G(i,j) = 1$ for every i. The generator can be defined in numerous ways (see [6]). We will discuss one specific mechanism, which we call the *uniformly distributed* mechanism, for illustration purposes. Before introducing it, we need some notation.

Let $\mathcal{N}(i)$ denote the set of neighbors of the solution indexed by i. This set may or may not include all the points (solutions) in the solution space, however, it must include at least one point other than i. *Note that it is not necessary for it to include i.*

Under the uniformly distributed mechanism, the generator matrix is:

$$G(i,j) = \begin{cases} \frac{1}{|\mathcal{N}(i)|} & \text{if } j \in \mathcal{N}(i) \\ 0 & \text{otherwise} \end{cases}.$$

Here, $G'(i,j) = 1$ for every (i,j)-pair when $j \in \mathcal{N}(i)$ and $G'(i,j) = 0$ otherwise; also $W(i) = |\mathcal{N}(i)|$. We illustrate the mechanism with a simple example.

Example for the G matrix. Assume that we have four solutions in the solution space, which are indexed as 1, 2, 3, and 4. Further, assume that $\mathcal{N}(1) = \{1,2\}$; $\mathcal{N}(2) = \{1,2,3\}$; $\mathcal{N}(3) = \{2,3,4\}$; and $\mathcal{N}(4) = \{3,4\}$. Then, $|\mathcal{N}(1)| = 2$; $|\mathcal{N}(2)| = 3$; $|\mathcal{N}(3)| = 3$; and $|\mathcal{N}(4)| = 2$. Then,

$$\mathbf{G} = \begin{bmatrix} 1/2 & 1/2 & 0 & 0 \\ 1/3 & 1/3 & 1/3 & 0 \\ 0 & 1/3 & 1/3 & 1/3 \\ 0 & 0 & 1/2 & 1/2 \end{bmatrix}.$$

When $W(i) = |\mathcal{N}(i)|$, we will refer to $W(i)$ as the *friendliness coefficient* of i, indicating that it is a measure of how many candidate solutions (neighbors) can be generated from a solution.

The basic idea in the stochastic ruler is straightforward. Assume that a and b are the lower and upper bounds, respectively, of the objective function, which we wish to *minimize.* Further assume that both bounds are known. Clearly then, the value of the objective function at the optimal solution is a or a value very close to a. Now if we generate random numbers from the distribution $Unif(a,b)$, then the probability that a generated random number will exceed a solution's objective function value should equal 1 when the solution *is* the optimal solution. However, if the solution is not optimal, this probability should be less than 1. Also, if the solution is at its worst point (where the objective function value is b), this probability should be 0. The stochastic ruler essentially seeks to maximize this probability, striving to move the solution to points where this probability is increased. In the limit, it reaches a point where this probability is maximized, which should clearly be the optimal solution.

The working mechanism of the algorithm is simple. One starts with any randomly selected solution as the current solution. A candidate (neighbor of a current solution) is chosen to determine if it is better than the current solution. To this end, replications of the candidate solution are performed via simulation. The ith replication is compared to the random number, U_i, generated from the distribution $Unif(a, b)$. A maximum of I (a pre-determined number) replications are performed. If U_i exceeds the value of the replication in each of the I occasions, then the new solution replaces the current solution, and the search continues. If U_i turns out to be smaller in any of the replications, the candidate is immediately rejected, performing no further replications. Also, the current solution is kept unchanged, and a new candidate is selected using the current solution.

We are now ready to present this algorithm formally. We present it in terms of minimizing the objective function value.

Steps in the modified stochastic ruler.

Step 1. Let the number of visits to a solution $\vec{x}$ be denoted by $V(\vec{x})$. Set $V(\vec{x})$ to 0 for all solutions. Set m, the number of iterations, to 1. Choose any solution to be the starting solution, and denote it by $\vec{x}_{current}$. Let $\vec{x}_*$ denote the estimated optimal solution. Set $\vec{x}_* \leftarrow \vec{x}_{current}$. Assign suitable values for a, b, $m_{\max}$, and I, the maximum number of replications performed for a solution in a given iteration. Select **G** and $W(.)$.

Step 2. Using the generator matrix **G** generate a neighbor. Denote the neighbor by $\vec{x}_{new}$. Set i, the replication number, to 0.

Step 3. Increment i by 1.

Step 4. Perform a replication of the solution $\vec{x}_{new}$ and denote its value by $f_i(\vec{x}_{new})$. Generate a random number, U_i, from the distribution $Unif(a, b)$.

- If $U_i < f_i(\vec{x}_{new})$: the implication is that in all likelihood there exist solutions that are better than the new solution, indicating that this (new) solution should be discarded. Hence, do not change the current solution and go to Step 5.
- Otherwise if $U_i \geq f_i(\vec{x}_{new})$: the implication is that the new solution may be better than the current solution. Hence, check to see if $i = I$. If yes, accept the new solution, i.e., set $\vec{x}_{current} \leftarrow \vec{x}_{new}$, and go to Step 5. Otherwise return to Step 3 to perform one more replication.

Step 5. Set $V(\vec{x}_{current}) \leftarrow V(\vec{x}_{current}) + 1$. If

$$\frac{V(\vec{x}_{current})}{W(\vec{x}_{current})} > \frac{V(\vec{x}_*)}{W(\vec{x}_*)},$$

set $\vec{x}_* \leftarrow \vec{x}_{current}$. Increment m by 1. If $m = m_{\max}$, STOP; otherwise return to Step 2.

Note that $\vec{x}_*$ will be returned as the estimated optimal solution from the algorithm. The optimal solution is estimated using the ratio test in Step 5, which constitutes a significant difference between the algorithm presented above and the original version of [329]. We can explain the intuition underlying this ratio test as follows whenever the friendliness coefficient, $W(.)$, is proportional to $|\mathcal{N}(.)|$: The friendliness coefficient in the denominator of the ratio is proportional to how many times the algorithm leaves the solution, while the numerator in the ratio is an indicator of how many times the algorithm enters the solution. Hence, a high value for the ratio is a measure of the attractiveness of a solution for the algorithm. After all, by its design, the algorithm has the highest incentive (probability) to visit the optimal and the least incentive to leave it. The ratio will therefore be the highest for the optimal solution.

The values of I and $m_{\max}$ depend on the availability of computational time. Clearly, large values are preferred for $m_{\max}$. Also, it is very unusual in simulations to use values less than 4 for I, but a larger value is likely to improve the efficacy of the algorithm. I is often increased with m to improve the accuracy as one approaches the optimal solution, like the number of iterations per phase is increased as the temperature falls in simulated annealing.

A special feature of this algorithm is that it is likely to discard a poor solution quickly without performing too many replications. This feature should on the average save a great deal of time in simulation optimization. In the algorithms preceding the stochastic ruler, we have assumed that *all* required replications are performed for a candidate (neighbor) when its objective function value is determined. The stochastic ruler is perhaps the first algorithm to utilize this aspect of simulation in optimization. Using a small value of I in the early iterations can also lead to significant time savings without sacrificing accuracy.

3.3.5 Nested Partitions

We conclude our discussion on discrete optimization with the nested partitions algorithm [273, 274, 275] that has had a significant influence on the field. The algorithm is somewhat unique in its strategy of exploring the solution space of a problem. It seeks to group solutions together into clusters called *regions*. The region is hence a set of solutions. Regions are formed in a way such that the union of all the regions forms the entire feasible solution space. In every iteration of the algorithm, each region is sampled for solutions, which are then simulated using one replication per solution. The region that yields the best value for the objective function is used to generate a so-called *promising* region for the next iteration. In the next iteration, one further partitions (divides) the promising region into sub-regions, and the process continues until one identifies a region carrying a single solution that is visited most frequently by the algorithm. Asymptotically, this region is guaranteed to be an optimal solution.

The partitioning approach has similarities with the notion of search used in a binary tree, which however divides the space it considers into only *two* regions at a time. However, nested partitions provides a more general framework to exhaustively explore the solution space. The precise mechanism to partition the solution space is left to the analyst however. We will now define some terms and notation that we will need.

- $\mathcal{S}$: The entire feasible solution, i.e., the set of all the solutions in the problem.
- A *singleton* set, i.e., a set with just one solution.
- $\mathcal{S}_0$: The set of all singleton sets in the problem.
- An *empty* set: A set that has no solutions.
- A *region*: A set whose elements are one or more of the solutions.

Consider a small discrete-optimization problem with five solutions, indexed by $1, 2, 3, 4$ and 5. Then, $\mathcal{S} = \{1, 2, 3, 4, 5\}$. Further, using our definitions above, a set such as $\{1, 3\}$ will be considered to be a *region* consisting of the solutions indexed by 1 and 3. Also each of the sets $\{1\}, \{2\}, \{3\}, \{4\}$, and $\{5\}$ will be considered to be singleton sets. Thus, for this problem,

$$\mathcal{S}_0 = \{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}\}.$$

The so-called "promising" region will be the region that is likely on the basis of the algorithm's computations to contain the global optimum. The promising region will be updated in every iteration based on the information obtained by the algorithm as it makes progress.

The algorithm will eventually repeatedly identify a singleton set as the promising region, which will be returned as the optimal solution.

In what follows, we present a version of the algorithm described in [274]. We present the algorithm in terms of *minimizing* the objective function value.

Steps in nested partitions.

Step 1. Set m, the number of iterations, to 1. Let $\mathcal{F}(m)$ denote the promising region in the mth iteration. Set $\mathcal{F}(m) = \mathcal{S}$ where $\mathcal{S}$ denotes the entire feasible region for the problem. $\mathcal{S}_0$ will denote the set of the singleton sets in the problem. For every singleton set, $\mathcal{X}$, in the problem, set $V(\mathcal{X})$ to 0. Here, $V(\mathcal{X})$ will denote the number of times the algorithm has selected (visited) the singleton set $\mathcal{X}$ as a promising region. Set $\mathcal{X}^*$, the estimated optimal solution, to any singleton set in the problem. Assign a suitable value for $m_{\max}$, the maximum number of iterations permitted.

Step 2. If $\mathcal{F}(m)$ is a singleton, i.e., $\mathcal{F}(m) \in \mathcal{S}_0$, then set $M = 1$ and $\mathcal{Y}_1 = \mathcal{F}(m)$. Otherwise, partition $\mathcal{F}(m)$ into M sub-regions, called $\mathcal{Y}_1, \mathcal{Y}_2, \ldots, \mathcal{Y}_M$, where M can depend on the contents of the region $\mathcal{F}(m)$ but not on the iteration number m. Note that these sub-regions are disjoint subsets and that their union must equal $\mathcal{F}(m)$.

Step 3. If $\mathcal{F}(m) = \mathcal{S}$, i.e., the promising region is the entire feasible region, then set $K = M$. Otherwise, aggregate (combine) the region surrounding $\mathcal{F}(m)$, i.e., $\mathcal{S} \backslash \mathcal{F}(m)$, into one region, call that region $\mathcal{Y}_{M+1}$, and set $K = M + 1$.

Step 4. Use a random sampling strategy to select $L(\mathcal{Y}_i)$ independent solutions (singleton sets) from each sub-region $\mathcal{Y}_i$ for $i = 1, 2, \ldots, K$, where $L(\mathcal{Y}_i)$ is a randomly generated positive integer for each i. Simulate the objective function at each of these solutions. For each sub-region, find the minimum (maximum in case of maximization) objective function value. Set the minimum objective function value for the ith sub-region to equal $\phi(\mathcal{Y}_i)$ for $i = 1, 2, \ldots, K$. Now select the sub-region that contains the best objective function as follows:

$$i_* \in \underset{i \in \{1,2,\ldots,K\}}{\arg\min} \ \phi(\mathcal{Y}_i),$$

where any ties that occur are broken randomly. Then, $\mathcal{Y}_{i_*}$ contains the best objective function value.

Step 5. If $\mathcal{F}(m) = \mathcal{S}$, set $\mathcal{F}(m+1) = \mathcal{Y}_{i_*}$; else set

$$\mathcal{F}(m+1) = \begin{cases} \mathcal{Y}_{i_*} & \text{if } i_* < K \\ \mathcal{S} & \text{otherwise} \end{cases} .$$

Step 6. If $\mathcal{F}(m+1)$ is a singleton set, then increment its selection counter by 1, i.e., $V(\mathcal{F}(m+1)) \leftarrow V(\mathcal{F}(m+1)) + 1$. If $V(\mathcal{F}(m+1)) > V(\mathcal{X}^*)$, set $\mathcal{X}^* \leftarrow \mathcal{F}(m+1)$.

Step 7. Increment m by 1. If $m < m_{\max}$, return to Step 2; otherwise return $\mathcal{X}_*$ as the optimal solution and STOP.

Remark 1. The value of $m_{\max}$ has to be sufficiently large such that the algorithm repeatedly visits at least one singleton set. In the end, the singleton set with the highest value for the selection counter is returned as the optimal solution, because asymptotically, the algorithm is guaranteed to visit the optimal solution infinitely many times.
Remark 2. In Step 5, the algorithm *retracts* (referred to as "backtracks" in the original work, but since we have used backtracking to mean moving to a worse solution in the context of BAS, we prefer using the word "retracts") to the entire feasible region if the best subregion is identified to be the surrounding region $\mathcal{S} \backslash \mathcal{F}(m)$. In another version of this algorithm from [274] that we do not consider here, the algorithm retracts to a "super-region" of the most feasible region. The super-region could be a superset of the surrounding region.
Remark 3. In the algorithm, K denotes the number of regions in which sampling is performed. The algorithm is an SAS technique primarily because it randomly samples a number of points in each region for function evaluation. This is a unique feature of this algorithm. The method of partitioning, like neighbor selection in simulated annealing and BAS, is left to the user. For the algorithm to be effective, one must use a practically efficient strategy for partitioning.
Remark 4. In practice, it is not necessary to store the selection counter, $V(.)$, for every singleton set. It is sufficient to store it only for the singleton sets visited by the algorithm. This can be accomplished using a dynamic memory structure in the computer program.

We now present some examples to illustrate Steps 3 and 4 in the algorithm.

Example 1. Consider the scenario where the promising region, $\mathcal{F}(m)$, is neither a singleton and nor is it the entire feasible region $\mathcal{S}$. Assume that $M = 3$ in Step 2, and hence we partition the promising region into three sub-regions, $\mathcal{Y}_1$, $\mathcal{Y}_2$, and $\mathcal{Y}_3$, such that $\mathcal{Y}_1 \cup \mathcal{Y}_2 \cup \mathcal{Y}_3 = \mathcal{F}(m)$. Then, the surrounding region, i.e., $\mathcal{S} \backslash \mathcal{F}(m)$, is defined to be $\mathcal{Y}_4$. Here, $K = M + 1 = 4$. Now assume that we randomly generate: $L(\mathcal{Y}_1) = 4$, $L(\mathcal{Y}_2) = 10$, $L(\mathcal{Y}_3) = 2$, and $L(\mathcal{Y}_4) = 5$. Then, in the ith sub-region, we sample $L(\mathcal{Y}_i)$ number of solutions, and then we set the minimum objective function for the ith sub-region to be $\phi(\mathcal{Y}_i)$. Now, if

$$2 = \arg\min_{i \in \{1,2,\ldots,4\}} \phi(\mathcal{Y}_i),$$

then $\mathcal{Y}_2$ is likely to contain the best solution. Hence, $\mathcal{F}(m+1) = \mathcal{Y}_2$.

Example 2. Suppose in the above example it turns out that

$$4 = \arg\min_{i \in \{1,2,\ldots,4\}} \phi(\mathcal{Y}_i),$$

then the algorithm retracts to the entire feasible region in the next iteration, i.e., $\mathcal{F}(m+1) = \mathcal{S}$, and essentially starts all over again.

Example 3. Assume that the promising region in the mth iteration is a singleton set, $\mathcal{T}$. Then, $M = 1$ and $K = M + 1 = 2$. Then, $\mathcal{Y}_1 = \mathcal{T}$, and the surrounding region, i.e., $\mathcal{S} \backslash \mathcal{T}$, equals $\mathcal{Y}_2$.

Example 4. Assume that the promising region in the mth iteration is the entire feasible region $\mathcal{S}$. Let $M = 2$. Then, we construct a total of $K = M = 2$ sub-regions, such that $\mathcal{Y}_1 \cup \mathcal{Y}_2 = \mathcal{S}$.

4. Concluding Remarks

Our discussion in this chapter was restricted by design to model-free search techniques. Our discussion for the continuous case was limited to finite differences, simultaneous perturbation, and the downhill simplex. In discrete optimization, we covered two meta-heuristics, namely the genetic algorithm and tabu search, and five SAS techniques, namely simulated annealing, BAS, LAST, the stochastic ruler, and nested partitions. A number of other techniques that we were unable to cover include meta-heuristics, such as scatter search [105], ant colony optimization [80], and particle swarm optimization [163], and SAS techniques, such as GRASP [84], MRAS [146], and COMPASS [142]. MRAS is a recent development that needs special mention because MRAS generates solutions from an “intermediate probabilistic model” [146] on the solution space, which is updated iteratively after each function evaluation and may lead to an intelligent search like in the case of LAST.

It is very likely that the field of model-free static simulation optimization will expand in the future because of its ability to attack problems that cannot be solved with analytical, model-based methods. Use of multiple meta-heuristics/SAS techniques on the same problem is not uncommon because it is often difficult to rely on any one algorithm, since one algorithm may behave well for some instances of a given problem while another may perform well in other instances. The OPTQUEST package [103] (an add-on feature in ARENA) utilizes a combination of scatter search and other techniques like RSM for simulation optimization. A recent interesting paper [207] presents an "interacting particle" algorithm in which the notion of temperature is used within a genetic algorithm, and the temperature itself is optimized, within a control optimization setting, to obtain desirable convergence properties.

Bibliographic Remarks. Some good references for review material on simulation optimization are Andradóttir [8], Fu [90, 89], Carson and Maria [58], and Kleijnen [171, 172]. The material presented in this chapter comes from a large number of sources. Due to our focus on model-free techniques and other reasons, we are not able to discuss a number of important related works in simulation optimization: sample path optimization [232, 77], ordinal optimization [138], stochastic comparisons [107], perturbation analysis [137], weak derivative estimation [226], the score function method [253], sensitivity analysis [10], the frequency domain method for estimating derivatives [152], and retrospective approximation [222].

Three noteworthy works that we have not been able to cover include MRAS [146, 62], COMPASS [142], and BEESE [240]. Both MRAS and COMPASS can be used in constrained parametric optimization, which is a topic beyond the scope of this text. However, the reader is referred to these works and references therein for material on constrained optimization.

For a detailed account on simultaneous perturbation [280], which is a remarkable development, the reader is referred to the text of Spall [281]. Simultaneous perturbation has been extended to discrete parameter optimization [98] and constrained optimization [311]. A significant body of work from Bhatnagar and his colleagues, which we are not able to cover, uses simultaneous perturbation and other schemes on multiple time scales for simulation optimization [37, 39, 40, 41]. We also could not discuss continuous optimization methods based on SAS, which have been covered extensively in [333, 334].

For a general discussion on meta-heuristics, a nice text is Pham and Karaboga [227]. A unified and comprehensive treatment of SAS can be found in the text by Zabinsky [333]. Other texts that cover SAS and meta-heuristics include Spall [281] and Gendreau and Potvin [97].

The genetic algorithm originated from the work of Holland [140], while tabu search was conceived by Glover [100]. Simulated annealing has its origins in the work of Metropolis et al. [203] in the 1950s, but it was only in the 1980s that it was used as an optimization method in Kirkpatrick et al. [169]. A very large number of papers have resulted from this work, e.g., [88, 5, 96]. An extensive coverage of simulated annealing can be found in [304, 333, 85]. LAST originated from the work of Thathachar and Sastry [298]. The method was presented as an optimization tool in [262, 297]. BAS was first developed by Kristinsdottir et al. [182]; see also [333]. The original version of the stochastic ruler is due to Yan and Mukai [329]; the modified version presented here is due to Alrefaei and Andradóttir [6]. The nested partitions algorithm is from Shi and Olafsson [273, 274, 275].

Chapter 6

CONTROL OPTIMIZATION WITH STOCHASTIC DYNAMIC PROGRAMMING

1. Chapter Overview

This chapter focuses on a problem of control optimization, in particular the Markov decision problem (or process). Our discussions will be at a very elementary level, and we will not attempt to prove any theorems. The central aim of this chapter is to introduce the reader to *classical* dynamic programming in the context of solving Markov decision problems. In the next chapter, the same ideas will be presented in the context of *simulation-based* dynamic programming. The main concepts presented in this chapter are (1) Markov chains, (2) Markov decision problems, (3) semi-Markov decision problems, and (4) classical dynamic programming methods.

2. Stochastic Processes

We begin with a discussion on *stochastic processes.* A stochastic (or random) process, roughly speaking, is an entity that has a property which changes **randomly** with time. We refer to this changing property as the **state** of the stochastic process. A stochastic process is usually associated with a stochastic *system.* Read Chap. 2 for a definition of a stochastic system. The concept of a stochastic process is best understood with an example.

Consider a queue of persons that forms in a bank. Let us assume that there is a single server (teller) serving the queue. See Fig. 6.1. The queuing system is an example of a stochastic system. We need to investigate further the nature of this queuing system to identify properties, associated with the queue, that change randomly with time.

A. Gosavi, *Simulation-Based Optimization*, Operations Research/Computer Science Interfaces Series 55,
DOI 10.1007/978-1-4899-7491-4_6

Let us denote

- The number of customers in the queue at time t by $X(t)$ and
- The number of busy servers at time t by $Y(t)$.

Then, clearly, $X(t)$ will change its value from time to time and so will $Y(t)$. By its definition, $Y(t)$ will equal 1 when the teller is busy serving customers, and will equal 0 when it is idle.

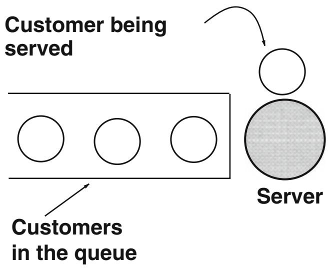

Figure 6.1. A single-server queue

Now if the state of the system is recorded after *unit time*, $X(t)$ could take on values such as: $3, 3, 4, 5, 4, 4, 3 \ldots$ The set $\{X(t)|t = 1, 2, \cdots, \infty\}$, then, defines *a* stochastic process. Mathematically, the *sequence* of values that $X(t)$ assumes in this example is a stochastic process.

Similarly, $\{Y(t)|t = 1, 2, \cdots, \infty\}$ denotes another stochastic process underlying the *same* queuing system. For example, $Y(t)$ could take on values such as $0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, \ldots$

It should be clear now that more than one stochastic process may be associated with any given stochastic system. The stochastic processes X and Y differ in their definition of the system **state**. For X, the state is the number of customers in the queue and for Y, the state is the number of busy servers.

An analyst selects the stochastic process that is of interest to him/her. E.g., an analyst interested in studying the *utilization* of the server (i.e., proportion of time the server is busy) will choose Y, while the analyst interested in studying the *length of the queue* will choose X. See Fig. 6.2 for a pictorial explanation of the word "state."

In general, choosing the appropriate definition of the state of a system is a part of "modeling." The state must be defined in a manner suitable for the optimization problem under consideration. To understand this better, consider the following definition of state. Let $Z(t)$ denote the total number of persons in the queue with *black hair*. Now,

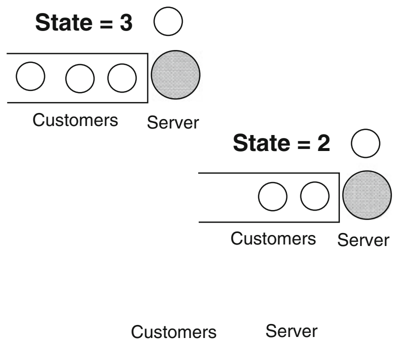

Figure 6.2. A queue in two different states: The "state" is defined by the number in the queue

although $Z(t)$ is a mathematically perfect example of a stochastic process, this definition may contain very little information of use, when it comes to controlling systems in a cost-optimal manner!

We have defined the state of a stochastic process. Now, it is time to closely examine an important stochastic process, namely, the Markov process.

3. Markov, Semi-Markov, and Decision Processes

A Markov process is of special interest to us because of its widespread use in studying real-life systems. In this section, we will study some of its salient features.

A stochastic process, usually, visits more than one state. We will assume throughout this book that the set of states visited by the stochastic process is a finite set denoted by $\mathcal{S}$.

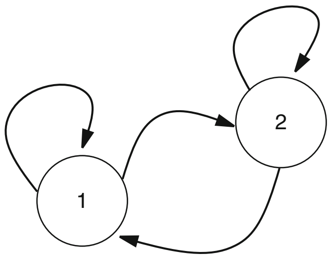

Figure 6.3. Schematic of a two-state Markov chain, where *circles* denote states

An important property of a Markov process is that it *jumps* regularly. In fact, it jumps after **unit time**. (Some authors do not use the "unit time" convention, and we will discuss this matter in detail later.) Hence, after unit time, the system either switches (moves) to a new state or else the system returns to the current state. We will refer to this phenomenon as a *state transition.*

To understand this phenomenon better, consider Fig. 6.3. The figure shows two states, which are denoted by circles, numbered 1 and 2. The arrows show the possible ways of transiting. This system has two states: 1 and 2. Assuming that we first observe the system when it is in state 1, it may for instance follow the trajectory given by: $1, 1, 2, 1, 1, 1, 2, 2, 1, 2, \ldots$

A state transition in a Markov process is usually a probabilistic, i.e., random, affair. Consider the Markov process in Fig. 6.3. Let us further assume that in its first visit to state 1, from state 1 the system jumped to state 2. In its next visit to state 1, the system may not jump to state 2 again; it may jump back to state 1. This should clarify that the transitions in a Markov chain are "random" affairs.

We now need to discuss our convention regarding the time needed for one jump (transition). In a Markov process, how much time is spent in one transition is really irrelevant to its analysis. As such, even if the time is not always unity, or even if it is not a constant, we **assume** it to be unity for our analysis. If the time spent in the transition becomes an integral part of how the Markov chain is analyzed, then the Markov process is not an appropriate model. In that case, the semi-Markov process becomes more appropriate, as we will see below.

When we study real-life systems using Markov processes, it usually becomes necessary to define a performance metric for the real-life system. It is in this context that one has to be careful with how the unit time convention is interpreted. A common example of a performance metric is: average reward *per unit time.* In the case of a Markov process, the phrase "per unit time" in the definition of average reward actually means "per jump" or "per transition." (In the so-called semi-Markov process that we will study later, the two phrases have different meanings.)

Another important property of the Markov process needs to be studied here. In a Markov process, the probability that the process jumps from a state i to a state j does **not** depend on the states visited by the system *before* coming to i. This is called the **memoryless** property. This property distinguishes a Markov process from other stochastic processes, and as such it needs to be understood clearly. Because of the memoryless property, one can associate a probability with a transition from a state i to a state j, that is,

$$i \longrightarrow j.$$

We denote the probability of this transition by $P(i,j)$. This idea is best explained with an example.

Consider a Markov chain with three states, numbered 1, 2, and 3. The system starts in state 1 and traces the following trajectory:

$$1,3,2,1,1,1,2,1,3,1,1,2,\ldots$$

Assume that: $P(3,1)=0.2$ and $P(3,2)=0.8$. When the system visits 3 for the first time in the above, it jumps to 2. Now, the probability of jumping to 2 is 0.8, and that of jumping to 1 is 0.2. When the system *revisits* 3, the probability of jumping to 2 will **remain at** 0.8, and that of jumping to 1 at 0.2. Whenever the system comes to 3, its probability of jumping to 2 will always be 0.8 and that of jumping to 1 be 0.2. In other words, when the system comes to a state i, the state to which it jumps depends *only* on the transition probabilities: $P(i,1), P(i,2)$ and $P(i,3)$. These probabilities are not affected by the sequence of states visited before coming to i. Thus, when it comes to jumping to a new state, the process does not "remember" what states it has had to go through in the past. The state to which it jumps depends only on the current state (say i) and on the probabilities of jumping from that state to other states, i.e., $P(i,1), P(i,2)$ and $P(i,3)$. In general, when the system is ready to leave state i, the next state j depends only on $P(i,j)$. Furthermore, $P(i,j)$ is completely independent of where the system has been before coming to i.

We now give an example of a non-Markovian process. Assume that a process has three states, numbered 1, 2, and 3. $X(t)$, as before, denotes the system state at time t. Assume that the law governing this process is given by:

$$\mathsf{P}\{X(t+1)=j|X(t)=i, X(t-1)=l\} = f(i,l,j), \qquad (6.1)$$

where $f(i,l,j)$ denotes a probability that depends on i, l, and j. This implies that if the process is in state i at time t (notice that $X(t)=i$ in the equation above is supposed to mean exactly this), and if it was in state l at time $(t-1)$, then the probability that the next state (i.e., the state visited at time $(t+1)$) will be j is a function of i, l, and j. In other words, at any point of time, the past (i.e., $X(t-1)$) **will** affect its future course.

Thus the process described above is not a Markov process. In this process, the state of the process at time $(t-1)$ *does* affect the probabilities of going to other states at time $(t+1)$. The path of this stochastic process is thus dependent on its past, not the entire past, but some of its past.

The Markov process, on the other hand, is governed by the following law:

$$\mathsf{P}\{X(t+1) = j | X(t) = i\} = f(i, j), \tag{6.2}$$

where $f(i, j)$ is the probability that the next state is j given that the current state is i. Also $f(i, j)$ is a constant for given values of i and j.

Carefully note the difference between Eqs. (6.2) and (6.1). Where the process resides *one step* before its current state has no influence on a Markov process. It should be obvious that in the Markov process, the transition probability (probability of going to one state to another in the stochastic process **in one step**) depends on two quantities: the present state (i) and the next state (j). In a non-Markovian process, such as the one defined by Eq. (6.1), the transition probability depended on the current state (i), the next state (j), **and** the previous state (l). An implication is that even if both the processes have the same number of states, we will have to deal with additional probabilities in the two-step stochastic process.

The quantity $f(i, j)$ is an element of a two-dimensional matrix. Note that $f(i, j)$ is actually $P(i, j)$, the one-step transition probability of jumping from i to j, which we have defined earlier.

All the transition probabilities of a Markov process can be conveniently stored in a matrix. This matrix is called the one-step transition probability matrix or simply the **transition probability matrix**, usually abbreviated as TPM. An example of a TPM with three states is:

$$\mathbf{P} = \begin{bmatrix} 0.7 & 0.2 & 0.1 \\ 0.4 & 0.2 & 0.4 \\ 0.6 & 0.1 & 0.3 \end{bmatrix}. \tag{6.3}$$

$P(i, j)$ here denotes the (i, j)th element of the matrix, $\mathbf{P}$, i.e., the element in the ith row and the jth column of $\mathbf{P}$. In other words, $P(i, j)$ denotes the one-step transition probability of jumping from state i to state j. Thus, for example, $P(3, 1)$, which is 0.6 above, denotes the one-step transition probability of going from state 3 to state 1.

We will also assume that a finite amount of time is taken in any transition and that *no time is actually spent in a state.* This is one convention (there are others), and we will stick to it in this book. Also, note that by our convention, the time spent in a transition is unity (1).

In summary, a Markov process possesses three important properties: (1) the jumpy property, (2) the memoryless property, and (3) the unit time property (by our convention).

3.1. Markov Chains

A Markov **chain** can be thought of as an entity that accompanies a stochastic process. Examples of stochastic processes accompanied by Markov chains that we will consider are the Markov process and the semi-Markov process.

We associate a unique TPM with a given Markov chain. It should be kept in mind that the Markov chain (not the Markov process) contains no information about how much time is spent in a given transition.

Example 1. Consider Fig. 6.4. It shows a Markov chain with two states, shown by circles, numbered 1 and 2. The arrow indicates a possible transition, and the number on the arrow denotes the probability of that transition. The figure depicts the following facts. If the process is in state 1, it goes to state 2 with a probability of 0.3, and with a probability of 0.7, it stays in the same state (i.e., 1). Also, if the process is in state 2, it goes to state 1 with a probability of 0.4 and stays in the same state (i.e., 2) with a probability of 0.6. The TPM of the Markov chain in the figure is therefore

$$\mathbf{P} = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix}.$$

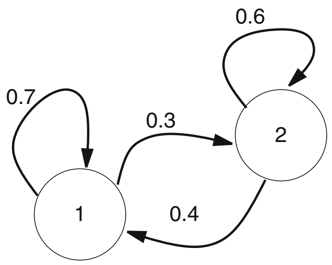

Figure 6.4. Schematic of a two-state Markov chain, where *circles* denote states, *arrows* depict possible transitions, and the numbers on the arrows denote the probabilities of those transitions

Figures 6.5 and 6.6 show some more examples of Markov chains with three and four states respectively. In this book, we will consider Markov chains with a finite number of states.

Estimating the values of the elements of the TPM is often quite difficult. This is because, in many real-life systems, the TPM is very large, and evaluating any given element in the TPM requires the setting up of complicated expressions, which may involve multiple integrals. In subsequent chapters, this issue will be discussed in depth.

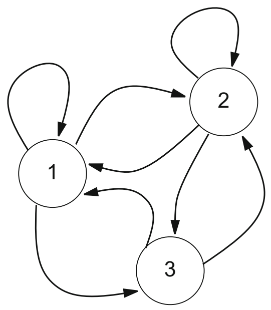

Figure 6.5. Schematic of a Markov chain with three states

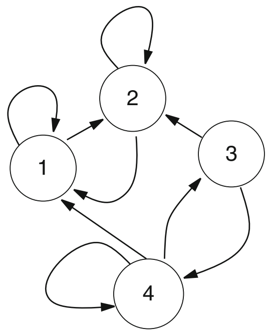

Figure 6.6. Schematic of a Markov chain with four states

n-step transition probabilities. The n-step transition probability of going from state i to state j is defined as the probability of starting at state i and being in state j after n steps (or jumps/transitions). From the one-step transition probabilities, it is possible to construct the two-step transition probabilities, the three-step transition probabilities, and the n-step transition probabilities, in general. The n-step transition probabilities are very often of great importance to the analyst.

The so-called Chapman-Kolmogorov theorem helps us find these probabilities. The theorem states that the n-step transition probabilities can be obtained by raising the one-step transition probability matrix to the nth power. We next state the theorem without proof.

THEOREM 6.1 *If* $\mathbf{P}$ *denotes the one-step transition probability matrix of a Markov chain and* $\mathbf{S} = \mathbf{P}^n$*, where* $\mathbf{P}^n$ *denotes the matrix* $\mathbf{P}$ *raised to the nth power, then* $S(i, j)$ *denotes the n-step transition probability of going from state i to state j.*

Basically, the theorem states says that the nth power of a TPM is also a transition probability matrix, whose elements are the n-step transition probabilities. Let us illustrate the meaning of this result with an example. Consider the TPM given by:

$$\mathbf{P} = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix}.$$

Now,

$$\mathbf{P}^2 = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix} \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix} = \begin{bmatrix} 0.61 & 0.39 \\ 0.52 & 0.48 \end{bmatrix}.$$

Here the value of $P^2(1,1)$ is 0.61. The theorem says that $P^2(1,1)$ *equals* the *two-step* transition probability of going from 1 to 1. Let us verify this from the basic principles of probability theory. Let C_{x-y-z} denote the probability of going from state x to state z in two transitions with y as the intermediate state. Consider the event of going from state 1 to state 1 in two steps. Then, clearly, the probability of this event should equal:

$$C_{1-1-1} + C_{1-2-1}.$$

From the values of $\mathbf{P}$, $C_{1-1-1} = (0.7)(0.7)$ and $C_{1-2-1} = (0.3)(0.4)$, and therefore the required probability should equal:

$$(0.7)(0.7) + (0.3)(0.4) = 0.61,$$

which is equal to $P^2(1,1)$. The verification is thus complete.

3.1.1 Regular Markov Chains

A regular Markov chain is one whose TPM satisfies the following property: *There exists a finite positive value for n, call it n_*, such that for all $n \geq n_*$, and all i and j*:

$$P^{n_*}(i,j) > 0.$$

In other words, by raising the TPM of a regular Markov chain to some positive power, one obtains a matrix in which each element is strictly greater than 0. An example of a regular Markov chain is:

$$\mathbf{P} = \begin{bmatrix} 0.7 & 0.0 & 0.3 \\ 0.4 & 0.6 & 0.0 \\ 0.2 & 0.7 & 0.1 \end{bmatrix}.$$

It is not hard to verify that P can be raised to a suitable power to obtain a matrix in which each element is strictly greater than 0. An example of a Markov chain that is not regular is:

$$\mathbf{P} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

This is not regular because:

$$\mathbf{P}^n = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \text{ for odd } n \text{ and } \mathbf{P}^n = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \text{ for even } n.$$

3.1.2 Limiting Probabilities

If one raises the TPM of a *regular* Markov chain to higher powers, the elements in any given column start *converging* to (that is, approaching) the same number. For example, consider **P** of the previous section, raised to the 8th power:

$$\mathbf{P}^8 = \begin{bmatrix} 0.5715 & 0.4285 \\ 0.5714 & 0.4286 \end{bmatrix}.$$

The elements of a given *column* in $\mathbf{P}^n$ start approaching each other, as we increase the power n, and notice that by $n = 8$, the elements are very close to each other. It can be proved that for a regular Markov chain as the power n tends to infinity, for every j, $P^n(i, j)$ starts converging to a unique finite number. In other words, for every value of j,

$$\lim_{n\to\infty} P^n(i, j) \text{ exists and is unique.}$$

In the example under consideration, the limit appears to be 0.57 for state 1 and 0.43 for state 2. We will denote the limit for state j by $\Pi(j)$. Mathematically,

$$\Pi(j) \equiv \lim_{n\to\infty} P^n(i, j).$$

The quantity $\Pi(j)$ will also be referred to as the **limiting** or **steady-state** or **invariant** probability of the state j.

Now, $\Pi(j)$, it must be understood, is the **long-run** probability of entering the state j **from any given state**. For instance, in the example given above, regardless of which state the Markov chain is in, the long-run (that is, when $n \to \infty$) probability of entering state 1 is 0.57. Similarly, the long-run probability of entering state 2 is 0.43. From this, we can make an important inference:

> Since the transitions are assumed to take unit time, 57 % of the time will be spent by the process in transitions to state 1 and 43 % of the time in transitions to state 2.

For the Markov process, the time taken in any transition is equal, and hence the limiting probability of a state also denotes the proportion of time spent in transitions to that particular state.

We will now show how we can obtain the limiting probabilities from the TPM without raising the TPM to large powers. The following important result provides a very convenient way for obtaining the limiting probabilities.

THEOREM 6.2 *Let $\Pi(i)$ denote the limiting probability of state i, and let $\mathcal{S}$ denote the set of states in the Markov chain. Then the limiting probabilities for all the states in the Markov chain can be obtained from the transition probabilities by solving the following set of linear equations:*

$$\sum_{i=1}^{|\mathcal{S}|} \Pi(i)P(i,j) = \Pi(j), \text{ for every } j \in \mathcal{S} \tag{6.4}$$

$$\text{and } \sum_{j=1}^{|\mathcal{S}|} \Pi(j) = 1, \tag{6.5}$$

where $|\mathcal{S}|$ denotes the number of elements in the set $\mathcal{S}$.

Equations (6.4) and (6.5) are often collectively called the *invariance* equation, since they help us determine the invariant (limiting) probabilities. Equation (6.4) is often expressed in the matrix form as:

$$[\Pi(1), \Pi(2), \ldots, \Pi(|\mathcal{S}|)]\mathbf{P} = [\Pi(1), \Pi(2), \ldots, \Pi(|\mathcal{S}|)],$$

or in the following abbreviated form: $\vec{\Pi}\mathbf{P} = \vec{\Pi}$, where $\vec{\Pi}$ is a row vector of the limiting probabilities.

Although we do not present its proof, the above is an important result from many standpoints. If you use the equations above to find the limiting probabilities of a Markov chain, you will notice that there is one *extra* equation in the linear system of equations defined by the theorem. You can eliminate *any* one equation from the system defined by (6.4), and then solve the remaining equations to obtain a unique solution. We demonstrate this idea with the TPM given in (6.3).

From equations defined by (6.4), we have:

$$\text{For } j = 1: \quad 0.7\Pi(1) + 0.4\Pi(2) + 0.6\Pi(3) = \Pi(1). \tag{6.6}$$

$$\text{For } j = 2: \quad 0.2\Pi(1) + 0.2\Pi(2) + 0.1\Pi(3) = \Pi(2). \tag{6.7}$$

$$\text{For } j = 3: \quad 0.1\Pi(1) + 0.4\Pi(2) + 0.3\Pi(3) = \Pi(3). \tag{6.8}$$

With some transposition, we can re-write these equations as:

$$-0.3\Pi(1) + 0.4\Pi(2) + 0.6\Pi(3) = 0. \tag{6.9}$$

$$0.2\Pi(1) - 0.8\Pi(2) + 0.1\Pi(3) = 0. \tag{6.10}$$

$$0.1\Pi(1) + 0.4\Pi(2) - 0.7\Pi(3) = 0. \tag{6.11}$$

Now from Eq. (6.5), we have:

$$\Pi(1) + \Pi(2) + \Pi(3) = 1. \tag{6.12}$$

Thus we have four Equations: (6.9)–(6.12) and three unknowns: $\Pi(1), \Pi(2)$, and $\Pi(3)$. Notice that the system defined by the three Eqs. (6.9)–(6.11) actually contains only two independent equations because any one can be obtained from the knowledge of the other two. Hence we select *any* two equations from this set. The two along with Eq. (6.12) can be solved to find the unknowns. The values are: $\Pi(1) = 0.6265, \Pi(2) = 0.1807$, and $\Pi(3) = 0.1928$.

Remark. The history-independent property of the Markov chain is somewhat misleading. You can incorporate as much history as you want into the state space, by augmenting the state space with historical information, to convert a history-dependent process into a Markov process. Obviously, however, this comes with a downside in that the size of the state space in the synthesized Markov process is much larger than that in the history-dependent process. Usually, trying to incorporate all the relevant history to transform the stochastic process into a Markov process can produce an unwieldy stochastic process that is difficult to analyze.

3.1.3 Ergodic Markov Chains

A state in a Markov chain is said to be **recurrent** if it is visited repeatedly (again and again). In other words, if one views a Markov chain for an infinitely long period of time, one will see that a recurrent state is visited infinitely many times. A **transient** state is one which is visited only a finite number of times in such an "infinite viewing."

An example of a transient state is one to which the system does not come back from any recurrent state in one transition. In Fig. 6.7, 1 is a transient state because once the system enters 2 or 3, it cannot come back to 1.

There may be more than one transient state in a Markov chain. In Fig. 6.8, 1a and 1b are transient states. If the system starts in any one of these two states, it can visit both but once it goes to 2, it can never come back to 1a or 1b. A state that is not transient is called a recurrent state. Thus 2 and 3 are recurrent states in both Figs. 6.7

and 6.8. Another type of state is the *absorbing* state. Once the system enters any absorbing state, it can never get out of that state and it remains there.

An **ergodic** Markov chain is one in which all states are recurrent and no absorbing states are present. Ergodic chains are also called irreducible chains. All regular Markov chains are ergodic, but the converse is not true. (Regular chains were defined in Sect. 3.1.1.) For instance, a chain that is not regular may be ergodic. Consider the Markov chain with the following TPM:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

This chain is not regular, but ergodic. It is ergodic because both states are visited infinitely many times in an infinite viewing.

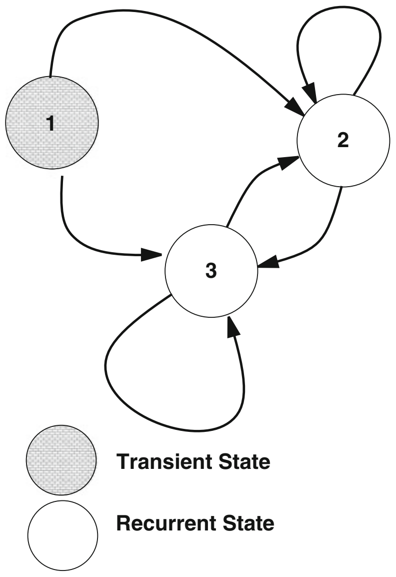

Figure 6.7. A Markov chain with one transient state

We will now discuss the semi-Markov process.

3.2. Semi-Markov Processes

A stochastic process that spends a random amount of time (which is not necessarily unity) in each transition, but is otherwise similar to a Markov process, is called a semi-Markov process. Consequently, underlying a semi-Markov process, there lurks a Markov *chain*

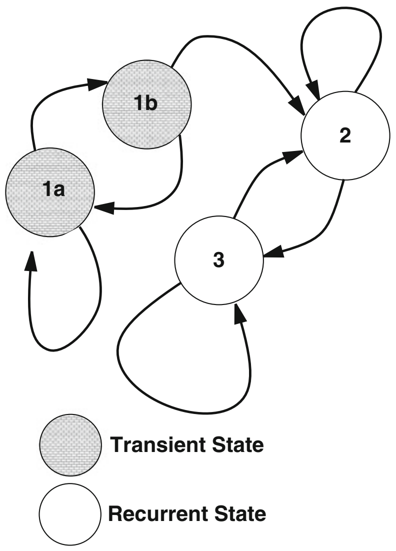

Figure 6.8. A Markov chain with two transient states

called the **embedded** Markov chain. The main difference between the semi-Markov process and the Markov process lies in the time taken in transitions.

In general, when the distributions for the transition times are arbitrary, the process goes by the name semi-Markov. If the time in every transition is an exponentially distributed random variable, the stochastic process is referred to as a *continuous time Markov process.*

Some authors refer to what we have called the continuous time Markov process as the "Markov process," and by a "Markov chain," they mean what we have referred to as the Markov process.

There is, however, a critical difference between the Markov chain underlying a Markov process and that underlying a semi-Markov process. In a semi-Markov process, the system jumps, but not necessarily after unit time, and when it jumps, it jumps to a state that is *different* than the current state. In other words, in a semi-Markov process, the system cannot jump back to the current state. (However, in a semi-Markov *decision* process, which we will discuss later, jumping back to the current state is permitted.) In a Markov process, on the other hand, the system *can* return to the current state after one jump.

If the time spent in the transitions is a deterministic quantity, the semi-Markov process has a transition time matrix analogous to the TPM, e.g.,

$$\begin{bmatrix} - & 17.2 \\ 1 & - \end{bmatrix}.$$

For an example of the most general model in which some or all of the transition times are random variables from any given distributions, consider the following transition time matrix:

$$\begin{bmatrix} - & unif(5,6) \\ expo(5) & - \end{bmatrix},$$

where $unif(\min, \max)$ denotes a random number from the uniform distribution with parameters, min and max, and $expo(\mu)$ denotes the same from the exponential distribution with parameter μ.

When we analyze a semi-Markov process, we begin by analyzing the Markov chain *embedded* in it. The next step usually is to analyze the time spent in each transition. As we will see later, the semi-Markov process is more powerful than the Markov process in modeling real-life systems, although very often its analysis can prove to be more complicated.

3.3. Markov Decision Problems

We will now discuss the topic that forms the central point of control optimization in this book. Thus far, we have considered Markov chains in which the transition from one state to another is governed by only one transition law, which is contained in the elements of the TPM. Such Markov chains are called *uncontrolled* Markov chains, essentially because in such chains there is no external agency that can control the path taken by the stochastic process.

We also have systems that can be run with different control mechanisms, where each control mechanism has its own TPM. In other words, the routes dictated by the control mechanisms are not the same. The control mechanism specifies the "action" to be selected in each state. When we are faced with the decision of choosing from *more than one* control mechanism, we have what is called a **Markov decision problem**, which is often abbreviated as an **MDP**. The MDP is also commonly called the Markov decision *process.*

The MDP is a problem of control optimization. In other words, it is the problem of finding the optimal **action** to be selected in each state. Many real-world problems can be set up as MDPs, and before we discuss any details of the MDP framework, let us study a simple example of an MDP.

Example. Consider a queuing system, such as the one you see in a supermarket, with a maximum of three counters (servers). You have probably noticed that when queues become long, more counters (servers) are opened. The decision-making problem we will consider here is to find the number of servers that should be open at any given time.

The people who function as servers also have other jobs to perform, and hence it does not make business sense to have them wait on the counters when there are no customers at the counters. At the same time, if very long queues build up but more counters are not opened when there is capacity, customers do not feel very happy about it, and may actually go elsewhere the next time. Hence in this situation, one seeks an optimal strategy (i.e., a control mechanism) to control the system. Next, we will discuss the idea of control mechanisms or policies with some examples.

Consider a system which has a maximum of three counters. Let us assume that the state of this system is defined by the number of people waiting for service. Let us further assume that associated with this state definition, a Markov process exists. (See Fig. 6.9 for a picture of the underlying Markov chain.) Thus when the system enters a new state of the Markov chain, one out of the following three **actions** can be selected:

{Open 1 counter, Open 2 counters, and Open 3 counters.}

One possible control mechanism (or policy) in this situation would look like this:

State = Number waiting for service	Action
0	Open 1 counter
1	Open 1 counter
2	Open 1 counter
3	Open 1 counter
4	Open 2 counters
5	Open 2 counters
6	Open 2 counters
7	Open 3 counters
8	Open 3 counters
.	Open 3 counters
.	Open 3 counters
.	Open 3 counters

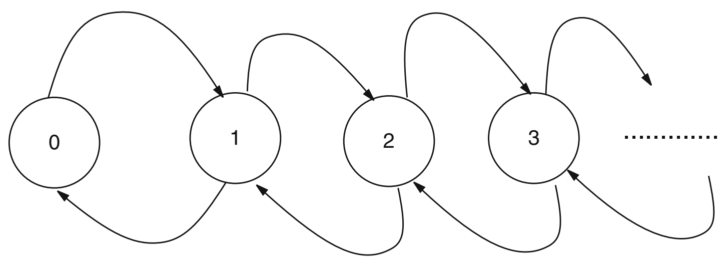

Figure 6.9. A Markov chain underlying a simple single-server queue

Another possible control mechanism could be:

State = Number waiting for service	Action
0	Open 1 counter
1	Open 1 counter
2	Open 1 counter
3	Open 1 counter
4	Open 1 counter
5	Open 1 counter
6	Open 2 counters
7	Open 2 counters
8	Open 3 counters
.	Open 3 counters
.	Open 3 counters
.	Open 3 counters

Note that the two control mechanisms are different. In the first, two counters are opened in state 4, while in the second, the same is done in state 6. From these two examples, it should be clear that there are, in fact, several different control mechanisms that can be used, and the effect on the system—in terms of the net profits generated—may differ with the control mechanism used. This because:

- A control mechanism that allows big queues to build up may lead to a cost reduction in some sense, since fewer employees may be necessary to run the system. But it may also lead to reduced profits, because in the future customers may choose to go elsewhere where queues are shorter.

- On the other hand, a control mechanism which is over-designed with a large number of servers (where customers hardly ever have to wait) may be expensive to maintain, because of the costs incurred in hiring a large number of servers. Eventually the high costs of running this system will be transmitted to the customers through higher-priced products. The latter is also likely to drive customers away.

It naturally makes business sense to use the control mechanism that produces the greatest net profits. Formulation of this problem as a Markov decision problem will help us identify the best control mechanism.

From our discussion above, we can conclude that each control mechanism is likely to have unique costs and profits. Further, since the system considered above is stochastic, associated with each control mechanism, a distinctive pattern of behavior is likely to emerge. Finally, the problem is one of finding the right control mechanism. The Markov decision framework is a sophisticated operations research model designed to solve this problem. We now provide details.

3.3.1 Elements of an MDP

The Markov decision framework is designed to solve the so-called Markov decision problem (MDP). The framework is made up of five important elements. They are: (1) A decision maker, (2) policies, (3) transition probability matrices, (4) transition reward matrices, and (5) a performance metric (objective function).

Decision maker. The decision maker is an entity that *selects* the control mechanism. It is also called the *agent* or *controller*.

Policies. The control mechanism is usually referred to as a **policy**. A policy for an MDP with n states is an n-tuple. Each element of this n-tuple specifies the action to be selected in the state associated with that element. For example, consider a 2-state MDP in which two actions are allowed in each state. An example of a policy for this MDP is: $(2, 1)$. This means that by adhering to this policy, the following would occur: In state 1, action 2 would be selected, and in state 2, action 1 would be selected. Thus in general, if $\hat{\mu}$ denotes a policy, the ith element of $\hat{\mu}$, that is, $\mu(i)$, denotes the action selected in the ith state for the policy $\hat{\mu}$.

In this book, unless otherwise stated, the word "policy" will imply a **stationary, deterministic** policy. The word stationary means that the policy does *not* change with time. This implies that if a policy dictates an action a be taken in a state x, then no matter how long the system has been operating, every time the system visits state x, it is action a that will be selected.

The word deterministic implies that in any given state, we can choose **only one (1)** action (out of the multiple actions allowed). In other words, with a probability of 1, a given action is selected. We will deal with stochastic policies in the context of learning automata and actor critics in Chap. 8 for control optimization. However, in this chapter, we will primarily consider stationary, deterministic policies.

We will assume throughout this book that the set of actions allowed in each state is a finite set. The set of actions allowed in state i will be denoted by $\mathcal{A}(i)$. We will also assume the set of states in the system to be a finite set, which will be denoted by $\mathcal{S}$.

Since the number of actions allowed in each state and the number of states themselves are finite quantities, we must have a finite number of policies. For instance in an MDP with two states and two actions allowed in each state, we have $2^2 = 4$ policies, which are:

$$(1,1), \quad (1,2), \quad (2,1), \quad \text{and } (2,2).$$

An MDP, let us reiterate, revolves around finding the most *suitable* policy.

A few more words about the term "state" are in order. We may encounter systems in which decisions are not made in every state. In other words, in some states, there is only one allowable action. As such, there is no decision making involved in these states. These states are called *non-decision-making* states. A state in which one has to choose from more than one action is hence called a *decision-making* state. In this book, by "state," we refer to a *decision-making* state. When we have models in which some states are not of the decision-making kind, we will distinguish between the two by the qualifiers: decision-making and non-decision-making.

Transition probability matrices. The decision maker executes the action to be used in each state of an MDP. Associated with **each action**, we usually have a transition probability matrix (TPM). Associated with each policy, also, we have a unique TPM. The TPM for a policy can be constructed from the TPMs associated with the individual actions in the policy.

We illustrate this construction with a 2-state MDP that has 2 actions allowed in each state. Let the TPM associated with action a be denoted by $\mathbf{P}_a$. Let

$$\mathbf{P}_1 = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix}, \text{ and } \mathbf{P}_2 = \begin{bmatrix} 0.1 & 0.9 \\ 0.8 & 0.2 \end{bmatrix}.$$

Now consider a policy $\hat{\mu} = (2, 1)$. The TPM associated with this policy will contain the transition probabilities of action 2 in state 1 and the transition probabilities of action 1 in state 2. The TPM of policy $\hat{\mu}$ is thus

$$\mathbf{P}_{\hat{\mu}} = \begin{bmatrix} 0.1 & 0.9 \\ 0.4 & 0.6 \end{bmatrix}.$$

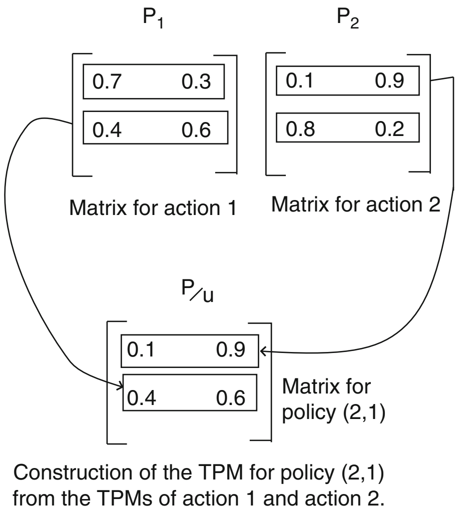

Figure 6.10. Schematic showing how the TPM of policy (2, 1) is constructed from the TPMs of action 1 and 2

See Fig. 6.10 for a pictorial demonstration of the construction process.

In general, we will use the following notation to denote a transition probability:

$$p(i, a, j).$$

This term will denote the one-step transition probability of going from state i to state j when action a is selected in state i. Now, if policy $\hat{\mu}$ is followed, then the action selected in state i will be denoted by $\mu(i)$, and as a result the transition probability of going from state i to state j will be denoted by

$$p(i, \mu(i), j).$$

Then $p(i, \mu(i), j)$ will define the element in the ith row and the jth column of the matrix $\mathbf{P}_{\hat{\mu}}$—the TPM associated with the policy $\hat{\mu}$.

In this section, we have used the phrase: "a transition probability **under the influence of an action**." The significance of this must be noted with care. In our previous discussions, we spoke of transition probabilities without reference to any action. That was because we were dealing with **uncontrolled** Markov chains—which had **only one action** in each state. Now, with the introduction of multiple actions in each state, we must be careful with the phrase "transition probability," and must specify the action along with a transition probability.

Now, as stated previously, the MDP is all about identifying the optimal policy (control mechanism). The TPM, we will see shortly, will serve an important purpose in evaluating a policy and will be essential in identifying the best policy. The other tool that we need for evaluating a policy is discussed in the next paragraph.

Transition reward matrices. With each transition in a Markov chain, we can associate a reward. (A negative value for the reward is equivalent to a cost.) We will refer to this quantity as the **immediate reward** or transition reward. The immediate reward helps us incorporate reward and cost elements into the MDP model. The immediate reward matrix, generally called the transition reward matrix (TRM), is very similar to the TPM. Recall that the (i, j)th element (the element in the ith row and jth column) of the TPM denotes the transition probability from state i to state j. Similarly, the (i, j)th element of the TRM denotes the immediate reward earned in a transition from state i to state j. Just as we have TPMs associated with individual actions and policies, we have TRMs associated with actions and policies. Let us examine some examples from a 2-state MDP, next.

Let $\mathbf{R}_a$ be the TRM associated with action a, and let:

$$\mathbf{R}_1 = \begin{bmatrix} 11 & -4 \\ -14 & 6 \end{bmatrix} \text{ and } \mathbf{R}_2 = \begin{bmatrix} 45 & 80 \\ 1 & -23 \end{bmatrix}.$$

Now consider a policy $\hat{\mu} = (2, 1)$. Like in the TPM case, the TRM associated with this policy will contain the immediate reward of action 2 in state 1 and the immediate rewards of action 1 in state 2. Thus the TRM of policy $\hat{\mu}$ can be written as

$$\mathbf{R}_{\hat{\mu}} = \begin{bmatrix} 45 & 80 \\ -14 & 6 \end{bmatrix}.$$

The TPM and the TRM of a policy together contain all the information one needs to evaluate the policy in an MDP. In terms of notation, we will denote the immediate reward, earned in going from state i to state j, under the influence of action a, by:

$$r(i, a, j).$$

When policy $\hat{\mu}$ is followed, the immediate reward earned in going from state i to state j will be denoted by:

$$r(i, \mu(i), j)$$

because $\mu(i)$ is the action that will be selected in state i when policy $\hat{\mu}$ is used.

Performance metric. To compare policies, one must define a performance metric (objective function). Naturally, the performance metric should involve reward and cost elements. To give a simple analogy, in a linear programming problem, one judges each solution on the basis of the value of the associated objective function. Any optimization problem has a performance metric, which is also called the **objective function**. In this book, for the most part, the MDP will be studied with respect to two performance metrics. They are:

1. Expected reward *per unit time* calculated over an infinitely long trajectory of system states: We will refer to this metric as the **average reward**.

2. Expected *total* discounted reward calculated over an infinitely long trajectory of system states: We will refer to this metric as the **discounted reward**.

It is the case that of the two performance metrics, average reward is easier to understand, although the average reward MDP is more difficult to analyze for its convergence properties. Hence, we will begin our discussion with the average reward performance criterion. Discounted reward will be defined later.

We first need to define the **expected immediate reward** of a state under the influence of a given action. Consider the following scenario. An action a is selected in state i. Under the influence of this action, the system can jump to three states: 1, 2, and 3 with probabilities of

$$0.2, 0.3, \text{ and } 0.5,$$

respectively. The immediate rewards earned in these three possible transitions are, respectively,

$$10, 12, \text{ and } -14.$$

Then the **expected** immediate reward that will be earned, when action a is selected in state a, will clearly be:

$$0.2(10) + 0.3(12) + 0.5(-14) = -1.4.$$

The expected immediate reward is calculated in the style shown above. (Also see Fig. 6.11). In general, we can use the following expression to calculate the expected immediate reward.

$$\bar{r}(i,a) = \sum_{j \in \mathcal{S}} p(i,a,j)r(i,a,j). \tag{6.13}$$

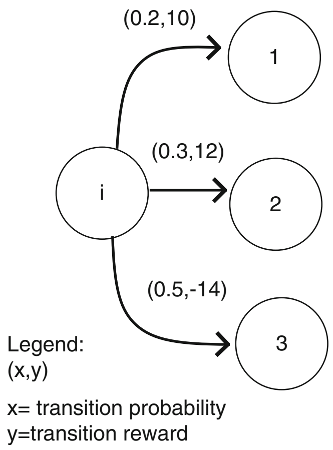

Figure 6.11. Calculation of expected immediate reward

The average reward of a given policy is the expected reward earned per unit time by running the Markov chain associated with the policy for an infinitely long period of time. It turns out that there is a very convenient way for representing the average reward of a policy, which employs the limiting probabilities associated with the policy. We will explain this with an example, and then generalize from there to obtain a generic expression for the average reward.

Consider a 3-state MDP with states numbered 1, 2, and 3. Let us assume that the system follows a fixed policy $\hat{\mu}$ and that the limiting probabilities of the three states with respect to this policy are: $\Pi_{\hat{\mu}}(1) = 0.3, \Pi_{\hat{\mu}}(2) = 0.5$, and $\Pi_{\hat{\mu}}(3) = 0.2$. Let us further assume that the expected immediate rewards earned in the three states are:

$$\bar{r}(1,\mu(1)) = 10, \bar{r}(2,\mu(2)) = 12, \text{ and } \bar{r}(3,\mu(3)) = 14.$$

Now from our discussion on limiting probabilities, we know that the limiting probability of a state denotes the proportion of time spent in transitions to that particular state in the long run. Hence if we observe the system over k transitions, $k\Pi_{\hat{\mu}}(i)$ will equal the number of transitions to state i in the long run. Now, the expected reward earned in each visit to state i under policy $\hat{\mu}$ is $\bar{r}(i,\mu(i))$. Then the total long-run expected reward earned in k transitions for this MDP can be written as:

$$k\Pi_{\hat{\mu}}(1)\bar{r}(1,\mu(1)) + k\Pi_{\hat{\mu}}(2)\bar{r}(2,\mu(2)) + k\Pi_{\hat{\mu}}(3)\bar{r}(3,\mu(3)).$$

Consequently the average reward associated with policy $\hat{\mu}$ can be written as:

$$\begin{aligned}\rho_{\hat{\mu}} &= \frac{k\Pi_{\hat{\mu}}(1)\bar{r}(1,\mu(1)) + k\Pi_{\hat{\mu}}(2)\bar{r}(2,\mu(2)) + k\Pi_{\hat{\mu}}(3)\bar{r}(3,\mu(3))}{k} \\ &= \sum_{i=1}^{3}\Pi_{\hat{\mu}}(i)\bar{r}(i,\mu(i))\end{aligned}$$

Then, in general, the average reward of a policy $\hat{\mu}$ can be written as:

$$\rho_{\hat{\mu}} = \sum_{i\in\mathcal{S}}\Pi_{\hat{\mu}}(i)\bar{r}(i,\mu(i)), \text{ where} \tag{6.14}$$

- $\Pi_{\hat{\mu}}(i)$ denotes the limiting probability of state i when the system (and hence the underlying Markov chain) is run with the policy $\hat{\mu}$
- $\mathcal{S}$ denotes the set of states visited in the system
- And $\bar{r}(i,a)$ denotes the expected immediate reward earned in the state i when action a is selected in state i

In the next section, we will discuss a simple method to solve the MDP. But before that we conclude this section by enumerating the assumptions we will make about the control optimization problem we will solve in this book.

ASSUMPTION 6.1 *The state space $\mathcal{S}$ and the action space $\mathcal{A}(i)$ for every $i \in \mathcal{S}$ is finite (although possibly quite large).*

ASSUMPTION 6.2 *The Markov chain associated with every policy in the problem is regular.*

ASSUMPTION 6.3 *The immediate reward earned in any state transition under any action is finite, i.e., for all (i,a,j), $|r(i,a,j)| < \infty$.*

3.3.2 Exhaustive Enumeration

The method that we will discuss in this section goes by the name exhaustive enumeration or exhaustive evaluation. Conceptually, this is the easiest method to understand, although in practice we can use it only on small problems. The method is based on the following idea: *Enumerate every policy that can possibly be selected, evaluate the performance metric associated with each policy, and then declare the policy that produces the best value for the performance metric to be the optimal policy.* We now explain this method with a simple example of an MDP that has just two states and two actions in each state. *This example will be used repeatedly throughout the remainder of this book.*

Example A. There are two states numbered 1 and 2 in an MDP, and two actions, which are also numbered 1 and 2, are allowed in each state. The transition probability matrices (TPM) associated with actions 1 and 2 are:

$$\mathbf{P}_1 = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix} \text{ and } \mathbf{P}_2 = \begin{bmatrix} 0.9 & 0.1 \\ 0.2 & 0.8 \end{bmatrix}.$$

The TRM for actions 1 and 2 are:

$$\mathbf{R}_1 = \begin{bmatrix} 6 & -5 \\ 7 & 12 \end{bmatrix} \text{ and } \mathbf{R}_2 = \begin{bmatrix} 10 & 17 \\ -14 & 13 \end{bmatrix}.$$

Pictorially, the MDP is represented in Fig. 6.12.

In this MDP, there are four possible policies that can be used to control the system. They are:

$$\hat{\mu}_1 = (1,1), \hat{\mu}_2 = (1,2), \hat{\mu}_3 = (2,1), \text{ and } \hat{\mu}_4 = (2,2).$$

The TPMs and TRMs of these policies are constructed from the individual TPMs and TRMs of each action. The TPMs are:

$$\mathbf{P}_{\hat{\mu}_1} = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix}; \mathbf{P}_{\hat{\mu}_2} = \begin{bmatrix} 0.7 & 0.3 \\ 0.2 & 0.8 \end{bmatrix};$$

$$\mathbf{P}_{\hat{\mu}_3} = \begin{bmatrix} 0.9 & 0.1 \\ 0.4 & 0.6 \end{bmatrix}; \mathbf{P}_{\hat{\mu}_4} = \begin{bmatrix} 0.9 & 0.1 \\ 0.2 & 0.8 \end{bmatrix}.$$

The TRMs are

$$\mathbf{R}_{\hat{\mu}_1} = \begin{bmatrix} 6 & -5 \\ 7 & 12 \end{bmatrix}; \mathbf{R}_{\hat{\mu}_2} = \begin{bmatrix} 6 & -5 \\ -14 & 13 \end{bmatrix},$$

$$\mathbf{R}_{\hat{\mu}_3} = \begin{bmatrix} 10 & 17 \\ 7 & 12 \end{bmatrix}; \mathbf{R}_{\hat{\mu}_4} = \begin{bmatrix} 10 & 17 \\ -14 & 13 \end{bmatrix}.$$

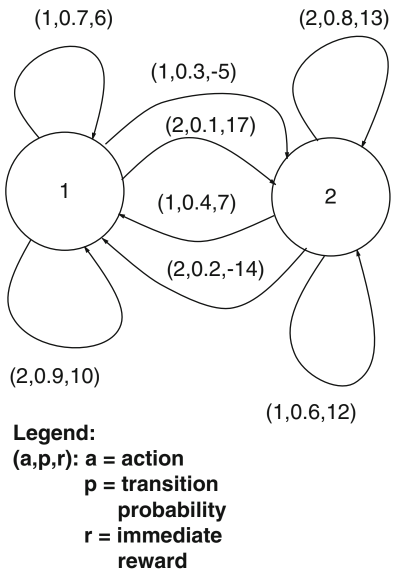

Figure 6.12. A two-state MDP

From the TPMs, using Eqs. (6.4) and (6.5), one can find the limiting probabilities of the states associated with each policy. They are:

$$\Pi_{\hat{\mu}_1}(1) = 0.5714 \text{ and } \Pi_{\hat{\mu}_1}(2) = 0.4286;$$

$$\Pi_{\hat{\mu}_2}(1) = 0.4000 \text{ and } \Pi_{\hat{\mu}_2}(2) = 0.6000;$$

$$\Pi_{\hat{\mu}_3}(1) = 0.8000 \text{ and } \Pi_{\hat{\mu}_3}(2) = 0.2000;$$

$$\Pi_{\hat{\mu}_4}(1) = 0.6667 \text{ and } \Pi_{\hat{\mu}_4}(2) = 0.3333.$$

We will next find the average reward of each of these four policies. We first evaluate the average *immediate* reward in each possible transition in the MDP, using Eq. (6.13). For this, we need the TPMs and the TRMs of each policy.

$$\bar{r}(1,\mu_1(1)) = p(1,\mu_1(1),1)r(1,\mu_1(1),1) + p(1,\mu_1(1),2)r(1,\mu_1(1),2)$$

$$= 0.7(6) + 0.3(-5) = 2.7.$$

$$\bar{r}(2,\mu_1(2)) = p(2,\mu_1(2),1)r(2,\mu_1(2),1) + p(2,\mu_1(2),2)r(2,\mu_1(2),2)$$

$$= 0.4(7) + 0.6(12) = 10.$$

$$\bar{r}(1,\mu_2(1)) = p(1,\mu_2(1),1)r(1,\mu_2(1),1) + p(1,\mu_2(1),2)r(1,\mu_2(1),2)$$

$$= 0.7(6) + 0.3(-5) = 2.7.$$

$$\bar{r}(2,\mu_2(2)) = p(2,\mu_2(2),1)r(2,\mu_2(2),1) + p(2,\mu_2(2),2)r(2,\mu_2(2),2)$$

$$= 0.2(-14) + 0.8(13) = 7.6.$$

$$\bar{r}(1,\mu_3(1)) = p(1,\mu_3(1),1)r(1,\mu_3(1),1) + p(1,\mu_3(1),2)r(1,\mu_3(1),2)$$

$$= 0.9(10) + 0.1(17) = 10.7.$$

$$\bar{r}(2,\mu_3(2)) = p(2,\mu_3(2),1)r(2,\mu_3(2),1) + p(2,\mu_3(2),2)r(2,\mu_3(2),2)$$

$$= 0.4(7) + 0.6(12) = 10.$$

$$\bar{r}(1,\mu_4(1)) = p(1,\mu_4(1),1)r(1,\mu_4(1),1) + p(1,\mu_4(1),2)r(1,\mu_4(1),2)$$

$$= 0.9(10) + 0.1(17) = 10.7.$$

$$\bar{r}(2,\mu_4(2)) = p(2,\mu_4(2),1)r(2,\mu_4(2),1) + p(2,\mu_4(2),2)r(2,\mu_4(2),2)$$

$$= 0.2(-14) + 0.8(13) = 7.6.$$

Now, using these quantities, we can now calculate the average reward of each individual policy. We will make use of Eq. (6.14).

Thus:

$$\rho_{\hat{\mu}_1} = \Pi_{\hat{\mu}_1}(1)\bar{r}(1,\mu_1(1)) + \Pi_{\hat{\mu}_1}(2)\bar{r}(2,\mu_1(2))$$

$$= 0.5741(2.7) + 0.4286(10) = 5.83,$$

$$\rho_{\hat{\mu}_2} = \Pi_{\hat{\mu}_2}(1)\bar{r}(1,\mu_2(1)) + \Pi_{\hat{\mu}_2}(2)\bar{r}(2,\mu_2(2))$$
$$= 0.4(2.7) + 0.6(7.6) = 5.64,$$

$$\rho_{\hat{\mu}_3} = \Pi_{\hat{\mu}_3}(1)\bar{r}(1,\mu_3(1)) + \Pi_{\hat{\mu}_3}(2)\bar{r}(2,\mu_3(2))$$

$$= 0.8(10.7) + 0.2(10) = 10.56, \text{ and}$$
$$\rho_{\hat{\mu}_4} = \Pi_{\hat{\mu}_4}(1)\bar{r}(1,\mu_4(1)) + \Pi_{\hat{\mu}_4}(2)\bar{r}(2,\mu_4(2))$$
$$= 0.6667(10.7) + 0.3333(7.6) = 9.6667.$$

It is clear that policy $\hat{\mu}_3 = (2,1)$ is the best policy, since it produces the *maximum* average reward.

Drawbacks of exhaustive enumeration. Clearly, exhaustive enumeration can only be used on small problems. Consider a problem with 10 states and 2 actions in each state. On this, if we were to use exhaustive enumeration, we would have to evaluate 2^{10} different policies. As such the computational burden of this method can quickly overwhelm it. We now turn to dynamic programming, which has a lower computational burden.

4. Average Reward MDPs and DP

The method of **dynamic programming** (**DP**), in the context of solving MDPs, was developed in the late 1950s with the pioneering work of Bellman [24] and Howard [144]. Dynamic programming has a considerably lower computational burden in comparison to exhaustive enumeration. The theory of DP has evolved significantly since the 1950s, and a voluminous amount of literature now exists on this topic. DP continues to be a main pillar of control optimization in discrete-event systems. Although the theory of DP is mathematically sophisticated, the main algorithms rest on simple systems of equations. As such, it is very easy to understand its basic principles, which form the main focus of the remainder of this chapter.

In this chapter and in the next, we will endeavor to present the main equations underlying DP without worrying about how the equations were derived. Also, we will present the main algorithms but will not attempt to *prove* that they generate optimal solutions (Chap. 11 presents details of that nature). Links to computer codes can be found at [121].

The system of equations, to which we were referring above, is often called the Bellman equation. The Bellman equation has several forms. In this section, we will concentrate on the average reward MDP, and in the next, on discounted reward. For both average and discounted reward, there are two different forms of the Bellman equation, based on which we have two different DP methods that go by the following names:

- Policy iteration, which uses the Bellman **policy** equation or the Poisson equation
- Value iteration, which uses the Bellman **optimality** equation

4.1. Bellman Policy Equation

We have seen above that associated with every given policy, there exists a scalar called the average reward of the policy. Similarly, associated with every policy, there exists a vector called the **value function** vector for the policy. The dimension of this vector is equal to the number of elements in $\mathcal{S}$, the set of decision-making states. The values of this vector's components can be determined by solving a linear system of equations, which is collectively called the Bellman equation *for a policy* or the **Bellman policy equation**. This equation, as stated above, is also sometimes called the *Poisson equation.*

A natural question at this stage is: what purpose will be served by finding the values of the elements of this vector? Much of DP revolves around this vector. The answer is that the values of these elements can actually help us find a *better* policy. The Bellman *policy* equation in the average reward context is:

$$h_{\hat{\mu}}(i) = \bar{r}(i, \mu(i)) - \rho_{\hat{\mu}} + \sum_{j=1}^{|\mathcal{S}|} p(i, \mu(i), j) h_{\hat{\mu}}(j) \text{ for each } i \in \mathcal{S}. \quad (6.15)$$

The above is a system of linear equations in which the number of equations equals the number of elements in the set $\mathcal{S}$, i.e., $|\mathcal{S}|$. The unknowns in the equations are the $h_{\hat{\mu}}$ terms. They are the elements of the value function vector associated with the policy $\hat{\mu}$. The other terms are defined below:

- $\mu(i)$ denotes the action selected in state i under the policy $\hat{\mu}$. Since the policy is known, each $\mu(i)$ is known.

- $\bar{r}(i, \mu(i))$ denotes the expected immediate reward in state i under policy $\hat{\mu}$. Each of these terms can be calculated from the TPMs and the TRMs.

- $p(i, \mu(i), j)$ denotes the one-step transition probability of jumping from state i to state j under the policy $\hat{\mu}$. Again, these terms can be obtained from the TPMs.

- $\rho_{\hat{\mu}}$ denotes the average reward associated with the policy $\hat{\mu}$, and it can be obtained, when the policy is known, from the TPMs and the TRMs as discussed in the context of exhaustive enumeration.

We will now discuss the celebrated policy iteration algorithm [144] to solve the average reward MDP.

4.2. Policy Iteration

The basic idea underlying policy iteration is to start with an arbitrarily selected policy, and then switch to a better policy in every iteration. This continues until no further improvement is possible. The advantage of policy iteration over exhaustive enumeration is that the solution is usually obtained using comparatively fewer iterations.

When a policy is selected, the Bellman policy equation is used to obtain the value function vector for that policy. This is called the *policy evaluation* stage, since in this stage we evaluate the value function vector associated with a policy. Then the value function vector is used to find a *better* policy. This step is called the *policy improvement* step.

The value function vector of the new (better) policy is then obtained. In other words, one returns to the policy evaluation step. This continues until the value function vector obtained from solving the Bellman policy equation cannot produce a better policy. (If you are familiar with the simplex algorithm of linear programming, you will see an analogy. In simplex, we start at a solution (corner point) and then move to a better point. This continues until no better point can be obtained.)

4.2.1 Steps in the Algorithm

Step 1. Set $k = 1$. Here k will denote the iteration number. Let the set of states be $\mathcal{S}$. Select any policy in an arbitrary manner. Let us denote the policy selected in the kth iteration by $\hat{\mu}_k$. Let $\hat{\mu}^*$ denote the optimal policy.

Step 2. (Policy Evaluation) Solve the following linear system of equations. For $i = 1, 2, \ldots, |\mathcal{S}|$,

$$h^k(i) = \bar{r}(i, \mu_k(i)) - \rho^k + \sum_{j=1}^{|\mathcal{S}|} p(i, \mu_k(i), j) h^k(j). \tag{6.16}$$

Here one linear equation is associated with each value of $i \in \mathcal{S}$. In this system, the unknowns are the h^k terms and ρ^k. The number of unknowns exceeds the number of equations by 1. Hence to solve the system, one should set any one of the h^k terms to 0, and then solve for the rest of the unknowns. The term ρ^k should **not** be set to 0.

Step 3. (Policy Improvement) Choose a new policy $\hat{\mu}_{k+1}$ such that

$$\mu_{k+1}(i) \in \arg\max_{a \in \mathcal{A}(i)} \left[\bar{r}(i, a) + \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) h^k(j) \right] \text{ for all } i \in \mathcal{S}.$$

If possible, one should set $\mu_{k+1}(i) = \mu_k(i)$ for each i. The significance of $\in$ in the above needs to be understood clearly. There may be more than one action that satisfies the argmax operator. Thus there may be multiple candidates for $\mu_{k+1}(i)$. However, the latter is selected in a way such that $\mu_{k+1}(i) = \mu_k(i)$ if possible.

Step 4. If the new policy is identical to the old one, that is, if $\mu_{k+1}(i) = \mu_k(i)$ for each i, then stop and set $\mu^*(i) = \mu_k(i)$ for every i. Otherwise, increment k by 1, and go back to the second step.

Policy iteration on Example A. We used policy iteration on Example A from Sect. 3.3.2. The results are shown in Table 6.1. The optimal policy is $(2, 1)$. See also the case study on a preventive maintenance problem at the end of the chapter. We will next discuss an alternative method, called value iteration, to solve the average reward MDP.

Table 6.1. Calculations in policy iteration for average reward MDPs on Example A

Iteration (k)	Policy selected	Values	ρ^k
1	$\hat{\mu}_1 = (1, 1)$	$h^1(1) = 0$ $h^1(2) = 10.43$	5.83
2	$\hat{\mu}_2 = (2, 1)$	$h^2(1) = 0$ $h^2(2) = -1.4$	10.56

4.3. Value Iteration and Its Variants

The value iteration algorithm is another very useful algorithm that can solve the MDP. It uses a form of the Bellman equation that is different than the form used in policy iteration. The idea of value iteration is due to Bellman [24].

Another major point of difference between value and policy iteration is that in value iteration, one does not solve any equations. This makes it easy to write computer program for the algorithm, which has contributed to its popularity. However, another advantage of value iteration will be realized in reinforcement learning, the main topic of Chap. 7.

We begin by presenting the Bellman **optimality** equation. A variant of this will be used in the value iteration and the relative value iteration algorithms.

$$J^*(i) = \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) - \rho^* + \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) J^*(j) \right] \text{ for each } i \in \mathcal{S}, \tag{6.17}$$

where $\mathcal{A}(i)$ denotes the set of actions allowed in state i and ρ^* denotes the average reward of the optimal policy. The J^* terms are the unknowns and are the components of the **optimal** value function vector $\vec{J}^*$. The number of elements in the vector $\vec{J}^*$ equals the number of states in the MDP.

Equation (6.17) is the famous Bellman **optimality** equation for average reward MDPs. Since the equation contains the max operator, it cannot be solved using linear algebra techniques such as Gauss elimination. It is considered to be non-linear because of the max operator.

4.3.1 Value Iteration: A Natural Version

Value iteration in its most natural form seeks to use the Bellman optimality equation in which ρ^* is forcibly set to zero. Note that ρ^* is unknown at the start. We will start with some arbitrary values for the value function vector. And then we will apply a transformation, i.e., an updating mechanism, derived from the Bellman optimality equation, on the vector repeatedly. This mechanism keeps updating (changing) the elements of the value function vector. The values themselves will *not* converge in this algorithm, i.e., the vectors generated in successive iterations will not approach each other. Therefore, we need some mechanism to terminate the algorithm. A suitable termination mechanism here employs the *span seminorm*, also called the span. We will denote the span seminorm of a vector in this book by $sp(.)$ and define it as:

$$sp(\vec{x}) = \max_i x(i) - \min_i x(i).$$

The span of a vector is by definition non-negative (take simple examples of vectors to verify this), and it denotes the *range* of values in the different components of the vector. It is important to inform the reader that a vector may keep changing in every iteration although its span may not change! This is because the span measures the range, and a changing vector may have a constant range. When the span stops changing, we will terminate our algorithm. We will establish later in Chap. 11 that the span actually *converges* to a finite number under certain conditions. We now present steps in natural value iteration.

Step 1: Set $k = 1$. Select any arbitrary values for the elements of a vector of size $|\mathcal{S}|$, and call the vector $\vec{J}^1$. Specify $\epsilon > 0$. Typically, ϵ is set to a small value such as 0.01.

Step 2: For each $i \in \mathcal{S}$, compute:

$$J^{k+1}(i) = \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) J^k(j) \right].$$

Step 3: If

$$sp(\vec{J}^{k+1} - \vec{J}^k) < \epsilon,$$

go to Step 4. Otherwise increase k by 1, and go back to Step 2.

Step 4: For each $i \in \mathcal{S}$, choose

$$d(i) \in \arg\max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) J^k(j) \right],$$

and stop. The ϵ-optimal policy is $\hat{d}$.

The implication of ϵ-optimality (in Step 4 above) needs to be understood. The smaller the value of ϵ, the closer we get to the optimal policy. Usually, for small values of ϵ, one obtains policies very close to optimal. The span of the difference vector $(\vec{J}^{k+1} - \vec{J}^k)$ keeps getting smaller and smaller in every iteration, and hence for a given positive value of ϵ, the algorithm terminates in a finite number of steps. Also note that in Step 2, we have used a *transformation* (see Chap. 1 for a definition) derived from the Bellman equation. Recall that a transformation takes a vector and produces a new vector from the vector supplied to it.

Do note that in Step 2 had we used the value of ρ^* (assuming that the value of ρ^* is known) instead of replacing it by zero, we would have obtained

- The same value for the span in Step 3 and
- The same sequence of maximizing actions in Step 2.

Together these two facts suggest that value iteration discussed above should yield the results obtained from using the Bellman transformation in Step 2. We will show in Chap. 11 that the Bellman transformation leads one to an optimal solution, and hence, it can be inferred that the value iteration algorithm above should also lead us to the optimal policy.

And yet a major difficulty with value iteration presented above is that the one or more of the values can become very large or very small during the iterations. This difficulty can be overcome by using *relative* value iteration, which we discuss in the next subsection. Before that, we present some numerical results to show the difficulty encountered with the value iteration algorithm above.

Value Iteration on Example A. See Table 6.2 for some sample calculations with value iteration on Example A (from Sect. 3.3.2). Note that the values gradually become larger with every iteration.

4.3.2 Relative Value Iteration

As discussed above, the major obstacle faced in using value iteration is that some of the values can become very large or very small. On problems with a small number of states with a value for ϵ that is not very small, one can *sometimes* use value iteration without running into this difficulty. However, usually, in larger problems, one of the values becomes very large or very small, and then the natural form of value iteration cannot be used. *Relative* value iteration (RVI), due to White [320], provides us with an elegant way out of this difficulty that we now discuss.

If the Markov chain of every policy is regular, which we have assumed above, we will select *any* state in $\mathcal{S}$ and call it the *distinguished* state. The updating mechanism (see Step 3 below) will use this distinguished state and keep the values from becoming too large or too small. Recall that in the natural form of value iteration, the corresponding updating mechanism (see Step 2 of natural value iteration) is derived from the Bellman equation by setting $\rho^* = 0$. We now present steps in RVI.

Step 1: Select any arbitrary state from $\mathcal{S}$ to be the distinguished state i^*. Set $k = 1$, and select arbitrary values for the vector $\vec{J}^1$. Specify the termination value $\epsilon > 0$.

Step 2: Compute for each $i \in \mathcal{S}$:

$$J^{k+1}(i) = \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) J^k(j) \right].$$

After calculations in this step for all states are complete, set $\rho = J^{k+1}(i^*)$.

Table 6.2. Calculations in value iteration for average reward MDPs: Note that the values get unbounded but the span of the difference vector gets smaller with every iteration. We start with $J^1(1) = J^1(2) = 0$

k	$J^k(1)$	$J^k(2)$	Span
2	10.7	10.0	0.7
3	21.33	20.28	0.35
4	31.925	30.70	0.175
.	...	...	...
.	...	...	...
.	...	...	...
153	1,605.40	1,604.00	0.001367
154	1,615.96	1,614.56	0.000684

Step 3: For each $i \in \mathcal{S}$, compute:

$$J^{k+1}(i) \leftarrow J^{k+1}(i) - \rho.$$

Step 4: If

$$sp(\vec{J}^{k+1} - \vec{J}^k) < \epsilon,$$

go to Step 5. Otherwise increase k by 1 and go back to Step 2.

Step 5: For each $i \in \mathcal{S}$, choose

$$d(i) \in \arg \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) J^k(j) \right],$$

and stop. The ϵ-optimal policy is $\hat{d}$; ρ is the optimal reward's estimate.

Essentially, the only difference between relative value iteration and regular value iteration is in the subtraction of a value. And yet, what a difference a subtraction makes! What is remarkable is that the values remain bounded (see Table 6.3). We would also like to state without proof here that:

Remark 1. The algorithm converges to the policy that value iteration would have converged to.

Remark 2. As k tends to ∞, ρ converges to ρ^*.

Remark 2 implies that when the algorithm converges, the values will converge to a solution of the Bellman optimality equation. Once $J(i^*)$ converges to ρ^*, it is clear from Step 3 that one would in fact be using a transformation based on the Bellman optimality equation. We will formally show in Chap. 11 that under some conditions, relative value iteration converges in the span to the optimal solution, i.e., a solution of the Bellman optimality equation.

Example A and RVI. In Table 6.3, we show the calculations with using relative value iteration on Example A from Sect. 3.3.2.

4.3.3 Average Reward: A General Expression

It is good to remind the reader here that we have assumed the time spent in a transition of a Markov process to be unity although the actual time spent may not be unity. This assumption is perfectly valid for the MDP where the transition time is not required to measure the performance metric. But if the time spent in each transition is used to measure the performance metric, which is the case in the semi-Markov decision process (SMDP), then this assumption is invalid.

We now present a *general* expression for the average reward per unit time in an MDP. Thus far, the expression used for average reward has used limiting probabilities (assuming regular Markov chains for all policies). The following expression, although equivalent, will not contain the limiting probabilities. If the time spent in a transition is not unity, one can view the expression below as that for the average reward per *jump* or *transition.* If the transition time is unity, per unit time and per jump are clearly equivalent.

The average reward per unit time calculated over an infinite time horizon, starting at state i and using a policy $\hat{\mu}$, can be expressed as:

$$\rho(i) = \liminf_{k\to\infty} \frac{\mathsf{E}\left[\sum_{s=1}^{k} r(x_s, \mu(x_s), x_{s+1}) | x_1 = i\right]}{k}, \text{ where}$$

k denotes the number of transitions (or time assuming that each transition takes unit time) over which the system is observed, x_s denotes the state from where the sth jump or state transition occurs under the policy $\hat{\mu}$, and E denotes the expectation operator over all trajectories that start under the condition specified within the square brackets. If you have trouble understanding why we use $\liminf$ here, you may replace it by lim at this stage.

It can be shown that for policies with regular Markov chains, the average reward is independent of the starting state i, and hence, $\rho(i)$ can be replaced by ρ. Intuitively, the above expression says that the average reward for a given policy is

$$\frac{\text{the expected sum of rewards earned in a very long trajectory}}{\text{the number of transitions in the same trajectory}}.$$

In the above, we assume that the associated policy is pursued within the trajectory. We now discuss the other important performance metric typically studied with an MDP: the discounted reward.

5. Discounted Reward MDPs and DP

The discounted reward criterion is another popular performance metric that has been studied extensively by researchers in the context of MDPs. In this section, we will focus on the use of DP methods

Table 6.3. Calculations in *Relative* value iteration for average reward MDPs: $\epsilon = 0.001$; ϵ-optimal policy found at $k = 12$; $J^1(1) = J^1(2) = 0$

Iteration (k)	$J^k(1)$	$J^k(2)$	Span	ρ
2	0	−0.7	0.7	10.760
3	0	−1.05	0.35	10.630
4	0	−1.225	0.175	10.595
5	0	−1.3125	0.0875	10.578
6	0	−1.35625	0.04375	10.568
7	0	−1.378125	0.021875	10.564
8	0	−1.389063	0.010938	10.562
9	0	−1.394531	0.005469	10.561
10	0	−1.397266	0.002734	10.561
11	0	−1.398633	0.001367	10.560
12	0	−1.399316	0.000684	10.560

to find the policy in an MDP that optimizes discounted reward. Links to computer codes for some DP algorithms are presented at [121].

The idea of discounting is related to the fact that the value of money reduces with time. To give a simple example: a dollar tomorrow is worth less than a dollar today. The discounting factor is the fraction by which money gets devalued in unit time. So for instance, if I earn \$3 today, \$5 tomorrow, \$6 the day after tomorrow, and if the discounting factor is 0.9 per day, then the present worth of my earnings will be:

$$3 + (0.9)5 + (0.9)^2 6.$$

The reason for raising 0.9 to the power of 2 is that tomorrow, the present worth of day-after-tomorrow's earning will be 0.9(6). Hence today, the present worth of this amount will be $0.9[0.9(6)] = (0.9)^2 6$.

In general, if the discounting factor is λ, and if $e(t)$ denotes the earning in the tth period of time, then the present worth of earnings over n periods of time can be denoted by

$$e(0) + \lambda e(1) + \lambda^2 e(2) + \cdots + \lambda^n e(n). \tag{6.18}$$

Note that the discounting factor is in fact defined as follows:

$$\lambda = \frac{1}{1+\gamma},$$

where γ is the rate of interest (see any standard text on engineering economics) expressed as a ratio. Since $0 < \gamma < 1$, we have that $\lambda < 1$. We now define the discounted reward.

5.1. Discounted Reward

By the discounted reward of a policy, we mean the expected total discounted reward earned in the Markov chain associated with the policy, when the policy is pursued over an infinitely long trajectory of the Markov chain. A technical definition for the discounted reward earned with a given policy $\hat{\mu}$ starting at state i is

$$v_{\hat{\mu}}(i) = \lim_{k\to\infty} \mathsf{E}\left[\sum_{s=1}^{k} \lambda^{s-1} r(x_s, \mu(x_s), x_{s+1}) | x_1 = i\right], \text{ where} \tag{6.19}$$

λ denotes the discounting factor, k denotes the number of transitions (or time assuming that each transition takes unit time) over which the system is observed, x_s denotes the state from where the sth jump or transition of system state occurs under the policy μ, and E denotes the expectation operator over all trajectories that operate under the condition specified in the square brackets.

In Eq. (6.19),

$$\mathsf{E}\left[\sum_{s=1}^{k}\lambda^{s-1}r(x_s,\mu(x_s),x_{s+1})|x_1=i\right]=$$

$$\mathsf{E}\left[r(i,\mu(i),x_2)+\lambda r(x_2,\mu(x_2),x_3)+\cdots+\lambda^{k-1}r(x_k,\mu(x_k),x_{k+1})\right].$$

This should make it obvious that the discounted reward of a policy is measured using the format discussed in Eq. (6.18).

5.2. Discounted Reward MDPs

In the discounted reward MDP, we have more than *one* performance metric (objective function), and hence the performance metric is expressed in the form of a vector. Remember that in the average reward case, the performance metric is the unique scalar quantity (ρ): the average reward of a policy. In the discounted problem, associated **with every state**, there exists a scalar quantity; we strive to maximize *each* of these quantities. These quantities are the elements of the so-called value function vector. There is no quantity like ρ in the discounted problem, rather there are as many quantities as there are states.

The value function for discounted reward was defined in Eq. (6.19). The term $v_{\hat{\mu}}(i)$ in Eq. (6.19) denotes the ith element of the value function vector. The number of elements in the value function vector is equal to the number of states. The definition in Eq. (6.19) is associated with a policy $\hat{\mu}$. The point to be noted is that the value function depends on the policy. (Of course it also depends on the discounting factor, the transition probabilities, and the transition rewards.)

Solving an MDP implies identifying the policy that returns a value function vector $\vec{v}^*$ such that

$$v^*(i)=\max_{\hat{\mu}}v_{\hat{\mu}}(i)\text{ for each }i\in\mathcal{S}.$$

The above means that the optimal policy will have a value function vector that satisfies the following property: each element of the vector is greater than or equal to the corresponding element of the value function vector of any other policy. This concept is best explained with an example.

Consider a 2-state Markov chain with 4 allowable policies denoted by $\hat{\mu}_1,\hat{\mu}_2,\hat{\mu}_3$, and $\hat{\mu}_4$. Let the value function vector be defined by

$$\begin{array}{llll} v_{\hat{\mu}_1}(1)=3; & v_{\hat{\mu}_2}(1)=8; & v_{\hat{\mu}_3}(1)=-4; & v_{\hat{\mu}_4}(1)=12; \\ v_{\hat{\mu}_1}(2)=7; & v_{\hat{\mu}_2}(2)=15; & v_{\hat{\mu}_3}(2)=1; & v_{\hat{\mu}_4}(2)=42; \end{array}$$

Now, from our definition of an optimal policy, policy $\hat{\mu}_4$ should be the optimal policy since the value function vector assumes the maximum value for this policy for each state. Now, the following question should rise in your mind at this stage. What if there is no policy for which the value function is maximized for each state? For instance consider the following scenario:

$$\begin{array}{cccc} v_{\hat{\mu}_1}(1) = 3; & v_{\hat{\mu}_2}(1) = 8; & v_{\hat{\mu}_3}(1) = -4; & v_{\hat{\mu}_4}(1) = 12; \\ v_{\hat{\mu}_1}(2) = 7; & v_{\hat{\mu}_2}(2) = 15; & v_{\hat{\mu}_3}(2) = 1; & v_{\hat{\mu}_4}(2) = -5; \end{array}$$

In the above setting, there is no one policy for which the value function is maximized for all the states. Fortunately, it has been proved that under the assumptions we have made above, there **exists** an optimal policy; in other words, there exists a policy for which the value function is maximized in *each state*. The interested reader is referred to [30, 270], among other sources, for the proof of this.

The important point that we need to address next is: how does one find the value function of any given policy? Equation (6.19) does not provide us with any direct mechanism for this purpose. Like in the average reward case, we will need to turn to the Bellman policy equation.

5.3. Bellman Policy Equation

The Bellman policy equation is a system of linear equations in which the unknowns are the elements of the value function associated with the policy. The Bellman **policy** equation (or the Bellman equation *for a given policy*) for the discounted reward MDP is

$$h_{\hat{\mu}}(i) = \bar{r}(i, \mu(i)) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i, \mu(i), j) h_{\hat{\mu}}(j) \text{ for each } i \in \mathcal{S}. \tag{6.20}$$

The number of equations in this system is equal to $|\mathcal{S}|$, the number of elements in the set $\mathcal{S}$. The unknowns in the equation are the $h_{\hat{\mu}}$ terms. They are the elements of the value function vector associated with the policy $\hat{\mu}$. The other terms are defined below.

- $\mu(i)$ denotes the action selected in state i under the policy $\hat{\mu}$ (since the policy is known, each $\mu(i)$ is also known).
- $\bar{r}(i, \mu(i))$ denotes the expected immediate reward in state i under policy $\mu(i)$.
- $p(i, \mu(i), j)$ denotes the one-step transition probability of jumping from state i to state j under the policy $\hat{\mu}$.

By solving the Bellman equation, one can obtain the value function vector associated with a given policy. Clearly, the value function vectors associated with each policy can be evaluated by solving the respective Bellman equations. Then, from the value function vectors obtained, it is possible to determine the optimal policy. This method is called the method of exhaustive enumeration.

Like in the average reward case, the method of exhaustive enumeration is not a very efficient method to solve the MDP, since its computational burden is enormous. For a problem of 10 states with two allowable actions in each, one would need to evaluate 2^{10} policies. The method of policy iteration is considerably more efficient.

5.4. Policy Iteration

Like in the average reward case, the basic principle of policy iteration is to start from any arbitrarily selected policy and then move to a better policy in every iteration. This continues until no further improvement in policy is possible.

When a policy is selected, the Bellman equation for a policy (Eq. (6.20)) is used to find the value function vector for that policy. This is known as the policy evaluation stage because in this stage we evaluate the value function vector associated with a policy. Then the value function vector is used to find a better policy. This step is called the policy improvement step. The value function vector of the new policy is then found. In other words, one returns to the policy evaluation step. This continues until the value function vector obtained from solving the Bellman equation cannot produce a better policy.

Steps in policy iteration for MDPs.

Step 1. Set $k = 1$. Here k will denote the iteration number. Let the number of states be $|\mathcal{S}|$. Select any policy in an arbitrary manner. Let us denote the policy selected in the kth iteration by $\hat{\mu}_k$. Let $\hat{\mu}^*$ denote the optimal policy.

Step 2. (Policy Evaluation) Solve the following linear system of equations for the unknown h^k terms. For $i = 1, 2, \ldots, |\mathcal{S}|$

$$h^k(i) = \bar{r}(i, \mu_k(i)) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i, \mu_k(i), j) h^k(j). \tag{6.21}$$

Step 3. (Policy Improvement) Choose a new policy $\hat{\mu}_{k+1}$ such that

$$\mu_{k+1}(i) \in \arg\max_{a\in\mathcal{A}(i)} \left[\bar{r}(i,a) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i,a,j)h^k(j) \right] \text{ for all } i \in \mathcal{S}.$$

If possible, one should set $\hat{\mu}_{k+1} = \hat{\mu}_k$.

Step 4. If the new policy is identical to the old one, i.e., if $\mu_{k+1}(i) = \mu_k(i)$ for each i, then stop and set $\mu^*(i) = \mu_k(i)$ for every i. Otherwise, increment k by 1, and return to Step 2.

Policy iteration on Example A. We used policy iteration on Example A from Sect. 3.3.2. The results are shown in Table 6.4. The optimal policy is $(2, 1)$.

Table 6.4. Calculations in policy iteration for discounted MDPs

k	Policy selected ($\hat{\mu}_k$)	Values
1	$(1, 1)$	$h^1(1) = 25.026$ $h^1(2) = 34.631$
2	$(2, 1)$	$h^2(1) = 53.03$ $h^2(2) = 51.87$

Like in the average reward case, we will next discuss the value iteration method. The value iteration method is also called the method of successive approximations (in the discounted reward case). This is because the successive application of the Bellman operator in the discounted case does lead one to the optimal value function. Recall that in the average reward case, the value iteration operator may not keep the iterates *bounded.* Fortunately this is not the case in the discounted problem.

5.5. Value Iteration

Like in the average reward case, the major difference between value iteration and policy iteration is that unlike in policy iteration, in value iteration, one does not have to solve any equations. This is a big plus—not so much in solving small MDPs, but in the context of simulation-based DP (reinforcement learning). We will discuss this issue in more detail in Chap. 7.

We begin by presenting the Bellman **optimality** equation for discounted reward.

$$J^*(i) = \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) J^*(j) \right] \text{ for each } i \in \mathcal{S}. \quad (6.22)$$

The notation is similar to that defined for the average reward case. Equation (6.22), i.e., the Bellman **optimality** equation for discounted reward contains the max operator; hence, it cannot be solved using linear algebra techniques, e.g., Gaussian elimination. However, the value iteration method forms a convenient solution method. In value iteration, one starts with some arbitrary values for the value function vector. Then a transformation, derived from the Bellman optimality equation, is applied on the vector successively until the vector starts approaching a fixed value. The fixed value is also called a *fixed point.* We will discuss issues such as convergence to fixed points in Chap. 11 in a more mathematically rigorous framework. However, at this stage, it is important to get an intuitive feel for a fixed point.

If a transformation has a unique fixed point, then no matter what vector you start with, if you keep applying the transformation repeatedly, you will eventually reach the fixed point. Several operations research algorithms are based on such transformations.

We will now present step-by-step details of the value iteration algorithm. In Step 3, we will need to calculate the max norm of a vector. See Chap. 1 for a definition of max norm. We will use the notation $||.||$ to denote the max norm.

Steps in value iteration for MDPs.

Step 1: Set $k = 1$. Select arbitrary values for the elements of a vector of size $|\mathcal{S}|$, and call the vector $\vec{J}^1$. Specify $\epsilon > 0$.

Step 2: For each $i \in \mathcal{S}$, compute:

$$J^{k+1}(i) \leftarrow \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) J^k(j) \right].$$

Step 3: If

$$||(\vec{J}^{k+1} - \vec{J}^k)|| < \epsilon(1-\lambda)/2\lambda,$$

go to Step 4. Otherwise increase k by 1 and go back to Step 2.

Step 4: For each $i \in \mathcal{S}$ choose

$$d(i) \in \arg\max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) J^k(j) \right] \text{ and stop.}$$

The reader should note that in Step 2, we have used a transformation derived from the Bellman equation. Also, to be noted is the fact that the max-norm of the difference vector $(\vec{J}^{k+1} - \vec{J}^k)$ decreases with every iteration. The reason for the use of the expression $\epsilon(1-\lambda)/2\lambda$ in Step 3 will be explained in more detail in Chap. 11. Essentially, the condition in Step 3 ensures that when the algorithm terminates, the max norm of the difference between the value function vector returned by the algorithm and the *optimal* value function vector is ϵ. Since the max norm is in some sense the "length" of the vector, the implication here is that the vector generated by the algorithm is different from the optimal value function vector by a length of ϵ. Thus by making ϵ small, one can obtain a vector that is as close to the optimal vector as one desires. Finally, unlike the average reward case, the values do *not* become unbounded in this algorithm.

Example A and value iteration. Table 6.5 shows the calculations with using value iteration on Example A from the Sect. 3.3.2. The discounting factor is 0.8.

5.6. Gauss-Siedel Value Iteration

The regular value iteration algorithm discussed above is slow and can take many iterations to converge. We now discuss its variant thatmay work faster: the Gauss-Siedel version. It differs from the

Table 6.5. Calculations in value iteration for discounted reward MDPs: The value of ϵ is 0.001. The norm is checked with $0.5\epsilon(1-\lambda)/\lambda = 0.000125$. When $k = 53$, the ϵ-optimal policy is found; we start with $J(1) = J(2) = 0$

k	$J(1)$	$J(2)$	Norm
2	10.700	10.000	10.7
3	19.204	18.224	8.504
4	25.984	24.892	6.781
.	...	...	...
.	...	...	...
53	53.032	51.866	0.000121

regular value iteration algorithm in that its updating mechanism uses *current* versions of the value function's elements; this is called **asynchronous** updating. This needs to be explained in more detail.

Review value iteration presented above. In Step 2, $J^{k+1}(i)$ is obtained using values of the vector $\vec{J}^k$. Now, consider a two-state example. When $J^{k+1}(2)$ is calculated, one already has the value of $J^{k+1}(1)$. But in Step 2 in the algorithm description we use $J^k(1)$. This is where the Gauss-Seidel version differs from the regular version. In Step 2 of Gauss Seidel, we will use $J^{k+1}(1)$.

In general, we will use $J^{k+1}(i)$ of every i for which the $(k+1)$th estimate is available. In other words, in the updating of the values, we will use the *latest* estimate of the value. The main idea is that the algorithm is based on the following update:

$$J(i) \leftarrow \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) J(j) \right].$$

Note that we avoid using the superscript k in the above, essentially implying that during any update, one should use the *latest* estimate of the value function of any state.

Example A and Gauss-Siedel value iteration. Table 6.6 shows the calculations with Gauss-Seidel value iteration on Example A from the Sect. 3.3.2. The discounting factor is 0.8. Note that it takes fewer iterations to converge in comparison to regular value iteration.

6. Bellman Equation Revisited

The validity of the Bellman equation will be proved using mathematical arguments in Chap. 11. However, in this section, we will motivate the Bellman equation in an intuitive manner. The intuitive explanation will throw considerable insight on the *physical* meaning of the value function.

We consider the Bellman policy equation (i.e., Bellman equation for a given policy) in the discounted case. The value function of a given policy for a given state *is* the *expected total discounted reward* earned over an infinitely long trajectory, if the given policy is adhered

Table 6.6. Gauss-Siedel value iteration for discounted reward MDPs: Here $\epsilon = 0.001$; the norm is checked with $0.5\epsilon(1-\lambda)/\lambda = 0.000125$; the ϵ-optimal is found at $k = 33$; we start with $J^1(1) = J^1(2) = 0$

k	$J(1)$	$J(2)$	Norm
2	12.644	18.610	18.612500
3	22.447	27.906	9.803309
.	...	...	...
.	...	...	...
33	39.674	43.645	0.000102

to in *every* state visited. Keeping this in mind, let us strive to obtain an expression for the value function of a given state i for a policy $\hat{d}$. Denoting this element by $v_{\hat{d}}(i)$, we have that the $v_{\hat{d}}(i)$ must equal the immediate reward earned in a transition from i *plus* the value function of the state to which it jumps at the end of the transition. For instance, if you were measuring the total reward along a trajectory and you encounter two states i and j along the trajectory—i first and then j—then the reward earned from i to the end of the trajectory would be the sum of (1) the immediate reward earned in going from i to j and (2) the total discounted reward earned from j to the end of the trajectory. See also Fig. 6.13. This is the basic idea underlying the Bellman equation.

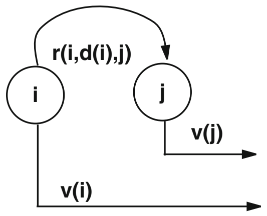

Figure 6.13. Total discounted reward calculation on an infinitely long trajectory

Let us next discuss what happens when the state transitions are probabilistic and there is a discounting factor λ. When the system is in a state i, it may jump to any one of the states in the system. Consider Fig. 6.14.

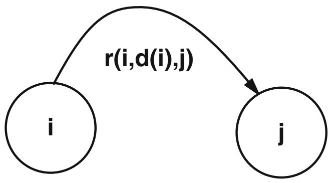

Figure 6.14. Immediate reward in one transition

So when the policy is $\hat{d}$, the expected total discounted reward earned from state i, which is $v_{\hat{d}}(i)$, can now be expressed as the sum of

a. The immediate reward earned in going to state j and

b. λ times the value function of the state j.

This is because the value function of state j denotes the expected total discounted reward from j onwards. But this quantity (that is the value function of state j) must be multiplied by λ, since it is earned one time step after leaving i. We assume that the immediate reward is earned *immediately* after state i is left. Hence the immediate reward is not multiplied by λ. However, j can actually be any one of the states in the Markov chain. Hence, in our calculation, we have to use an expectation over j. Thus, we have that

$$v_{\hat{d}}(i) = \sum_{j=1}^{|\mathcal{S}|} p(i, d(i), j) \left[r(i, d(i), j) + \lambda v_{\hat{d}}(j) \right],$$

which turns out to be the Bellman equation for a policy ($\hat{d}$) for state i. We hope that this discussion has served as an intuitive basis for the Bellman policy equation.

The Bellman *optimality* equation has a similar intuitive explanation: In each transition, to obtain the optimal value function at the current state, i, one seeks to add the *maximum* over the sum of immediate reward to the next state j and the "best" (optimal) value function from state j. Of course, like in the policy equation, we must compute an expectation over all values of j. We now discuss semi-Markov decision problems.

7. Semi-Markov Decision Problems

The **semi-Markov decision problem or process (SMDP)** is a more general version of the Markov decision problem (or process) in which the time spent in each state is an important part of the model. In the SMDP, we assume that the time spent in any transition of the Markov chain is not necessarily unity; in fact the time spent could be different for each transition. Furthermore, the time could be a deterministic quantity or a random variable. It should be clear now that the MDP is a special case of the SMDP in which the time of transition is always one.

One may think of an SMDP as a problem very like an MDP with the difference that every transition can take an amount of time that is not necessarily equal to 1 unit. Our discussions on the SMDP will be based on this argument. The Markov chain associated to any policy (or any action) in the SMDP is called the *embedded* Markov chain.

Since the transition time from one state to another depends on the state pair in question, it is common practice to store the expected transition times in matrices. We will first treat the case of deterministic transition times.

In the case of deterministic transition times, the transition time is fixed for a given state-state pair and for a given action. The transition time of going from state i to state j under the influence of action a can then be denoted by $\bar{t}(i,a,j)$ in a manner similar to the notation of the immediate reward $(r(i,a,j))$ and the transition probability $(p(i,a,j))$. Also, each value of $\bar{t}(i,a,j)$ can then be stored in a matrix—the transition time matrix (TTM). This matrix would be very like the transition reward matrix (TRM) or the transition probability matrix (TPM). An SMDP in which the time spent in any transition is a deterministic quantity will be referred to as a deterministic time semi-Markov decision problem (DeTSMDP).

When the transition times are random variables from any general (given) distribution, we have what we will call the *generalized* SMDP. The DeTSMDP is, of course, a special case of the generalized SMDP. When the distribution is exponential, we have a so-called continuous time Markov decision problem (CTMDP). Thus, we have three types of SMDPs: (i) generalized SMDPs, (ii) CTMDPs, and (iii) DeTSMDPs.

For the generalized SMDP, we will treat the term $\bar{t}(i,a,j)$ to be the **expected** transition time from i to j under action a. These times can also be stored within a matrix, which will be referred to as the TTM. For solving the SMDP, via dynamic programming, in addition to the TPMs and TRMs, we need the TTMs.

The exponential distribution (the distribution for transition times in CTMDPs) is said to have a memoryless property. This memoryless property is sometimes also called the Markov property, but the reader should not confuse this with the Markov property of the MDPs that we have discussed before. In fact, the Markov property of the exponential distributions for the transition times of a CTMDP does *not* allow us to treat the CTMDP as an MDP. Via a process called **uniformization**, however, one can usually convert a CTMDP into an MDP. The "uniformized" CTMDP can be treated as an MDP; but note that it has transition probabilities that are **different** from those of the original CTMDP. Thus, the CTMDP in its raw form is in fact an SMDP and not an MDP. We will *not* discuss the process of uniformization here. For this topic, the reader is referred to other texts, e.g., [30].

By a process called *discretization*, which is different than the uniformization mentioned above, we can also convert a generalized SMDP into an MDP. We will discuss discretization later in the context of average reward SMDPs. We now illustrate the notion of transition times with some simple examples.

Assume that the SMDP has two states numbered 1 and 2. Also, assume that the time spent in a transition from state 1 is uniformly distributed with a minimum value of 1 and a maximum of 2 (Unif(1,2)), while the same from state 2 is exponentially distributed with a mean of 3 (EXPO(3)); these times are the same for every action. Then, for generating the TTMs, we need to use the following values. For all values of a,

$$\bar{t}(1,a,1) = (1+2)/2 = 1.5; \bar{t}(1,a,2) = (1+2)/2 = 1.5;$$

$$\bar{t}(2,a,1) = 3; \bar{t}(2,a,2) = 3.$$

Obviously, the time could follow *any* distribution. If the distributions are not available, we must have access to the expected values of the transition time, so that we have values for each $\bar{t}(i,a,j)$ term in the model. These values are needed for solving the problem via dynamic programming.

It could also be that the time spent depends on the action. Thus for instance, we could also represent the distributions within the TTM matrix. We will use $\mathbf{T}_a$ to denote the TTM for action a. For a 2-state, 2-action problem, consider the following data:

$$\mathbf{T}_1 = \begin{bmatrix} \mathit{Unif}(5,6) & 12 \\ \mathit{Unif}(14,16) & \mathit{Unif}(5,12) \end{bmatrix}; \mathbf{T}_2 = \begin{bmatrix} \mathit{Unif}(45,60) & \mathit{Unif}(32,64) \\ \mathit{Unif}(14,16) & \mathit{Unif}(12,15) \end{bmatrix}.$$

In the above, the random values denoted by the distributions should be replaced by their mean values when performing calculations. We need to clarify that the distributions in a given TTM need not belong to the same family, and some (or all) entries in the TTM can also be constants.

7.1. Natural and Decision-Making Processes

In many MDPs and most SMDPs, we have two stochastic processes associated with every Markov chain. They are:

1. The natural process (NP).

2. The decision-making process (DMP).

Any stochastic process keeps track of the changes in the state of the associated system. The natural process keeps track of **every** state change that occurs. In other words, whenever the state changes, it gets recorded in the natural process. Along the natural process, the system does not return to itself after one transition. This implies that the natural process remains in a state for a certain amount of time and then jumps to a **different** state.

The decision process has a different nature. It records only those states in which an action needs to be selected by the decision-maker. Thus, the decision process may come back to itself after one transition. A decision-making state is one in which the decision-maker makes a decision. All states in a Markov chain may not be decision-making states; there may be several states in which no decision is made. Thus typically a subset of the states in the Markov chain tends to be the set of decision-making states. Clearly, as the name suggests, the decision-making process records only the decision-making states.

For example, consider a Markov chain with three states numbered 1, 2, 3, and 4. States 1 and 2 are decision-making states while 3 and 4 are not. Now consider the following trajectory:

$$\mathbf{1}, 3, 4, 3, \mathbf{2}, 3, \mathbf{2}, 4, \mathbf{2}, 3, 4, 3, 4, 3, 4, 3, 4, \mathbf{1}.$$

In this trajectory, the NP will look identical to what we see above. The DMP however will be:

$$\mathbf{1}, \mathbf{2}, \mathbf{2}, \mathbf{2}, \mathbf{1}.$$

This example also explains why the NP may change several times between one change of the DMP. It should also be clear that the DMP and NP coincide on the decision-making states (1 and 2).

We need to calculate the value functions of only the decision-making states. In our discussions on MDPs, when we said "state," we meant a decision-making state. Technically, for the MDP, the non-decision-making states enter the analysis only when we calculate the immediate rewards earned in a transition from a decision-making state to another decision-making state. In the SMDP, calculation of the transition rewards and the transition times needs taking into account the non-decision-making states visited. This is because in the transition from one decision-making state to another, the system may have visited non-decision-making states multiple times, which can dictate (1) the value of the immediate reward earned in the transition and (2) the transition time.

In simulation-based DP (reinforcement learning), the issue of identifying non-decision-making states becomes less critical because the simulator calculates the transition reward and transition time in transitions between decision-making states; as such we need not worry about the existence of non-decision-making states. However, if one wished to set up the model, i.e., the TRM and the TPM, careful attention must be paid to this issue.

In SMDPs, the time spent in the different transitions of the Markov chain is not necessarily the same. In the MDP, the time is irrelevant, and as such, in the calculation of the performance metric, we **assume** the time spent to be unity. This means that the average reward per unit time for an MDP is actually average reward per **transition** (jump) of the underlying Markov chain. For the SMDP, however, one must be careful in this respect. In the SMDP, average reward per unit time cannot be substituted by average reward per unit transition because the two quantities are different. In the SMDP, the average reward per transition will not mean the same thing.

In the next two subsections, we will discuss the average reward and the discounted reward SMDPs. Our discussions will be centered on algorithms that can be used to solve these problems.

7.2. Average Reward SMDPs

We first need to define the average reward of an SMDP. The average reward of an SMDP is a quantity that we want to *maximize.* Recall that $\bar{t}(i, a, j)$ is the mean transition time from state i to state j under the influence of action a. Now, the average reward, using a unit time basis, starting from a state i and following a policy $\hat{\mu}$, can be mathematically expressed as:

$$\rho_{\hat{\mu}}(i) \equiv \liminf_{k\to\infty} \frac{\mathsf{E}[\sum_{s=1}^{k} r(x_s, \mu(x_s), x_{s+1})|x_1 = i]}{\mathsf{E}[\sum_{s=1}^{k} \bar{t}(x_s, \mu(x_s), x_{s+1})|x_1 = i]}$$

where x_s is the state from where the sth jump (or state transition) occurs. The expectation is over the different trajectories that may be followed under the conditions within the square brackets.

The notation *inf* denotes the *infimum* (and *sup* denotes the *supremum*). An intuitive meaning of inf is minimum and that of sup is maximum. Technically, the infimum (supremum) is not equivalent to the minimum (maximum); however at this stage you can use the two interchangeably. The use of the infimum here implies that the average reward of a policy is the ratio of the minimum value of the total reward divided by the total time in the trajectory. Thus, it provides us with *lowest* possible value for the average reward.

It can be shown that the average reward is not affected by the state from which the trajectory of the system starts. Therefore, one can get rid of i in the definition of average reward. The average reward on the other hand depends on the policy used. Solving the SMDP means finding the policy that returns the highest average reward.

7.2.1 Exhaustive Enumeration

Like in the MDP case, for the SMDP, exhaustive enumeration is not an attractive proposition from the computational angle. However, from the conceptual angle, it is an important method to be first understood. Hence we now discuss it briefly.

We begin by defining the **average** time spent in a transition from state i under the influence of action a as follows:

$$\bar{t}(i,a) = \sum_{j=1}^{|\mathcal{S}|} p(i,a,j)\bar{t}(i,a,j),$$

where $\bar{t}(i,a,j)$ is the *expected* time spent in one transition from state i to state j under the influence of action a. Now, the average reward of an SMDP can also be defined as:

$$\rho_{\hat{\mu}} = \frac{\sum_{i=1}^{|\mathcal{S}|} \Pi_{\hat{\mu}}(i)\bar{r}(i,\mu(i))}{\sum_{i=1}^{|\mathcal{S}|} \Pi_{\hat{\mu}}(i)\bar{t}(i,\mu(i))} \tag{6.23}$$

where

- $\bar{r}(i,\mu(i))$ and $\bar{t}(i,\mu(i))$ denote the expected immediate reward earned and the expected time spent, respectively, in a transition from state i under policy $\hat{\mu}$ and
- $\Pi_{\hat{\mu}}(i)$ denotes the limiting probability of the underlying Markov chain for state i when policy $\hat{\mu}$ is followed.

The numerator in the above denotes the expected immediate reward in any given transition, while the denominator denotes the expected time spent in any transition. The above formulation (see e.g., [30]) is based on the renewal reward theorem (see Johns and Miller [155]), which essentially states that

$$\rho = \text{average reward per unit time} = \frac{\text{expected reward earned in a cycle}}{\text{expected time spent in a cycle}}. \tag{6.24}$$

Example B. The example discussed next is similar to the Example A in Sect. 3.3.2 as far as the TPMs and the TRMs are considered. What is new here is the TTM (transition time matrix). There are two states numbered 1 and 2 in an MDP and two actions, also numbered 1 and 2, are allowed in each state. The TTM for action a is represented by $\mathbf{T}_a$. Similarly, the TPM for action a is denoted by $\mathbf{P}_a$ and the TRM for action a by $\mathbf{R}_a$.

$$\mathbf{T}_1 = \begin{bmatrix} 1 & 5 \\ 120 & 60 \end{bmatrix}; \mathbf{T}_2 = \begin{bmatrix} 50 & 75 \\ 7 & 2 \end{bmatrix}.$$

$$\mathbf{P}_1 = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix}; \mathbf{P}_2 = \begin{bmatrix} 0.9 & 0.1 \\ 0.2 & 0.8 \end{bmatrix}.$$

$$\mathbf{R}_1 = \begin{bmatrix} 6 & -5 \\ 7 & 12 \end{bmatrix}; \mathbf{R}_2 = \begin{bmatrix} 10 & 17 \\ -14 & 13 \end{bmatrix}.$$

There are 4 possible policies that can be used to control the system:

$$\hat{\mu}_1 = (1,1), \hat{\mu}_2 = (1,2), \hat{\mu}_3 = (2,1), \text{ and } \hat{\mu}_4 = (2,2).$$

The TTMs of these policies are constructed from the individual TTMs of each action. The TTMs are:

$$\mathbf{T}_{\hat{\mu}_1} = \begin{bmatrix} 1 & 5 \\ 120 & 60 \end{bmatrix}; \mathbf{T}_{\hat{\mu}_2} = \begin{bmatrix} 1 & 5 \\ 7 & 2 \end{bmatrix}; \mathbf{T}_{\hat{\mu}_3} = \begin{bmatrix} 50 & 75 \\ 120 & 60 \end{bmatrix}; \mathbf{T}_{\hat{\mu}_4} = \begin{bmatrix} 50 & 75 \\ 7 & 2 \end{bmatrix}.$$

The TPMs and TRMs were calculated in Sect. 3.3.2. The value of each $\bar{t}(i, \mu(i))$ term can be calculated from the TTMs in a manner similar to that used for calculation of $\bar{r}(i, \mu(i))$. The values are:

$$\bar{t}(1, \mu_1(1)) = p(1, \mu_1(1), 1)\bar{t}(1, \mu_1(1), 1) + p(1, \mu_1(1), 2)\bar{t}(1, \mu_1(1), 2)$$

$$= 0.7(1) + 0.3(5) = 2.20;$$

$$\bar{t}(2, \mu_1(2)) = 84; \bar{t}(1, \mu_2(1)) = 2.20; \bar{t}(2, \mu_2(2)) = 3.00; \bar{t}(1, \mu_3(1)) = 52.5;$$

$$\bar{t}(2, \mu_3(2)) = 84.0; \bar{t}(1, \mu_4(1)) = 52.5; \text{ and } \bar{t}(2, \mu_4(2)) = 3.00.$$

The corresponding $\bar{r}$ terms, needed for computing ρ, were calculated before; see Sect. 3.3.2. Then, using Eq. (6.23), the average rewards of all the policies are as follows:

$$\rho_{\hat{\mu}_1} = 0.1564, \rho_{\hat{\mu}_2} = 2.1045, \rho_{\hat{\mu}_3} = 0.1796, \text{ and } \rho_{\hat{\mu}_4} = 0.2685.$$

Clearly, policy $\hat{\mu}_2$ is an optimal policy. The optimal policy here is different from the optimal policy for the MDP (in Sect. 3.3.2), which had the same TPMs and TRMs. Thus, although it is perhaps obvious, we note that the time spent in each state makes a difference to the average reward per unit time. Hence, it is the SMDP model rather than the MDP model that will yield the optimal solution, when time is used for measuring the average reward. We will now discuss the more efficient DP algorithms for solving SMDPs.

7.2.2 Policy Iteration

Policy iteration of average reward MDPs has a clean and nice extension to SMDPs. This is unlike value iteration which for average reward does not extend easily from MDPs to SMDPs. We will begin by presenting the Bellman equation for a given policy in the context of average reward SMDPs.

The Bellman **policy** equation in the average reward context for SMDPs is:

$$h_{\hat{\mu}}(i) = \bar{r}(i, \mu(i)) - \rho_{\hat{\mu}}\bar{t}(i, \mu(i)) + \sum_{j=1}^{|\mathcal{S}|} p(i, \mu(i), j) h_{\hat{\mu}}(j) \text{ for each } i \in \mathcal{S}. \tag{6.25}$$

In the above, notation is as usual; $\rho_{\hat{\mu}}$ denotes the average reward per unit time generated by policy $\hat{\mu}$. The only difference between the above equation and the Bellman policy equation for average reward MDPs lies in the time term, i.e., the $\bar{t}(i, \mu(i))$ term. The MDP version can be obtained by setting the time term to 1 in the above. Like in the MDP case, the above is a system of linear equations in which the number of equations is equal to the number of elements in the set $\mathcal{S}$. The unknowns in the equation are the $h_{\hat{\mu}}$ terms. They are the elements of the value function vector associated with the policy $\hat{\mu}$.

Steps in policy iteration for SMDPs.

Step 1. Set $k = 1$. Here k will denote the iteration number. Let the number of states be $|\mathcal{S}|$. Select any policy in an arbitrary manner. Let us denote the policy selected by $\hat{\mu}_k$. Let $\hat{\mu}^*$ denote the optimal policy.

Step 2. (Policy Evaluation) Solve the following linear system of equations.

$$h^k(i) = \bar{r}(i, \mu_k(i)) - \rho^k \bar{t}(i, \mu_k(i)) + \sum_{j=1}^{|\mathcal{S}|} p(i, \mu(i), j) h^k(j). \tag{6.26}$$

Here one linear equation is associated to each value of i. In this system, the unknowns are the h^k terms and ρ^k. Any one of the h^k terms should be set to 0.

Step 3. (Policy Improvement) Choose a new policy $\hat{\mu}_{k+1}$ so that for all $i \in \mathcal{S}$

$$\mu_{k+1}(i) \in \arg\max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) - \rho^k \bar{t}(i,a) + \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) h^k(j) \right].$$

If possible, one should set $\hat{\mu}_{k+1} = \hat{\mu}_k$.

Step 4. If the new policy is identical to the old one, i.e., if $\mu_{k+1}(i) = \mu_k(i)$ for each i, then stop, and set $\mu^*(i) = \mu_k(i)$ for every i. Otherwise, increment k by 1, and go back to the second step.

Policy iteration on Example B. We will show the use of policy iteration on generic example from Sect. 7.2.1. The results are shown in Table 6.7. The optimal policy is $(2, 1)$.

7.2.3 Value Iteration

Recall that value iteration for MDPs in the average reward context poses a few problems. For instance, the value of the average reward (ρ^*) of the optimal policy is seldom known beforehand, and hence the Bellman optimality equation cannot be used directly. Let us first analyze the Bellman optimality equation for average reward SMDPs.

$$J^*(i) = \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) - \rho^* \bar{t}(i,a) + \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) J^*(j) \right] \text{ for each } i \in \mathcal{S}. \tag{6.27}$$

The following remarks will explain the notation.

- The J^* terms are the unknowns. They are the components of the **optimal** value function vector $\vec{J}^*$. The number of elements in the vector $\vec{J}^*$ equals the number of states in the SMDP.
- The term $\bar{t}(i,a)$ denotes the expected time of transition from state i when action a is selected in state i.
- The term ρ^* denotes the average reward associated with the optimal policy.

Now in an MDP, although ρ^* is unknown, it is acceptable to replace ρ^* by 0 (which is the practice in regular value iteration for MDPs), or to replace it by the value function associated with some state of the Markov chain (which is the practice in relative value iteration for MDPs). In the MDP, this replacement is acceptable, because it does not affect the policy to which value iteration converges (although it does affect the actual values of the value function to which the algorithm converges). However, in the SMDP, such a replacement violates the Bellman equation. This is the reason we have difficulties in developing a regular value iteration algorithm for the average reward SMDP. We now illustrate this difficulty with an example.

7.2.4 Regular Value Iteration

Our example will show that naive application of value and relative value iteration will not work in the SMDP. Let us define $W(i,a)$ as follows:

$$W(i,a) \equiv \bar{r}(i,a) - \rho^* \bar{t}(i,a) + \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) J^k(j)$$

for all (i,a) pairs, where $J^k(i)$ denotes the estimate of the value function element for the ith state in the kth iteration of the value iteration algorithm.Let us define $W'(i,a)$, which deletes the ρ^* and also the time

Table 6.7. Calculations in policy iteration for average reward SMDPs (Example B)

Iteration (k)	Policy selected $(\hat{\mu}_k)$	Values	ρ
1	$(1,1)$	$h^1(1) = 0$ $h^1(2) = -7.852$	0.156
2	$(2,2)$	$h^2(1) = 0$ $h^2(2) = 33.972$	0.2685
3	$(1,2)$	$h^3(1) = 0$ $h^3(2) = 6.432$	2.1044

term $(\bar{t}(.,.))$, as follows:

$$W'(i,a) \equiv \bar{r}(i,a) + \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) J^k(j)$$

for all (i,a) pairs.

Provided the value of ρ^* is known in advance (this is not the case in practice, but let us s assume this to be the case for a minute), a potential value iteration update with the Bellman equation will be:

$$J^{k+1}(i) \leftarrow \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) - \rho^* \bar{t}(i,a) + \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) J^k(j) \right]. \quad (6.28)$$

Now, consider an SMDP with two actions in each state, where $\bar{t}(i,1) \neq \bar{t}(i,2)$. For this case, the above value iteration update can be written as:

$$J^{k+1}(i) \leftarrow \max\{W(i,1), W(i,2)\}. \tag{6.29}$$

If regular value iteration, as defined for the MDP, is used here, one must not only ignore the ρ^* term but also the time term. Then, an update based on a regular value iteration for the SMDP will be (we will show below that the following equation is meaningless):

$$J^{k+1}(i) \leftarrow \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) J^k(j) \right], \tag{6.30}$$

which can be written as:

$$J^{k+1}(i) \leftarrow \max\{W'(i,1), W'(i,2)\}. \tag{6.31}$$

Now, since $\bar{t}(i,1) \neq \bar{t}(i,2)$, it is entirely possible that using (6.29), one obtains $J^{k+1}(i) = W(i,1)$, while using (6.31), one obtains $J^{k+1}(i) = W'(i,2)$.

In other words, the update in (6.31) will not yield the same maximizing action as (6.29), where (6.29) is based on the Bellman equation, i.e., Eq. (6.28). (Note that both will yield the same maximizing action if all the time terms are equal to 1, i.e., in the MDP). Thus regular value iteration, i.e., (6.31), which is based on Eq. (6.30), is not an acceptable update for the SMDP in the manner shown above. It should be clear thus that Eq. (6.28) cannot be modified to eliminate ρ^* without eliminating $\bar{t}(.,.)$ at the same time; i.e., Eq. (6.30) has no validity for SMDPs! And herein lies the difficulty with value iteration for SMDPs. There is, however, a way around this difficulty, which we now discuss.

7.2.5 Discretization for Generalized SMDPs

For circumventing the difficulty described above, one recommended procedure is to "discretize" the SMDP and thereby convert it into an MDP. This process transforms the SMDP into an MDP by converting the immediate rewards to rewards measured on a unit time basis. The "discretized" Bellman optimality equation is:

$$J^*(i) = \max_{a \in \mathcal{A}(i)} \left[\bar{r}^{\vartheta}(i,a) - \rho_*^{\vartheta} + \sum_{j=1}^{|\mathcal{S}|} p^{\vartheta}(i,a,j) J^*(j) \right], \text{ for all } i \in \mathcal{S}, \tag{6.32}$$

where the replacements for $\bar{r}(i,a)$ and $p(i,a,j)$ are denoted by $\bar{r}^{\vartheta}(i,a)$ and $p^{\vartheta}(i,a,j)$, respectively, and are defined as:

$$\bar{r}^{\vartheta}(i,a) = \bar{r}(i,a)/\bar{t}(i,a);$$

$$p^{\vartheta}(i,a,j) = \begin{cases} \vartheta p(i,a,j)/\bar{t}(i,a) & \text{if } i \neq j \\ 1 + \vartheta[p(i,a,j) - 1]/\bar{t}(i,a) & \text{if } i = j \end{cases}$$

In the above, ϑ is chosen such that:

$$0 \leq \vartheta \leq \bar{t}(i,a)/(1 - p(i,a,j)) \text{ for all } a, i \text{ and } j.$$

That the above Bellman optimality equation in Eq. (6.32) is perfectly valid for the average reward SMDP and is equivalent to using Eq. (6.27) for value iteration purposes is proved in [30], amongst other sources. The advantage of Eq. (6.32) for value iteration is that unlike Eq. (6.27) it contains the term ρ_*^{ϑ} without time as a coefficient.

It is now possible to use natural value iteration using this transformed Bellman equation to solve the SMDP after setting ρ_*^{ϑ} to 0 like in the MDP (note that the resulting equation will be different than Eq. (6.30) because of the differences in the transition rewards and probabilities). It is also possible to use the relative value iteration algorithm just as in the MDP. Before moving on to the next topic, we note that the discretization procedure changes the transition probabilities of the Markov chain. Also, very importantly, the transformed Bellman equation has transition time encoded within the transformed transition probabilities and rewards. We do not present additional details of this topic, however, because in the simulation-based context (as we will see in the next chapter), a discretization of this nature can be avoided.

7.3. Discounted Reward SMDPs

Our attention in this subsection will be focussed on the DeTSMDP under an additional structural assumption on how the rewards are earned. This is done to keep the analysis simple. In the next chapter that considers a simulation-based approach, which is after all the focus of this book, we will cover a more general model (the generalized SMDP) for discounting without these assumptions.

We begin with the definition of an appropriate discounting factor for the SMDP in discounting. From standard engineering economics, we know that the present value and future value share a relationship due to the rate of interest (or inflation). If a fixed rate of interest is used that is compounded after a **fixed time** τ (e.g., typically, annually in our engineering economics problems), the relationship is:

$$\text{Present Value} = \frac{1}{(1+\gamma)^{\tau}} \text{Future Value},$$

where γ is the rate of return or rate of interest. In the above formula, γ should be expressed as a fraction (not in percent, although we usually think of rate of interest in percentages). If $\tau = 1$, i.e., transition time in the process is equal to 1, we have an MDP. Then λ, which we used to denote the discounting factor, is:

$$\lambda = \frac{1}{1+\gamma}.$$

If $\gamma > 0$, we have that $0 < \lambda < 1$. (Note that in average reward problems, we assume $\gamma = 0$.) Now, let us consider the case of **continuously compounded** interest, where the time period can be small (as opposed to fixed time period above). The discount factor λ will now be raised to τ, which leads to:

$$\lambda^{\tau} = \left(\frac{1}{1+\gamma}\right)^{\tau} \approx \left(e^{-\gamma}\right)^{\tau} = e^{-\gamma\tau}.$$

The above exponential approximation based on the exponential series, which is true when γ is small, has been well-known in the operations research community because the principle is widely used in the engineering economics community, where it is used to relate the present to the future value:

$$\text{Present Value} = e^{-\gamma\tau} \text{Future Value}.$$

In other words, to obtain the present value (worth) the future value is multiplied by $e^{-\gamma\tau}$, which, in a rough sense, becomes the discounting factor here. Here, we have continuously compounded rates, since the rewards are earned over time continuously. The Bellman equations must hence account for this new discounting factor. We now make two assumptions about the SMDP that we first study.

Assumption 6.4 *The discounted SMDP is a DeTSMDP.*

Assumption 6.5 *The immediate reward is earned in a lump sum immediately after the transition starts.*

Assumption 6.5 implies that the immediate reward term, $r(i, a, j)$, will denote the reward earned as soon as action a is selected in state

i, i.e., as soon as the transition begins. Hence, we assume that no reward is earned after that instant during that transition, and that all the reward associated with a transition is earned as a lump sum at the start of the transition. This assumption is not needed in the average reward problem. The reasons are as follows. In the average reward problem, since there is no discounting, the immediate reward earned at *any* time during the transition can simply be lumped together into $r(i,a,j)$. However, in the discounted SMDP, if a part of the immediate reward is earned *during* the transition, then since we assume continuous compounding, that portion of the immediate reward earned during the transition must be properly discounted. This can be done by a suitable modification of the Bellman equation that we will discuss at the end of this subsection. For the time being, in order to keep the analysis simple, we assume that the immediate reward is earned as soon as the transition starts.

Under Assumptions 6.4 and 6.5, the Bellman equation for a policy is given by:

$$h_{\hat{\mu}}(i) = \bar{r}(i,\mu(i)) + \sum_{j=1}^{|\mathcal{S}|} e^{-\gamma\bar{t}(i,\mu(i),j)} p(i,\mu(i),j) h_{\hat{\mu}}(j)$$

for each $i \in \mathcal{S}$. The role of the discounting factor here is played by $e^{-\gamma\bar{t}(i,\mu(i),j)}$.

7.3.1 Policy Iteration

In what follows, we present a discussion on the use of policy iteration in solving discounted reward SMDPs under Assumptions 6.4 and 6.5. The policy iteration algorithm is very similar to that used for MDPs. The discounting factor accommodates the time element.

Step 1. Set $k=1$. Here k will denote the iteration number. Let the set of states be denoted by $\mathcal{S}$. Select a policy in an arbitrary manner. Let us denote the policy selected by $\hat{\mu}_k$. Let $\hat{\mu}^*$ denote the optimal policy.

Step 2. (Policy Evaluation) Solve the following linear system of equations. For $i = 1, 2, \ldots, |\mathcal{S}|$,

$$h^k(i) = \bar{r}(i,\mu_k(i)) + \sum_{j=1}^{|\mathcal{S}|} e^{-\gamma\bar{t}(i,\mu_k(i),j)} p(i,\mu(i),j) h^k(j). \qquad (6.33)$$

Step 3. (Policy Improvement) Choose a new policy $\hat{\mu}_{k+1}$ such that

$$\mu_{k+1}(i) = \arg\max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \sum_{j=1}^{|\mathcal{S}|} e^{-\gamma \bar{t}(i,a,j)} p(i,a,j) h^k(j) \right].$$

If possible one should set $\hat{\mu}_{k+1} = \hat{\mu}_k$.

Step 4. If the new policy is identical to the old one, i.e., if $\mu_{k+1}(i) = \mu_k(i)$ for each i, then stop, and set $\mu^*(i) = \mu_k(i)$ for every i. Otherwise, increment k by 1, and go back to the second step.

7.3.2 Value Iteration

Value iteration can be carried out on discounted reward SMDPs under Assumptions 6.4 and 6.5.

Step 1: Set $k = 1$. Select arbitrary values for the elements of a vector of size $|\mathcal{S}|$, and call the vector $\vec{J}^1$. Specify $\epsilon > 0$.

Step 2: For each $i \in \mathcal{S}$, compute:

$$J^{k+1}(i) \leftarrow \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \sum_{j=1}^{|\mathcal{S}|} e^{-\gamma \bar{t}(i,a,j)} p(i,a,j) J^k(j) \right].$$

Step 3: If

$$sp(\vec{J}^{k+1} - \vec{J}^k) < \epsilon,$$

go to Step 4. Otherwise increase k by 1, and go back to Step 2.

Step 4: For each $i \in \mathcal{S}$, choose

$$d(i) \in \arg\max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \sum_{j}^{|\mathcal{S}|} e^{-\gamma \bar{t}(i,a,j)} p(i,a,j) J^k(j) \right],$$

and stop.

7.3.3 Generalized SMDPs Without Assumptions 6.4 and 6.5

We now develop the Bellman equations for the generalized SMDP where Assumptions 6.4 and 6.5 are not needed, i.e., the time spent in a transition need not be deterministic and the reward can be earned any time during the transition. This is the most general case that requires a suitable modification of the model presented above for the discounted SMDP.

Since we will assume that the transition time can be random, we need to introduce its distribution function. Also, we now need to distinguish between the immediate reward earned at start, which we call the lump sum reward, and the reward earned continuously over the transition. We present some notation needed to set up the Bellman equations. Let $f_{i,a,j}(.)$ denote the *pdf* of the transition time from i to j under the influence of action a, $r_C(i,a,j)$ denote the continuous rate of reward (e.g., in dollars per hour) from state i to j under action a, and $r_L(i,a,j)$ denote the immediate reward (e.g., in dollars) earned from i *immediately* after action a is taken and the system goes to j (lump sum reward). The Bellman equation for a policy $\hat{\mu}$ is: For every $i \in \mathcal{S}$, $J_\mu(i) =$

$$\sum_{j \in S} p(i,\mu(i),j) r_L(i,\mu(i),j) + R(i,\mu(i)) + \sum_{j \in S} \left[\int_0^\infty e^{-\gamma\tau} f_{i,\mu(i),j}(\tau) J_\mu(j) d\tau \right],$$

$$\text{where } R(i,a) = \sum_{j \in \mathcal{S}} \left[\int_0^\infty r_C(i,a,j) \frac{1-e^{-\gamma\tau}}{\gamma} f_{i,a,j}(\tau) d\tau \right]$$

$$\text{and } \lim_{\tau \to \infty} f_{i,a,j}(\tau) = p(i,a,j).$$

The Bellman optimality equation is given by: For every $i \in \mathcal{S}$, $J(i) =$

$$\max_{a \in \mathcal{A}(i)} \left[\sum_{j \in S} p(i,a,j) r_L(i,a,j) + R(i,a) + \sum_{j \in S} \left[\int_0^\infty e^{-\gamma\tau} f_{i,a,j}(\tau) J(j) d\tau \right] \right].$$

These equations involve integrals, and in order to use them within policy and value iteration, one must perform these integrations, which is possible when the *pdfs* are available. In a simulation-based context, however, these integrations can be avoided; we will see this in Chap. 7. Since our focus is on simulation-based solutions in this book, we bypass this topic. See [242, 30] for additional details.

8. Modified Policy Iteration

An approach outside of value and policy iteration goes by the name **modified policy iteration**. It combines advantages of value iteration and policy iteration and in some sense is devoid of the disadvantages of both. At this point, a discussion on the relative merits and demerits of both approaches is in order.

Policy iteration converges in fewer iterations than value iteration, but forces us to solve a system of linear equations in every iteration.

Value iteration is slower (computer time) than policy iteration, but does not require the solution of any system of equations.

Solving linear equations is not difficult if one has a small system of linear equations. But in this book, we are interested in models which can work without transition probabilities, and as such formulating a linear system is to be avoided. The real use of this algorithm will therefore become obvious in simulation-based approaches to DP. We will discuss this algorithm for MDPs only.

Modified policy iteration uses value iteration in the policy evaluation stage of policy iteration; thereby it avoids having to solve linear equations. Instead of using Gauss elimination or some other linear equation solver, the algorithm uses value iteration to solve the system. However, it uses the scheme of policy iteration of searching in policy space. In other words, it goes from one policy to another in search of the best doing value iteration in the interim.

An interesting aspect of the modified policy iteration algorithm is that the policy evaluation stage can actually be an *incomplete* value iteration. We now present details for the discounted MDP.

8.1. Steps for Discounted Reward MDPs

Step 1. Set $k = 1$. Here k will denote the iteration number. Let the set of states be denoted by $\mathcal{S}$. Assign arbitrary values to a vector $\vec{J}^k$ of size $|\mathcal{S}|$. Choose a sequence $\{m^k\}_{k=0}^{\infty}$ where the elements of this sequence take non-decreasing integer values, e.g., $\{5, 6, 7, 8, \ldots\}$. Specify $\epsilon > 0$.

Step 2. (Policy Improvement) Choose a policy $\hat{\mu}_{k+1}$ such that

$$\mu_{k+1}(i) \in \arg\max_{a\in\mathcal{A}(i)} \left[\bar{r}(i,a) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) J^k(j) \right].$$

If possible one should set $\hat{\mu}_{k+1} = \hat{\mu}_k$ when $k > 0$.

Step 3. (Partial Policy Evaluation)

Step 3a. Set $q = 0$ and for each $i \in \mathcal{S}$, compute:

$$W^q(i) \leftarrow \max_{a\in\mathcal{A}(i)} \left[\bar{r}(i,a) + \lambda \sum_{j} p(i,a,j) J^k(j) \right].$$

Step 3b. If

$$||(\vec{W}^q - \vec{J}^k)|| < \epsilon(1-\lambda)/2\lambda,$$

go to Step 4. Otherwise go to Step 3c.

Step 3c. If $q = m^k$, go to Step 3e. Otherwise, for each $i \in \mathcal{S}$, compute:

$$W^{q+1}(i) \leftarrow \left[\bar{r}(i, \mu_{k+1}(i)) + \lambda \sum_{j \in \mathcal{S}} p(i, \mu_{k+1}(i), j) W^q(j) \right].$$

Step 3d. Increment q by 1, and return to Step 3c.

Step 3e. For each $i \in \mathcal{S}$, set:

$$J^{k+1}(i) \leftarrow W^{m^k}(i),$$

increment k by 1 and return to Step 2.

Step 4. The policy $\hat{\mu}_{k+1}$ is the ϵ-optimal policy.

Remark 1: A Gauss-Seidel version of the value iteration performed in Step 3a and Step 3c can be easily developed. This can further enhance the rate of convergence.

Remark 2: The algorithm generates an ϵ-optimal policy, unlike regular policy iteration.

Remark 3: The algorithm can be shown to converge for **any** values of the sequence $\{m^k\}_{k=0}^{\infty}$. However, the values of the elements can affect the convergence rate. The choice of the elements in the sequence $\{m^k\}_{k=0}^{\infty}$ naturally affects how complete the policy evaluation is. An increasing sequence, such as $\{5, 6, 7, \ldots\}$, ensures that the evaluation becomes more and more complete as we approach the optimal policy. In the first few iterations, a very incomplete policy evaluation should not cause much trouble.

8.2. Steps for Average Reward MDPs

Step 1. Set $k = 1$. Here k will denote the iteration number. Let the set of states be denoted by $\mathcal{S}$. Assign arbitrary values to a vector $\vec{J}^k$ of size $|\mathcal{S}|$. Choose a sequence $\{m^k\}_{k=0}^{\infty}$ where the elements of this sequence take non-decreasing integer values, e.g., $\{5, 6, 7, 8, \ldots\}$. Specify $\epsilon > 0$.

Step 2. (Policy Improvement) Choose a policy $\hat{\mu}_{k+1}$ such that

$$\mu_{k+1}(i) \in \arg\max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) J^k(j) \right].$$

If possible, one should set $\hat{\mu}_{k+1} = \hat{\mu}_k$, when $k > 0$.

Step 3. (Partial Policy Evaluation)

Step 3a. Set $q = 0$, and for each $i \in \mathcal{S}$, compute:

$$W^q(i) = \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \sum_{j \in \mathcal{S}} p(i,a,j) J^k(j) \right].$$

Step 3b. If

$$sp(\vec{W}^q - \vec{J}^k) \le \epsilon,$$

go to Step 4. Otherwise go to Step 3c.

Step 3c. If $q = m^k$, go to Step 3e. Otherwise, for each $i \in \mathcal{S}$, compute:

$$W^{q+1}(i) = \left[\bar{r}(i, \mu_{k+1}(i)) + \sum_{j \in \mathcal{S}} p(i, \mu_{k+1}(i), j) W^q(j) \right].$$

Step 3d. Increment q by 1 and return to Step 3c.

Step 3e. For each $i \in \mathcal{S}$, set: $J^{k+1}(i) = W^{m^k}(i)$, increment k by 1 and return to Step 2.

Step 4. The policy $\hat{\mu}_{k+1}$ is the ϵ-optimal policy.

Some elements of the value function $\vec{W}$ can become very large or very small, because value iteration for average reward can make the iterates unbounded. As a safeguard, one can use relative value iteration in the policy evaluation step of modified policy iteration.

9. The MDP and Mathematical Programming

It is possible to set up both the average reward and the discounted reward MDPs as Linear Programs (LPs). The LP is a heavily researched topic in operations research, and efficient methods exist for solving relatively large LPs. (For an elaborate discussion on LP, see any undergraduate text, e.g., [293].) However, when casting even a small MDP as an LP, one creates large LPs. Hence, it is unclear whether this is the best approach for solving the MDP. However, the LP provides us with some interesting insight on the nature of the dynamic program. It should be noted that it is a parametric optimization (static) approach to solving what is essentially a control (dynamic) problem. We begin with the average reward case.

Average reward MDPs: An LP formulation. We present without proof an LP formulation that exploits the Bellman equation:

$$\text{Minimize } \rho \text{ subject to}$$

$$\rho + v(i) - \sum_{j=1}^{|\mathcal{S}|} p(i,\mu(i),j)v(j) \geq \bar{r}(i,\mu(i)) \text{ for } i = 1,2,\ldots,|\mathcal{S}| \text{ and all } \mu(i) \in \mathcal{A}(i).$$

Here all values of $v(j)$ for $j = 1,2,\ldots,|\mathcal{S}|$ and ρ are unrestricted in sign (URS). The decision variables in the LP are the $v(.)$ terms and ρ. The optimal policy can be determined from the optimal value of the vector $\vec{v}$. (See the last step of any value iteration algorithm.)

An alternative LP formulation, which is more popular (although it is difficult to see its relationship with the Bellman equation), is the dual of the LP above:

$$\text{Maximize } \sum_{i \in \mathcal{S}} \sum_{a \in \mathcal{A}(i)} \bar{r}(i,a)x(i,a) \text{ such that}$$

$$\text{for all } j \in \mathcal{S}, \quad \sum_{a \in \mathcal{A}(j)} x(j,a) - \sum_{i \in \mathcal{S}} \sum_{a \in \mathcal{A}(i)} p(i,a,j)x(i,a) = 0, \tag{6.34}$$

$$\sum_{i \in \mathcal{S}} \sum_{a \in \mathcal{A}(i)} x(i,a) = 1, \tag{6.35}$$

$$\text{and } x(i,a) \geq 0 \quad \forall (i,a). \tag{6.36}$$

In the above, the $x(i,a)$ terms are the decision variables. It can be proved [251] that a *deterministic* policy, i.e., a policy that prescribes a unique action for each state, results from the solution of the above. The deterministic policy can be obtained as follows from the optimal solution of the LP. For all $i \in \mathcal{S}$ and $a \in \mathcal{A}(i)$, compute:

$$d(i,a) = \frac{x^*(i,a)}{\sum_{b \in \mathcal{A}(i)} x^*(i,b)}$$

where $x^*(i,a)$ denotes the optimal value of $x(i,a)$ obtained from solving the LP above, and $d(i,a)$ will contain the optimal policy. Here

$d(i,a)$ must be interpreted as the probability of selecting action a in state i. Since it can be proved that a deterministic optimal policy will be contained in the $d(i,a)$ terms, it will be the case that $d(i,a)$ will in fact equal 1 if action a is to be chosen in state i and 0 otherwise. Thus, the optimal policy can be readily obtained from the values of $d(i,a)$.

SMDP: For the SMDP, the LP described above can be used after replacing constraint (6.35) with:

$$\sum_{i\in\mathcal{S}}\sum_{a\in\mathcal{A}(i)} x(i,a)\bar{t}(i,a) = 1.$$

Discounted reward MDPs: An LP formulation. It can be shown that if

$$v(i) \geq \bar{r}(i,\mu(i)) + \lambda\sum_{j=1}^{|\mathcal{S}|} p(i,\mu(i),j)v(j)$$

for all policies $\hat{\mu}$, then $v(i)$ is an upper bound for the optimal value $v^*(i)$. This paves the way for an LP. The formulation, using the x and v terms as decision variables, is:
Minimize $\sum_{j=1}^{|\mathcal{S}|} x(j)v(j)$ subject to
$\sum_{j=1}^{|\mathcal{S}|} x(j) = 1$ and $v(i) - \lambda\sum_{j=1}^{|\mathcal{S}|} p(i,\mu(i),j)v(j) \geq \bar{r}(i,\mu(i))$ for $i = 1,2,\ldots,|\mathcal{S}|$ and all $\mu(i)\in\mathcal{A}(i)$;
$x(j) > 0$ for $j = 1,2,\ldots,|\mathcal{S}|$, and $v(j)$ is URS for $j = 1,2,\ldots,|\mathcal{S}|$.

10. Finite Horizon MDPs

In the finite horizon MDP, the objective function is calculated over a finite time horizon. This is different than the infinite time horizon we have assumed in all the problems above.

With the finite time horizon, we can think of two objective functions:

1. Total expected reward and

2. Total expected discounted reward.

Note that in the infinite horizon setting, the total expected reward is usually infinite, but that is the not the case here. As such the total expected reward is a useful metric in the finite horizon MDP.

In this setting, every time the Markov chain jumps, we will assume that the number of **stages** (or time) elapsed since the start increases by 1. In each stage, the system can be in any one of the states in the state space. As a result, the value function depends not only on the state but also on the stage.

In the infinite horizon MDP, when we visit a state, we do not concern ourselves with whether it is the first jump of the Markov chain or the nth jump. Thus, for instance in the infinite horizon problem, state 15 visited in stage 1 is the same thing as state 15 visited in stage 119. This is because the value function is associated with only the state. In other words, in the infinite horizon problem, for every state, we associate a unique element of the value function. In the finite horizon MDP, on the other hand, we need to associate a unique element of the value function with a given state-stage pair. Thus state 15 visited in stage 1 and the same state 15 visited in stage 119 will have different value functions. Further, in the infinite horizon MDP, we have a unique transition probability matrix and a unique transition reward matrix *for a given action.* In the finite horizon MDP, we have a unique transition probability matrix and a unique transition reward matrix *for a given stage-action pair.*

It should be clear then that the finite horizon problem can become more difficult than the infinite horizon problem with the same number of states, if there is a large number of stages. For a small number of stages, however, the finite horizon problem can be solved easily. The popular method used for solving a finite horizon problem is called backward recursion dynamic programming. As we will see below, is has the flavor of value iteration.

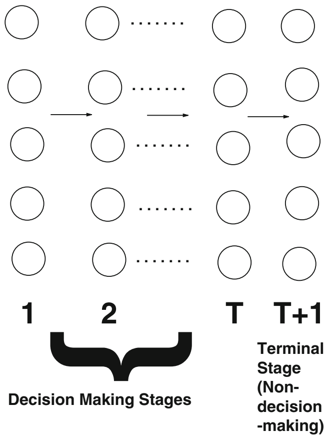

Figure 6.15. A finite horizon MDP

See Fig. 6.15 to obtain a pictorial idea of a finite horizon MDP. In each of the stages, numbered 1 through T, the system can visit a subset of the states in the system. It is entirely possible that in each stage the set of states visited is a proper subset of the total state space; that is in each stage, *any* state in the system may be visited. Thus in general, we can construct a transition probability matrix and a transition reward matrix for each action after replacing the state by a state-stage combination. In each stage, the system has a finite number of actions to choose from, and the system proceeds from one stage to the next until the horizon T is met. The goal is to maximize the total expected reward (or the total expected discounted reward) in the T transitions.

Our approach is based on the value iteration idea. We will endeavor to compute the optimal value function for each state-stage pair. We will need to make the following assumption:

- Since there is no decision making involved in the states visited in the $(T+1)$th stage, (where the trajectory ends), the value function in every state in the $(T+1)$th stage will have the same value. For numerical convenience, the value will be 0.

The Bellman optimality equation for the finite horizon problem for expected total discounted reward is given by:

$$J^*(i,s) \leftarrow \max_{a \in \mathcal{A}(i,s)} \left[\bar{r}(i,s,a) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i,s,a,j,s+1) J^*(j,s+1) \right] \tag{6.37}$$

for $i = 1, 2, \ldots, |\mathcal{S}|$, and $s = 1, 2, \ldots, T$ with some new notation:

- $J^*(i,s)$: the value function for the ith state when visited in the sth stage of the trajectory
- $\mathcal{A}(i,s)$: the set of actions allowed in state i when visited in the sth stage of the trajectory
- $\bar{r}(i,s,a)$: the expected immediate reward earned when in state i, in the sth stage, action a is selected
- $p(i,s,a,j,s+1)$: the transition probability of going from state i in the sth stage to state j in the $(s+1)$th stage when action a is selected in state i in the sth stage

The optimality equation for *expected total reward* can be obtained by setting λ, the discount factor, to 1 in Eq. (6.37). The algorithm that will be used to solve the finite horizon problem is *not* an iterative algorithm. It just makes one sweep through the stages and produces the optimal solution. We will now discuss the main idea underlying the backward recursion technique for solving the problem.

The backward recursion technique starts with finding the values of the states in the Tth stage (the final decision-making stage). For this, it uses (6.37). In the latter, one needs the values of the states in the next stage. We assume the values in the $(T+1)$th stage to be known (they will all be zero by our convention). Having determined the values in the Tth stage, we will move one stage backwards, and then determine the values in the $(T-1)$th stage.

The values in the $(T-1)$th stage will be determined by using the values in the Tth stage. In this way, we will proceed backward one stage at a time and find the values of all the stages. During the evaluation of the values, the optimal actions in each of the states will also be identified using the Bellman equation. We now present a step-by-step description of the backward recursion algorithm in the context of discounted reward. The expected *total reward* algorithm will use $\lambda = 1$ in the discounted reward algorithm.

A Backward Recursion. Review notation provided for Eq. (6.37).

Step 1. Let T (a positive integer) be the number of stages in the finite horizon problem. Decision-making will not be made in the $(T+1)$th stage of the finite horizon problem. The notation $u^*(i,s)$ will denote the optimal action in state i when the state is visited in the sth stage. The stages are numbered as $1, 2, \ldots, T+1$. For $i = 1, 2, \ldots, |\mathcal{S}|$, set

$$J^*(i, T+1) \leftarrow 0.$$

Step 2a. Set $s \leftarrow T$.

Step 2b. For $i = 1, 2, \ldots, |\mathcal{S}|$, choose

$$u^*(i,s) \in \arg\max_{a \in \mathcal{A}(i,s)} \left[\bar{r}(i,s,a) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i,s,a,j,s+1) J^*(j,s+1) \right].$$

Step 2c. For $i = 1, 2, \ldots, |\mathcal{S}|$, set

$$J^*(i,s) \leftarrow \left[\bar{r}(i,s,u^*(i,s)) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i,s,u^*(i,s),j,s+1) J^*(j,s+1) \right].$$

Step 2d. If $s > 1$, decrement s by 1 and return to Step 2b; otherwise STOP.

11. Conclusions

This chapter discussed the fundamental ideas underlying MDPs and SMDPs. The focus was on a finite state and action space within discrete-event systems. The important methods of value and policy iteration (DP) were discussed, and the two forms of the Bellman equation, the optimality equation and the policy equation, were presented for both average and discounted reward. The modified policy iteration algorithm was also discussed. Some linear programming for solving MDPs along with finite horizon control was covered towards the end briefly. Our goal in this chapter was to provide some of the theory underlying dynamic programming for solving MDPs and SMDPs that can also be used in the simulation-based context of the next chapter.

Bibliographic Remarks. Much of the material presented in this chapter is classical and can be found in [250, 242, 30, 270]. For an in-depth discussion on stochastic processes, the reader is referred to [295]. For highly intuitive and accessible accounts on MDPs and Markov chains, see [134, 293]. Filar and Vrieze [86] present the MDP as a special case of a game-theoretic problem, thereby tying MDP theory to the much broader framework of stochastic games. They also discuss several open research issues of related interest.

Much of the pioneering work in algorithms and DP theory was done by Bellman [23], Shapley [271] (value iteration, 1953), Howard [144] (policy iteration, 1960), White [320] (relative value iteration, 1963), and van Nunen [305] (modified policy iteration, 1976). Shapley [271] made important parallel contributions in game theory (which are applicable to MDPs) at a very early date (1953). Finally, we need to point out that in this book we have focussed on the case of discrete-event systems with finite state and action spaces governed by Markov chains; the theory of control optimization is much broader in its scope.

Case study on total productive maintenance. Our case study on total productive maintenance, drawn from [112, 126], is meant to illustrate the use of DP on a problem larger than Example A. Many systems fail with an increasing probability as they age, and such systems can often benefit from preventive maintenance. Well-known examples of such systems are manufacturing (production) lines, bridges, roads, and electric power plants.

We consider a production line that deteriorates with time. We make the following assumptions about it: (1) The line is needed every day, and every morning the manager must decide whether to continue with one more production cycle or shut the line down for preventive maintenance. (2) If the line fails during the day, the repair takes the remainder of the day; further the line becomes available for production after repair only the next morning. (3) The preventive maintenance takes the entire day. (4) After a repair or a maintenance, the line is as good as new. (5) Let i denote the number of days elapsed since the last preventive maintenance or repair (subsequent to a failure); then the probability of failure during the ith day can be modeled as $1-\xi\psi^{i+2}$, where ξ and ψ are scalars in the interval $(0,1)$, whose values can be estimated from the data for time between successive failures of the system.

We will use i to denote the state of the system, since this leads to a Markov chain. In order to construct a finite Markov chain, we define for any given positive value of $\epsilon \in \Re$, $\bar{i}_\epsilon$ to be the minimum integer value of i such that the probability of failure on the $\bar{i}$th day is less than or equal to $(1-\epsilon)$. Since we will set ϵ to some pre-fixed value, we can drop ϵ from our notation. In theory, the line will have some probability of not failing after *any* given day, making the state space infinite, but our definition of $\bar{i}$ permits truncation of the infinite state space to a finite one. The resulting state space will be: $\mathcal{S} = \{0,1,2,\ldots,\bar{i}\}$. This means that the probability of failure on the $\bar{i}$th day (which is very close to 1) will be assumed to equal 1.

Clearly, when a maintenance or repair is performed, i will be set to 0. If a successful day of production occurs, i.e., the line does not fail during the day, the state of the system is incremented by 1. The action space is: $\{produce, maintain\}$. C_m and C_r denote the cost of one maintenance and one repair respectively. Then, we have the following transition probabilities for the system.

For action *produce*: For $i = 0,1,2,\ldots,\bar{i}-1$

$$p(i, produce, i+1) = \xi\psi^{i+2}; \quad p(i, produce, 0) = 1-\xi\psi^{i+2}.$$

For $i=\bar{i}$, $p(i, produce, 0) = 1$. For all other cases not specified above, $p(., produce, .) = 0$. Further, for all values of i,

$$r(i, produce, 0) = -C_r; \quad r(i, produce, j) = 0 \text{ when } j \neq 0.$$

For the action *maintain*: For all values of i, $p(i, maintain, 0) = 1$ and $r(i, maintain, 0) = -C_m$. For all other cases not specified above,

$$p(., maintain, .) = 0; \quad r(., maintain, .) = 0.$$

Numerical Result: We set $\xi = 0.99$, $\psi = 0.96$, $C_m = 4$, $C_r = 2$, and $\bar{i} = 30$. Thus, we have 31 states and 2 actions. Our objective function is average reward, and the optimal policy, which is determined via policy iteration, turns out to be one with a threshold nature; i.e., the action is to produce for $i = 0,1,\ldots,5$ and to maintain for all values of i from 6 onwards. The policy iteration algorithm took 3 iterations and generated the following values in the final iteration:

$v(0){=}0; v(1) = -0.4098; v(2){=}{-}0.7280; v(3){=}{-}0.9701; v(4) = -1.1438;$

$v(5) = -1.2490$; and for all $i \geq 6, v(i) = -1.2757$. In policy iteration, the first policy chooses production in each state. Further, $\rho^1 = -0.7566$, $\rho^2 = -0.7425$ and $\rho^3 = \rho^* = -0.7243$. Note that all the values are negative, because we use rewards (to be consistent with our notation in this chapter), whereas we only have costs in this model.

Chapter 7

CONTROL OPTIMIZATION WITH REINFORCEMENT LEARNING

1. Chapter Overview

This chapter focuses on a relatively new methodology called reinforcement learning (RL). RL will be presented here as a form of simulation-based dynamic programming, primarily used for solving Markov and semi-Markov decision problems. Pioneering work in the area of RL was performed within the artificial intelligence community, which views it as a "machine learning" method. This perhaps explains the roots of the word "learning" in the name reinforcement learning. We also note that within the artificial intelligence community, "learning" is sometimes used to describe function approximation, e.g., regression. Some kind of function approximation, as we will see below, usually accompanies RL. The word "reinforcement" is linked to the fact that RL algorithms can be viewed as agents that learn through trials and errors (feedback).

But other names have also been suggested for RL. Some examples are *neuro-dynamic programming* (NDP) (see Bertsekas and Tsitsiklis [33]) and *adaptive* or *approximate dynamic programming* (ADP) (coined in Werbös [314]). Although we will stick to the original name, reinforcement learning, we emphasize that our presentation here will be through the viewpoint of dynamic programming.

For this chapter, the reader should review the material presented in the previous chapter. In writing this chapter, we have followed an order that *differs somewhat* from the one followed in the previous chapter.

© Springer Science+Business Media New York 2015
A. Gosavi, *Simulation-Based Optimization*, Operations Research/Computer Science Interfaces Series 55,
DOI 10.1007/978-1-4899-7491-4_7

We *begin* with discounted reward, and then discuss average reward. Like in the previous chapter, we discuss finite horizon problems at the end.

This is a long chapter, and hence a note on how it is organized. In Sect. 2, we discuss the twin curses of DP and the power of RL that helps in breaking them. Sect. 3 provides the fundamental ideas underlying the RL machinery. Sect. 4 focusses on RL methods for MDPs, while Sect. 5 presents their counterparts for SMDPs. Sect. 6 outlines the main ideas underlying model-building algorithms. Finite horizon MDPs are discussed in Sect. 7. Function approximation is presented in Sect. 8. Sect. 9 presents some concluding remarks. Codes in C can be found in [121].

2. The Twin Curses of DP

It is important to motivate the need for RL at this time. After all, dynamic programming (DP) is guaranteed to generate optimal solutions for MDPs (Markov Decision Problems) and SMDPs (semi-Markov Decision Problems). Obtaining the transition probabilities, rewards, and times (transition times are needed in SMDPs) is often a difficult and tedious process that involves complex mathematics. DP requires the values of all these quantities. RL does not, and despite this, it can generate near-optimal solutions. Herein lies its power.

The transition probabilities, rewards, and times together constitute what is known as the **theoretical model** of a system. Evaluating the transition probabilities often involves evaluating multiple integrals that contain the *pdfs* and/or *pmfs* of random variables. A complex stochastic system with many random variables can make this a very challenging task. It is for this reason that DP is said to be plagued by the **curse of modeling**. In Chap. 12, we will discuss examples of such systems.

Although the MDP model can lead to optimal solutions, it is often avoided in practice, and inexact or *heuristic approaches* (also called heuristics) are preferred. This is primarily because, as stated above, constructing the theoretical model required for solving an MDP is challenging for complex systems. Hence, a method that solves an MDP without constructing its theoretical model is bound to be very attractive. It turns out that RL is one example of methods of this kind.

Even for many relatively *small* real-world stochastic systems with complex theoretical models, computing the transition probabilities may be a cumbersome process. Hence the curse of modeling may apply even for small problems. For large systems, however, an additional curse applies, and classical DP is simply ruled out. Consider for instance the following example: an MDP with 1,000 states and 2 actions in each state. The TPM (transition probability matrix) of each action would contain $1{,}000^2 = 10^6$ (a million) elements. It is difficult if not impossible to store such large matrices in our computers. It is thus clear that in the face of large-scale problems with huge solution spaces with thousands of states, DP is likely to break down. This difficulty is often called the **curse of dimensionality**.

DP is thus doubly cursed—by the curse of dimensionality and the curse of modeling.

2.1. Breaking the Curses

RL can provide a way out of both of these curses:

1. RL does not need apriori knowledge of the transition probability matrices. The precise mechanism of avoiding these transition probabilities will be explained below. RL is said to avoid the curse of modeling, since it does not require the theoretical model. (Note that it *does* require the distributions of the governing random variables, and hence is not a non-parametric approach.)

2. RL *does* need the elements of the value function of DP, however. The value function in RL is defined in terms of the so-called Q-factors. When the MDP has millions of states, it translates into millions of Q-factors. In RL, one does not store these explicitly, i.e., instead of reserving one storage space in the computer's memory for each Q-factor, one stores the Q-factors in the form of a *handful* of scalars, via function-approximation techniques. In other words, a relatively small number of scalars are stored, which upon suitable manipulation can provide the value of any Q-factor in the state-action space. Function approximation thus helps RL avoid the dimensionality curse.

Heuristics are *inexact* but produce solutions in a reasonable amount of computational time, usually without extensive computation. Heuristics typically make some simplifying assumptions about the system. RL on the other hand is rooted in the DP framework and thereby inherits several of the latter's attractive features, e.g., generating high-quality solutions. Although the RL approach is also approximate, it uses the Markov decision model and the DP framework, which is quite powerful, and even near-optimal solutions, which RL is capable of generating, can often outperform heuristics.

We now discuss the tradeoffs in using heuristics versus DP methods. Consider a stochastic decision-making problem that can theoretically be cast as an MDP (or an SMDP) and can also be solved with heuristic methods. It may be difficult to solve the problem using the MDP model, since the latter needs the setting up of the theoretical model. On the other hand, the heuristic approach to solve the same problem may require less effort technically. Let us compare the two approaches against this backdrop.

- A DP algorithm is iterative and requires several computations, thereby making it computationally intensive. In addition, it needs the transition probabilities—computing which can get very technical and difficult. But if it manages to produce a solution, the solution is of a high quality.

- A heuristic algorithm usually makes several modeling assumptions about the system. Further, heuristics have a light computational burden (unlike DP with its TPM generation and iterative updating). It is another matter, however, that heuristic solutions may be far away from the optimal!

See Table 7.1 for a tabular comparison of RL, DP, and heuristics in terms of the level of modeling effort and the quality of solutions generated.

To give an example, the EMSR (expected marginal seat revenue) technique [201] is a heuristic used to solve the so-called seat-allocation problem in the airline industry. It is a simple model that quickly provides a good solution. However, it does not take into account several characteristics of the real-life airline reservation system, and as such its solution may be far from optimal. On the other hand, the MDP approach, which can be used to solve the same problem, yields superior solutions but is considerably more difficult because one has to first estimate the transition probabilities and then use DP. In such cases, not surprisingly, the heuristic is often preferred to the exact MDP approach in practice. So where is RL in all of this?

RL can generally outperform heuristics, and at the same time, the modeling effort in RL is lower than that of DP. Furthermore, on large-scale MDPs, DP breaks down, but RL does not. In summary, for solving problems whose transition probabilities are hard to estimate, RL is an attractive near-optimal approach that may outperform heuristics.

The main “tool” employed by RL is simulation. It uses simulation to avoid the computation of the transition probabilities. A simulator

Table 7.1. A comparison of RL, DP, and heuristics: Note that both DP and RL use the MDP model

Method	Level of modeling effort	Solution quality
DP	High	High
RL	Medium	High
Heuristics	Low	Low

does need the distributions of the random variables that govern the system's behavior. The transition rewards and the transition times are automatically calculated within a simulator. The avoidance of transition probabilities is not a miracle, but a fact backed by simple mathematics. We will discuss this issue in great detail.

To summarize our discussion, RL is a useful technique for large-scale MDPs and SMDPs on which DP is infeasible and heuristics provide poor solutions. In general, however, if one has access to the transition probabilities, rewards, and times, DP should be used because it is guaranteed to generate optimal solutions, and RL is not necessary there.

2.2. MLE and Small MDPs

Since obtaining the TPM (transition probability matrix) may serve as a major obstacle in solving a problem, one way around it is to simulate the system and generate the TPM and TRM from the simulator. After the transition probabilities and the transition rewards are obtained, one can then use the classical DP algorithms discussed in the previous chapter. We will now discuss how exactly this task may be performed. We remind the reader of an important, although obvious, fact first.

> To simulate a stochastic system that has a Markov chain underlying it, the transition probabilities of the system are *not* needed. For simulation, however, one *does* need distributions of the input random variables (that govern the system's behavior).

For example, consider a queuing system, e.g., M/M/1 system, that can be modeled as a Markov chain. For simulation, the transition probabilities of the underlying Markov chain are not needed. It is sufficient to know the distributions of the inter-arrival time and the service time.

To estimate transition probabilities via simulators, one can employ a relatively straightforward counting procedure, which works as follows: Suppose we want to calculate the transition probability $p(i, a, j)$. Recall that this is the one-step transition probability of going from state i to state j when action a is selected in state i. In the simulation program, we will need to keep two counters, $V(i, a)$ and $W(i, a, j)$, both of which are initialized to 0. Whenever action a is selected in state i in the simulator, the counter $V(i, a)$ will be incremented by 1. When, as a result of this action, the system goes to j from i in one step, $W(i, a, j)$ will be incremented by 1. From the definition of probabilities (see Appendix), then it follows that:

$$p(i, a, j) \simeq \frac{W(i, a, j)}{V(i, a)}.$$

The above is obviously an estimate, which *tends* to the correct value as $V(i, a)$ tends to infinity. This implies that in order to obtain good estimates of the TPM, each state-action pair should be tried infinitely often in the simulator. The expected immediate rewards can also be obtained in a similar fashion. The process described above for estimation is also called a **Maximum Likelihood Estimation** (MLE) and is a well-studied topic in statistics.

After generating estimates of the TPMs and the expected immediate rewards, one can solve the MDP via DP algorithms. This, unfortunately, is rarely efficient, because it takes a long time (many samples) to generate good estimates of the TPMs and the expected immediate rewards in the simulator. However, there is nothing inherently wrong in this approach, and the analyst may certainly want to try this for small systems (whose transition probabilities are difficult to find theoretically) as an alternative to RL. A strength of this approach is that after the TPMs and the expected immediate rewards are generated, DP methods, which we know are guaranteed to generate optimal solutions, can be employed.

For those small systems where the curse of modeling applies and it is tedious to theoretically determine the transition probabilities, this method may be quite suitable. But for large systems with several thousand states, it is ruled out because one would have to store each element of the TPM and the expected immediate rewards. It may be possible then to use a "state-aggregation" approach in which several states are combined to form a *smaller* number of "aggregate" states; this will lead to an approximate model with a *manageable* number of states. However, state aggregation is not always feasible, and even when it is, it may not yield the best results. Hence, we now present

RL in which the computation of the TPM and the immediate rewards can be avoided altogether.

3. Reinforcement Learning: Fundamentals

At the very outset, we need to state the following:

> RL is an offshoot of DP. It provides a way of performing dynamic programming (DP) within a simulator.

We have used this perspective throughout the book, so that RL does not come across as a heuristic tool that magically works well in practice. Furthermore, we will describe RL algorithms in a manner that clearly exposes their roots in the corresponding DP algorithms. The reassuring fact about this is that we know DP algorithms are guaranteed to generate optimal solutions. Therefore, as much as is possible, we will derive RL algorithms from their DP counterparts; however, detailed mathematical analysis of optimality for RL has been relegated to Chap. 11.

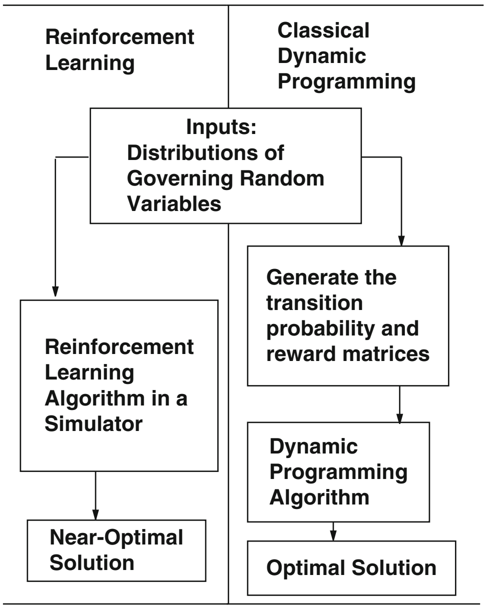
Reinforcement Learning
Classical Dynamic Programming
Inputs: Distributions of Governing Random Variables
Reinforcement Learning Algorithm in a Simulator
Near-Optimal Solution
Generate the transition probability and reward matrices
Dynamic Programming Algorithm
Optimal Solution

Figure 7.1. A schematic highlighting the differences in the methodologies of RL and DP

The main difference between the RL and DP philosophies has been depicted in Fig. 7.1. Like DP, RL needs the distributions of the random variables that govern the system's behavior. In DP, the first step is to generate the TPMs and TRMs, and the next step is to use these matrices in a suitable algorithm to generate a solution. In RL, we do not estimate the TPM or the TRM but instead simulate the system using the distributions of the governing random variables. Then, within the simulator, a suitable algorithm (clearly different than the DP algorithm) is used to obtain a solution.

In what follows in this section, we discuss some fundamental RL-related concepts. A great deal of RL theory is based on the Q-factor, the Robbins-Monro algorithm [247], and step sizes, and on how these ideas come together to help solve MDPs and SMDPs within simulators. Our discussion in this chapter will be geared towards helping build an intuitive understanding of the algorithms in RL. Hence, we will derive the algorithms from their DP counterparts to strengthen our intuition. More sophisticated arguments of convergence and existence will be dealt with later (in Chap. 11).

For the reader's convenience, we now define the following sets again:

1. $\mathcal{S}$: the set of states in the system. These states are those in which decisions are made. In other words, $\mathcal{S}$ denotes the set of decision-making states. Unless otherwise specified, in this book, a state will mean the same thing as a *decision-making* state.

2. $\mathcal{A}(i)$: the set of actions allowed in state i.

Both types of sets, $\mathcal{S}$ and $\mathcal{A}(i)$ (for all $i \in \mathcal{S}$), will be assumed to be finite in our discussion. We will also assume that the Markov chain associated with every policy in the MDP (or the SMDP) is regular. Please review the previous chapter for a definition of regularity.

3.1. Q-Factors

RL algorithms (for the most part) use the value function of DP. In RL, the value function is stored in the form of the so-called Q-factors. Recall the definition of the value function associated with the optimal policy for discounted reward MDPs. It should also be recalled that this value function is defined by the Bellman **optimality** equation, which we restate here:

$$J^*(i) = \max_{a \in \mathcal{A}(i)} \left(\sum_{j=1}^{|\mathcal{S}|} p(i,a,j) \left[r(i,a,j) + \lambda J^*(j) \right] \right) \text{ for all } i \in \mathcal{S}, \text{ where} \tag{7.1}$$

- $J^*(i)$ denotes the ith element of the value function vector associated with the optimal policy

- $\mathcal{A}(i)$ is the set of actions allowed in state i
- $p(i, a, j)$ denotes the one-step transition probability of going from state i to state j under the influence of action a
- $r(i, a, j)$ denotes the immediate reward earned in state i when action a is selected in it and the system transitions to state j as a result
- $\mathcal{S}$ denotes the set of states in the Markov chain
- λ stands for the discounting factor

In DP, we associate one element of the value function vector with a given state. In RL, we associate one element of the so-called Q-factor vector with a given *state-action* pair. To understand this idea, consider an MDP with three states and two actions allowed in each state. In DP, the value function vector $\vec{J}^*$ would have three elements as shown below.

$$\vec{J}^* = (J^*(1), J^*(2), J^*(3)) .$$

In RL, we would have six Q-factors because there are six state-action pairs. Thus if $Q(i, a)$ denotes the Q-factor associated with state i and action a, then

$$\vec{Q} = (Q(1, 1), Q(1, 2), Q(2, 1), Q(2, 2), Q(3, 1), Q(3, 2)) .$$

Now, we present the important definition of a Q-factor (also called Q-value or the state-action value). For a state-action pair (i, a),

$$Q(i, a) = \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) \left[r(i, a, j) + \lambda J^*(j) \right] . \tag{7.2}$$

Now, combining Eqs. (7.1) and (7.2), one has that

$$J^*(i) = \max_{a \in \mathcal{A}(i)} Q(i, a). \tag{7.3}$$

The above establishes the important relationship between the value function of a state and the Q-factors associated with the state. Then, it should be clear that, if the Q-factors are known, one can obtain the value function of a given state from Eq. (7.3). For instance, for a state i with two actions if the two Q-factors are $Q(i, 1) = 95$ and $Q(i, 2) = 100$, then

$$J^*(i) = \max\{95, 100\} = 100.$$

Using Eq. (7.3), Eq. (7.2) can be written as:

$$Q(i,a) = \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) \left[r(i,a,j) + \lambda \max_{b \in \mathcal{A}(j)} Q(j,b) \right]. \tag{7.4}$$

for all (i,a). Equation (7.4) is an extremely important equation. It can be viewed as the Q-factor version of the Bellman optimality equation for discounted reward MDPs. This equation leads us to a Q-factor version of value iteration. This version can be used to determine the optimal Q-factors for a given state and forms the Q-factor counterpart of the value iteration algorithm of DP. We now present its step-by-step details.

3.2. Q-Factor Value Iteration

It is important to note that we are still very much in the DP arena; i.e., we have not made the transition to RL yet, and the algorithm we are about to present is completely equivalent to the regular value iteration algorithm of the previous chapter.

Step 1: Set $k = 1$, specify $\epsilon > 0$, and select arbitrary values for the vector $\vec{Q}^0$, e.g., set for all $i \in \mathcal{S}$ and $a \in \mathcal{A}(i)$, $Q^0(i,a) = 0$.

Step 2: For each $i \in \mathcal{S}$ and $a \in \mathcal{A}(i)$, compute:

$$Q^{k+1}(i,a) \leftarrow \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) \left[r(i,a,j) + \lambda \max_{b \in \mathcal{A}(j)} Q^k(j,b) \right].$$

Step 3: Calculate for each $i \in \mathcal{S}$:

$$J^{k+1}(i) = \max_{a \in \mathcal{A}(i)} Q^{k+1}(i,a) \text{ and } J^k(i) = \max_{a \in \mathcal{A}(i)} Q^k(i,a).$$

Then, if $||(\vec{J}^{k+1} - \vec{J}^k)|| < \epsilon(1-\lambda)/2\lambda$, go to Step 4. Otherwise increase k by 1, and return to Step 2.

Step 4: For each $i \in \mathcal{S}$, choose $d(i) \in \arg\max_{b \in \mathcal{A}(i)} Q(i,b)$, where $\hat{d}$ denotes the ϵ-optimal policy, and stop.

It should be noted that the updating rule in Step 2 is derived from Eq. (7.4). The equivalence of this algorithm to regular value iteration, which estimates the value function, is then easy to see. Instead of estimating the value function, this algorithm estimates the

Q-factors; we call the algorithm: Q-factor value iteration. The conceptual significance of this algorithm needs to be emphasized because we will derive RL algorithms from it. In RL, like the algorithm above, we estimate Q-factors, but the updating rule is different than the one shown in Step 2. Now, to make the transition to RL, we need to introduce the Robbins-Monro algorithm.

3.3. Robbins-Monro Algorithm

The Robbins-Monro algorithm is a widely-used and popular algorithm invented in the 1950s [247] that can help us estimate the mean of a random variable from its samples. The idea underlying it is very simple. We know that the mean of a random variable can be estimated from the samples of the random variable by using a straightforward averaging process. Let us denote the ith independent sample of a random variable X by x_i and the expected value (mean) by $\mathsf{E}(X)$. Then with probability 1, the estimate produced by

$$\frac{\sum_{i=1}^{k} x_i}{k}$$

tends to the real value of the mean as $k \to \infty$. (This follows from the strong law of large numbers (Theorem 2.1)). In other words, with probability 1,

$$\mathsf{E}[X] = \lim_{k\to\infty} \frac{\sum_{i=1}^{k} x_i}{k}.$$

The samples, it should be understood, can be generated in a simulator. Now, we will derive the Robbins-Monro algorithm from this simple averaging process. Let us denote the estimate of X in the kth iteration—i.e., after k samples have been obtained—by X^k. Thus:

$$X^k = \frac{\sum_{i=1}^{k} x_i}{k}. \tag{7.5}$$

$$\begin{aligned}
\text{Now, } X^{k+1} \quad &= \quad \frac{\sum_{i=1}^{k+1} x_i}{k+1} \\
&= \quad \frac{\sum_{i=1}^{k} x_i + x_{k+1}}{k+1} \\
&= \quad \frac{X^k k + x_{k+1}}{k+1} \text{ (using Eq. (7.5))} \\
&= \quad \frac{X^k k + X^k - X^k + x_{k+1}}{k+1}
\end{aligned}$$

$$\begin{aligned}
&= \frac{X^k(k+1) - X^k + x_{k+1}}{k+1} \\
&= \frac{X^k(k+1)}{k+1} - \frac{X^k}{k+1} + \frac{x_{k+1}}{k+1} \\
&= X^k - \frac{X^k}{k+1} + \frac{x_{k+1}}{k+1} \\
&= \left(1 - \alpha^{k+1}\right) X^k + \alpha^{k+1} x_{k+1} \text{ if } \alpha^{k+1} = 1/(k+1),
\end{aligned}$$

$$\text{i.e., } X^{k+1} = \left(1 - \alpha^{k+1}\right) X^k + \alpha^{k+1} x_{k+1}. \tag{7.6}$$

The above is called the Robbins-Monro algorithm. When $\alpha^{k+1} = 1/(k+1)$, it should be clear from the above that the Robbins-Monro algorithm is equivalent to direct averaging. However, rules other than $1/(k+1)$ are more commonly used for α, also called *step size* or the *learning rate* (or learn rate). We will discuss a number of step-size rules below. Further, note that we will use the Robbins-Monro algorithm extensively in RL.

3.4. Robbins-Monro and Q-Factors

It turns out that the Robbins-Monro algorithm can in fact be used to estimate Q-factors within simulators. Recall that value iteration seeks to estimate the optimal value function. Similarly, from our discussion on the Q-factor version of value iteration, it should be obvious that the Q-factor version of value iteration should seek to estimate the optimal Q-factors.

It can be shown that every Q-factor can be expressed as an average of a random variable. Recall the definition of the Q-factor in the Bellman equation form (that is Eq. (7.4)):

$$Q(i,a) = \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) \left[r(i,a,j) + \lambda \max_{b \in \mathcal{A}(j)} Q(j,b) \right] \tag{7.7}$$

$$= \mathsf{E} \left[r(i,a,j) + \lambda \max_{b \in \mathcal{A}(j)} Q(j,b) \right] \tag{7.8}$$

$$= \mathsf{E}[\text{SAMPLE}], \tag{7.9}$$

where the quantity in the square brackets of (7.8) is the random variable of which $\mathsf{E}[\cdot]$ is the expectation operator. Thus, if samples of the random variable can be generated within a simulator, it is possible to use the Robbins-Monro scheme for evaluating the Q-factor.

Instead of using Eq. (7.7) to estimate the Q-factors (as shown in the Q-factor version of value iteration), we could instead use the Robbins-Monro scheme in a simulator. Using the Robbins-Monro algorithm (see Eq. (7.6)), Eq. (7.7) becomes:

$$Q^{k+1}(i,a) \leftarrow (1-\alpha^{k+1})Q^k(i,a) + \alpha^{k+1}\left[r(i,a,j) + \lambda \max_{b\in\mathcal{A}(j)} Q^k(j,b)\right] \tag{7.10}$$

for each (i,a) pair.

Perhaps, the most exciting feature of the above is that it is devoid of the transition probabilities! I.e., we do not need to know the transition probabilities of the underlying Markov chain in order to use the above in an algorithm. All we will need is a simulator of the system. Thus, the mechanism shown in (7.10) enables us to avoid transition probabilities in RL. An algorithm that does not use (or need) transition probabilities in its updating equations is called a **model-free** algorithm.

What we have derived above is the main update in the Q-learning algorithm for discounted MDPs. This was first invented by Watkins [312]. The above discussion was provided to show that the Q-Learning algorithm can be derived from the Bellman **optimality** equation for discounted reward MDPs.

3.5. Asynchronous Updating and Step Sizes

Using the Q-learning algorithm in a simulator creates a few difficulties. Since there is more than one state, it becomes necessary to simulate a trajectory of the Markov chain within a simulator. Now, in regular DP (or in the Q-factor version of value iteration), one proceeds in a synchronous manner for updating the value function (or the Q-factors). For instance, if there are three states, one updates the value function (or the Q-factors) associated with state 1, then that of state 2, then that of state 3, then returns to state 1 and so on. In a simulator, it is difficult to guarantee that the trajectory is of a nice $1, 2, 3, 1, 2, 3, \ldots$ form. Instead the trajectory can take the following form:

$$1, 3, 1, 3, 2, 3, 2, 2, 1, 3, 3, 1 \ldots$$

The above is just one example of what a trajectory could look like. Along this trajectory, updating occurs in the following way. After updating some Q-factor in state 1, we visit state 3, update a Q-factor there and return to state 1, then update a Q-factor in state 1, and so on. When a Q-factor of a state is updated, the updating equation usually needs Q-factors from some other state, and the latest estimate of these Q-factors from the other state need to be used in this scenario. It is to be understood that in such a *haphazard* style of updating, at any given time, at any given time, it is usually the case the different Q-factors get updated with differing frequencies.

Updating in this style is called **asynchronous** updating. In asynchronous updating, the Q-factors used within an update may not have been updated in the past with the same frequency. Fortunately, it can be shown that under suitable conditions on the step size and the algorithm, even with asynchronism, the algorithm can produce an optimal solution.

Now, thus far, we have defined the step size as

$$\alpha^k = 1/k$$

where k is the number of samples generated. A prime difficult with this rule is that $1/k$ decays rapidly with the value of k, and in only a few iterations, the step size can become too small to perform any updating. Hence, other step-size rules have been suggested in the literature. One such rule, given in Darken et al. [71], is:

$$\alpha^{k+1} = \frac{T_1}{1 + \frac{k^2}{1+T_2}}, \qquad (7.11)$$

where T_1 is the starting value for α and k^2 denotes k raised to 2. Possible values for T_1 and T_2 are 0.1 and 10^6 respectively. Another popular (simpler) rule of which a special case is the $1/k$ rule is:

$$\alpha^k = A/(B + k)$$

with e.g., $A = 5$ and $B = 10$. With suitable values for scalars A and B (the tuning parameters), this rule does not decay as fast as $1/k$ and can potentially work well. However, for it to work well, it is necessary to determine suitable values for A and B. Finally, a rule that does not have any scalar parameters to be tuned and does not decay as fast as $1/k$ is the following log-rule:

$$\alpha^k = \frac{\log(k)}{k},$$

where log denotes the natural logarithm. Experiments with these rules on small problems have been conducted in Gosavi [113].

It is necessary to point out that to obtain convergence to optimal solutions, it is essential that the step sizes follow a set of conditions.

Two well-known conditions from this set are:

$$\sum_{k=1}^{\infty} \alpha^k = \infty \text{ and } \sum_{k=1}^{\infty} \left(\alpha^k\right)^2 < \infty.$$

Some of the other conditions needed are more technical, and the reader is referred to [46]. Fortunately, the $A/(B+k)$ rule and the log rule satisfy the two conditions specified above and those in [46]. Under the standard mathematical analysis in the literature, convergence of the algorithm to optimality can be ensured when all of these conditions are satisfied by the step sizes. We will assume in this chapter that the step sizes will be updated using one of the rules documented above.

A constant is an attractive choice for the step size since it may ensure rapid convergence if its value is close to 1. However, a constant step size violates the second condition above, i.e., $\sum_{k=1}^{\infty} \left(\alpha^k\right)^2 < \infty$, which is why in RL, constant step sizes are usually not used.

4. MDPs

In this section, we will discuss simulation-based RL algorithms for MDPs in detail. This section forms the heart of this chapter, and indeed of the part of the book devoted to control optimization. We will build upon the ideas developed in the previous section. We will first consider discounted reward and then average reward.

4.1. Discounted Reward

Our discussion here is based on value and policy iteration. The first subsection will focus on value iteration and the second will focus on policy iteration.

4.1.1 Value Iteration

As discussed previously, value iteration based RL algorithms are centered on computation of the optimal Q-factors. The Q-factors in value iteration are based on the Bellman equation.

A quick overview of "how to incorporate an RL optimizer within a simulator" is in order at this point. We will do this with the help of an example.

Consider a system with two Markov states and two actions allowed in each state. This will mean that there are four Q-factors that need to be evaluated:

$$Q(1,1), Q(1,2), Q(2,1), \text{ and } Q(2,2),$$

where $Q(i, a)$ denotes the Q-factor associated with the state-action pair (i, a). The bottom line here is that we want to estimate the values of these Q-factors from running a simulation of the system. For the estimation to be perfect, we must obtain an infinite number of samples for each Q-factor. For this to happen, each state-action pair should be, theoretically, tried infinitely often. A safe strategy to attain this goal is to try each action in each state with equal probability and simulate the system in a fashion such that each state-action pair is tried a large number of times. However, other strategies are also acceptable as long as each state-action pair is tried infinitely often.

The job of the RL algorithm, which should be embedded within the simulator, is to update the values of the Q-factors using the equation given in (7.10). The simulator moves the system from one state to another selecting each action with equal probability in each state. *The simulator is not concerned with what the RL algorithm does* as long as each action is selected infinitely often. We now discuss the scheme used in updating the Q-factors in a more technical manner.

Let us assume (consider Fig. 7.2) that the simulator selects action a in state i and that the system goes to state j as a result of the action. During the time interval in which the simulator goes from state i to state j, the RL algorithm collects information from within the simulator; the information is $r(i, a, j)$, which is the immediate reward earned in going from state i to state j under the influence of action a.

When the simulator reaches state j, it uses $r(i, a, j)$ to generate a new sample of $Q(i, a)$. See Eq. (7.8) to see what is meant by a sample of $Q(i, a)$ and why $r(i, a, j)$, among other quantities, is needed to find the value of the sample. Then $Q(i, a)$ is updated via Eq. (7.10), with the help of the new sample. Thus, the updating for a state-action pair is done *after* the transition to the next state. This means that the simulator should be written in a style that permits updating of quantities after each state transition.

In what follows, we will present a step-by-step technical account of simulation-based value iteration, which is also called Q-learning.

Steps in Q-Learning.

Step 1. Initialize the Q-factors. In other words, set for all (l, u) where $l \in \mathcal{S}$ and $u \in \mathcal{A}(l)$: $Q(l, u) = 0$. Set k, the number of state transitions, to 0. We will run the algorithm for $k_{\max}$ iterations, where $k_{\max}$ is chosen to be a sufficiently large number. Start system simulation at any arbitrary state.

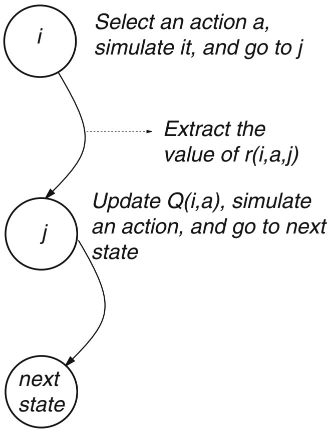
i
Select an action a, simulate it, and go to j
Extract the value of r(i,a,j)
Update Q(i,a), simulate an action, and go to next state
j
next state

Figure 7.2. The updating of the Q-factors in a simulator: Each *arrow* denotes a state transition in the simulator. After going to state j, the Q-factor for the previous state i and the action a selected in i, that is, $Q(i, a)$, is updated

Step 2. Let the current state be i. Select action a with a probability of $1/|\mathcal{A}(i)|$.

Step 3. Simulate action a. Let the next state be j. Let $r(i, a, j)$ be the immediate reward earned in the transition to state j from state i under the influence of action a. The quantity $r(i, a, j)$ will be determined by the simulator. Increment k by 1. Then update α suitably, using one of the rules discussed above (see Sect. 3.5).

Step 4. Update $Q(i, a)$ using the following equation:

$$Q(i,a) \leftarrow (1-\alpha)Q(i,a) + \alpha \left[r(i,a,j) + \lambda \max_{b \in \mathcal{A}(j)} Q(j,b) \right].$$

Step 5. If $k < k_{\max}$, set $i \leftarrow j$, and go to Step 2. Otherwise, go to Step 6.

Step 6. For each $l \in \mathcal{S}$, select

$$d(l) \in \arg\max_{b \in \mathcal{A}(l)} Q(l,b).$$

The policy (solution) generated by the algorithm is $\hat{d}$. Stop.

Please make note of the following.

1. *Notation:* The term $\max_{b \in \mathcal{A}(j)} Q(j, b)$ in the main updating equation for the Q-factors represents the maximum Q-factor in state j.

Also, $\arg\max_{b\in\mathcal{A}(j)} Q(j,b)$ denotes the action associated with the maximum Q-factor in state j.

2. *Look-up tables:* The algorithm presented above is for the so-called **look-up table** implementation in which all Q-factors are stored explicitly in the computer's memory. This implementation is possible only when we have a manageable number of state-action pairs, e.g., up to 10,000. All the RL algorithms that appear in the following pages (unless specified otherwise) will be presented in a look-up table format. We will discuss function approximation, in which all the Q-factors are not stored explicitly, in Sect. 8.

RL: A Learning Perspective. RL was developed by researchers in the artificial intelligence (AI) community. In the AI community, RL was viewed as a machine "learning" technique. We now explain this viewpoint.

We can view the decision maker in an MDP as a *learning agent.* The "task" assigned to this agent is to obtain the optimal action in each state of the system. The operation of an RL algorithm can be described in the following manner. The algorithm starts with the same value for each Q-factor associated with a given state. All possible actions are simulated in every state. *The actions that produce good immediate rewards are rewarded and those that produce poor rewards are punished.* The agent accomplishes this in the following manner. For any given state i, it *raises* the values of the Q-factors of the good actions and *diminishes* the values of the Q-factors of the bad actions.

We have viewed (earlier) the updating process in RL as the use of the Robbins-Monro algorithm for the Q-factor, and the definition of the Q-factor was obtained from the Bellman equation. Although the two viewpoints are essentially equivalent, it is considered instructive to understand this intuitive idea underlying RL; the idea is to reward the good actions, punish the bad actions, and in general **learn** from trial-and-error in the simulator. See Fig. 7.3 for a schematic.

Trial and error is a fundamental notion underlying any *learning* algorithm. We also note that "learning" is used to describe any kind of function approximation, e.g., regression, in the machine learning community. Since function approximation is usually an integral part of any application of RL on a large-scale problem, RL is also called a "learning" technique.

It is invariably interesting to determine whether the learning agent can learn (improve its behavior) within the simulator *during the run time of the simulation.* Since the discounted reward problem has a

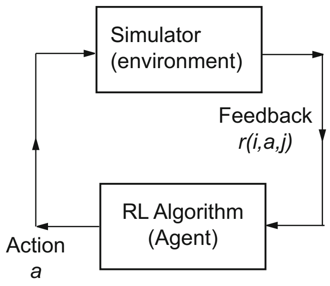

Figure 7.3. Trial and error mechanism of RL: The action selected by the RL agent (algorithm) is fed into the simulator. The simulator simulates the action, and the resultant feedback (immediate reward) obtained is fed back into the knowledge-base (Q-factors) of the agent. The agent uses the RL algorithm to update its knowledge-base, becomes smarter in the process, and then selects a better action

large number of performance metrics (the value function of each state is a performance metric for the discounted MDP), it is difficult to determine a trend when the simulation is running. In the average reward case, however, this can be done since the performance metric is unique (the average reward).

The performance of an algorithm is, however, best judged **after** the learning. To evaluate the performance, as mentioned above, we can re-simulate the system using the policy learned. This, of course, is done after the learning phase is complete in the phase called the *frozen* phase. The word "frozen" refers to the fact that the Q-factors do not change during this phase.

On-Line and Off-Line. The word "learning" makes a great deal of sense in robotic problems [299]. In these problems, the robot improves its behavior on a real-time basis like a human. For numerous reasons, in the field of robotics, learning in simulators is not considered to be as effective as learning in real time. In most problems that we will be considering in this book, it will be sufficient to learn in simulators. This is primarily because we will assume that the distributions of the random variables that govern the system's behavior are available, i.e., we can simulate the system, and hence there is no need to learn on a real-time basis.

In this book for the most part, we will work within a simulator before implementing the solution in the real-life system. As a result, the word "learning" may not apply to problems for which good simulators are available. In industrial problems (in the manufacturing and service industry), learning in real time can be extraordinarily costly, and in fact, simulations are preferred because they do not disturb the actual system.

Within the machine learning community, RL algorithms are often described as either being on-line or off-line *algorithms*—causing a great deal of confusion to the beginner. Algorithms can be run off-line or on-line with respect to separate qualifiers, which are: (1) the implementation of the algorithm on an application and (2) the internal updating mechanism of the algorithm.

The implementation aspect: In an off-line *implementation*, the algorithm is run within the simulator before the solution is implemented on the real system. On the other hand, in an on-line *implementation*, the algorithm does *not* use a simulator, but rather operates on a real-time basis within the real system. Now, all the model-free algorithms that we will discuss in this book can be implemented in either an off-line or an on-line sense.

The updating aspect: This is an internal issue of the algorithm. Most algorithms discussed in this book update the Q-factor of a state-action pair *immediately* after it is tried. Such an update is called an on-line *update*. Now, there are some algorithms that update Q-factors *after a finite number of state transitions*, e.g., some so-called TD($\bar{\lambda}$) (where $\bar{\lambda}$ is a decaying factor) algorithms work in this fashion. Examples of these can be found in [285, 224, 123]. This kind of updating is often referred to as an off-line *update* and has nothing to do with an off-line *implementation*.

An algorithm with an on-line updating mechanism may have an off-line implementation. All four combinations are possible. In this book, we will for the most part focus on the algorithms that are implemented off-line and updated on-line. We must point out that algorithms that require off-line updates are less elegant, since from the perspective of writing the computer code, they require the storage of values from multiple previous decision instants. However, this is only a programming aspect that carries no mathematical significance.

Exploration. In general, any action-selection strategy can be described as follows: When in state i, select action u with a probability of $\mathsf{p}^k(i)$, where k denotes the number of state transitions that have occurred thus far.

If $\mathsf{p}^k(i) = 1/|\mathcal{A}(i)|$, one obtains the strategy that we have discussed all along, one in which every action is tried with the same probability.

In the Robbins-Monro algorithm, for the averaging process in the Robbins-Monro algorithm to take place, each Q-factor must be tried infinitely often (theoretically); one way to ensure this is to select each action with equal probability, i.e., $\mathsf{p}^k(i) = 1/|\mathcal{A}(i)|$ for all i. Although this strategy is quite robust, it can be quite time-consuming in practice. Hence, other strategies have been suggested in the literature. We now discuss one such strategy, namely the *exploratory* strategy.
Exploratory strategy: In the exploratory strategy, one selects the action that has the highest Q-factor with a high probability and the other actions with low, non-zero probabilities. The action(s) that has the highest Q-factor is called the **greedy** action. The other actions are called the **non-greedy** actions. Consider a scenario with two actions. In the kth iteration of the algorithm, select an action u in state i with probability p^k, where

$$u = \arg\max_{b \in \mathcal{A}(i)} Q(i,b) \text{ and } \mathsf{p}^k = 1 - \frac{B^k}{k}, \tag{7.12}$$

and select the other action with probability B^k/k, where e.g., $B^k = 0.5$ for all k. It is clear that when such an exploratory strategy is pursued, the learning agent will select the non-greedy (exploratory) action with probability B^k/k. As k starts becoming large, the probability of selecting the non-greedy (exploratory) action diminishes. At the end, when k is very large, the action selected will clearly be a greedy action. As such, at the end, the action selected will also be the action prescribed by the policy learned by the algorithm. Another example for decaying the exploration would be as follows:

$\mathsf{p}^k = \mathsf{p}^{k-1}A$ with $A < 1$, or via any scheme for decreasing step sizes.

GLIE strategy: We now discuss an important class of exploration

that is required for some algorithms such as SARSA and R-SMART. In the so-called GLIE (greedy in the limit with infinite exploration) policy [277], the exploration is reduced in a manner such that in the limit, one obtains a greedy policy and still all the state-action pairs are visited infinitely often. An example of such a policy is one that would use the scheme in (7.12) with $B^k = A/\bar{V}^k(i)$ where $0 < A < 1$ and $\bar{V}^k(i)$ denotes the number of times state i has been visited thus far in the simulator. That a policy of this nature satisfies the GLIE property can be shown (see [277] for proof and other such schemes). It is important to note, however, that this specific example of a GLIE policy will need keeping track of an additional variable, $\bar{V}^k(i)$, for each state.

One potential benefit of using an exploratory action-selection strategy is that the run time of the algorithm can be reduced. This is because as k increases, the non-greedy Q-factors are evaluated with *decreasing* frequencies. One hopes that with time the non-greedy Q-factors are not optimal, and that their "perfect" evaluation does not serve any purpose. Note, however, that this may lead to an imperfect evaluation of many Q-factors, unless we use GLIE exploration, and this in turn can cause the solution to deviate from optimality (after all, the values of the Q-factors are inter-dependent via the Bellman equation). In addition, theoretically, any action selection strategy should ensure that each Q-factor is updated infinitely often. This is essential, because for the Robbins-Monro scheme to work (i.e., averaging to occur), an infinitely large number of samples must be collected.

A Worked-Out Example for Q-Learning. We next show some sample calculations performed in a simulator using Q-Learning. We will use Example A (see Sect. 3.3.2 of Chap. 6). We repeat the problem details below. The TPM associated with action a is $\mathbf{P}_a$ and the associated TRM is $\mathbf{R}_a$.

$$\mathbf{P}_1 = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix}; \mathbf{P}_2 = \begin{bmatrix} 0.9 & 01 \\ 0.2 & 0.8 \end{bmatrix}; \mathbf{R}_1 = \begin{bmatrix} 6 & -5 \\ 7 & 12 \end{bmatrix}; \mathbf{R}_2 = \begin{bmatrix} 10 & 17 \\ -14 & 13 \end{bmatrix}.$$

The performance metric is discounted reward with $\lambda = 0.8$. In what follows, we show step-by-step calculations of Q-Learning along one trajectory. Let us assume that the system starts in state 1.

State 1. Set all the Q-factors to 0:

$$Q(1,1) = Q(1,2) = Q(2,1) = Q(2,2) = 0.$$

The set of actions allowed in state 1 is $\mathcal{A}(1) = \{1,2\}$ and that allowed in state 2 is $\mathcal{A}(2) = \{1,2\}$. Clearly $|\mathcal{A}(i)| = 2$ for $i = 1,2$. Let the step size α be defined by $A/(B+k)$, where $A = 5$, $B = 10$, and k denotes the number of state transitions.

Select an action with probability $1/|\mathcal{A}(1)|$. Let the selected action be 1. Simulate action 1. Let the next state be 2.

State 2. The current state (j) is 2 and the old state (i) was 1. The action (a) selected in the old state was 1. So we now have to update $Q(1,1)$. Now: $k = 0; \alpha = 5/10 = 0.5$.

$$r(i,a,j) = r(1,1,2) = -5;$$

$$\max_b Q(j,b) = \max_b Q(2,b) = \max\{0,0\} = 0;$$

$$\begin{aligned} Q(1,1) &\leftarrow (1-\alpha)Q(1,1) + \alpha[-5+\lambda(0)] \\ &= 0.5(0) + 0.5[-5+0] = -2.5. \end{aligned}$$

Select an action with probability $1/|\mathcal{A}(2)|$. Let the selected action be 2. Simulate action 2. Let the next state be 2.

State 2 (again). The current state (j) is 2 and the old state (i) was also 2. The action (a) selected in the old state was 2. So we now have to update $Q(2,2)$. Now: $k = 1; \alpha = 5/(10+1) = 0.454$.

$$r(i,a,j) = r(2,2,2) = 13;$$

$$\max_b Q(j,b) = \max_b Q(2,b) = \max\{0,0\} = 0;$$

$$\begin{aligned} Q(2,2) &\leftarrow (1-\alpha)Q(2,2) + \alpha[13+\lambda(0)] \\ &= 0.546(0) + 0.454[13] = 5.902. \end{aligned}$$

Select an action with probability $1/|\mathcal{A}(2)|$. Let the selected action be 1. Simulate action 1. Let the next state be 1.

State 1 (again). The current state (j) is 1 and the old state (i) was 2. The action (a) selected in the old state was 1. So we now have to update $Q(2,1)$. Now: $k = 2; \alpha = 5/(10+2) = 0.416$.

$$r(i,a,j) = r(2,1,1) = 7;$$

$$\max_b Q(j,b) = \max_b Q(1,b) = \max\{-2.5,0\} = 0;$$

$$\begin{aligned} Q(2,1) &\leftarrow (1-\alpha)Q(2,1) + \alpha[7+\lambda(0)] \\ &= (1-0.416)0 + 0.416[7] = 2.912. \end{aligned}$$

Select an action with probability $1/|\mathcal{A}(1)|$. Let the selected action be 2. Simulate action 2. Let the next state be 2.

State 2 (a third time). The current state (j) is 2 and the old state (i) was 1. The action (a) selected in the old state was 2. So we now have to update $Q(1,2)$. Now: $k = 3; \alpha = 5/(10+3) = 0.386$.

$$r(i,a,j) = r(1,2,2) = 17;$$

$$\max_b Q(j,b) = \max_b Q(2,b) = \max\{2.912, 5.902\} = 5.902;$$

$$\begin{aligned} Q(1,2) &\leftarrow (1-\alpha)Q(1,2) + \alpha[17 + \lambda(1.3)] \\ &= (1-0.386)(0) + 0.386[17 + \lambda 5.902] = 8.384. \end{aligned}$$

Select an action with probability $1/|\mathcal{A}(2)|$. Let the selected action be 2. Simulate action 2. Let the next state be 1.

State 1 (a third time). The current state (j) is 1 and the old state (i) was 2. The action (a) selected in the old state was 2. So we now have to update $Q(2,2)$. Now: $k = 4; \alpha = 5/(10+4) = 0.357$;

$$r(i,a,j) = r(2,2,1) = -14;$$

$$\max_b Q(j,b) = \max_b Q(1,b) = \max\{-2.5, 8.384\} = 8.384;$$

Then $Q(2,2) \leftarrow (1-0.357)5.902+0.357[-14+\lambda(8.384)]=1.191$,

and so on.

Table 7.2. The table shows Q-factors for Example A with a number of different step-size rules. Here $\mathsf{p}^k = 1/|\mathcal{A}(i)|$

Method	$Q(1,1)$	$Q(1,2)$	$Q(2,1)$	$Q(2,2)$
Q-factor-value iteration	44.84	53.02	51.87	49.28
Q-Learning with $\alpha = 150/(300+k)$	44.40	52.97	51.84	46.63
Q-Learning with $\alpha = 1/k$	11.46	18.74	19.62	16.52
Q-Learning with $\alpha = log(k)/k$	39.24	47.79	45.26	42.24
Q-Learning with $\alpha(i,a) = 1/V(i,a)$	38.78	49.25	44.64	41.45

Step-Size Rules and Their Impact. We now present the results of using a number of different step-size rules on the above problem and compare those to the results obtained from Q-factor value iteration (see Sect. 3.2) which are guaranteed to be optimal. It is clear that our aim is to produce results that approach those produced by Q-factor value iteration. Table 7.2 presents the numerical results, most of which are from [113].

An inspection of the results in Table 7.2 indicates that the step-size rule $\alpha = A/(B+k)$ produces the best results, since the Q-factor values produced by it are closest to the values obtained from Q-factor value iteration. Note that all the step-size rules produce the optimal policy: $(2, 1)$. The rule $\alpha = 1/k$ produces the optimal policy, but the values stray *considerably* from the optimal values generated by the Q-factor value iteration. This casts some doubt on the usefulness of the $1/k$ rule, which has been used by many researchers (including this author before [113] was written). Note that $1/k$ decays very quickly to 0, which can perhaps lead to computer round-off errors and can cause problems. One advantage of the $1/k$ rule (other than its theoretical guarantees) is that it does not have any tuning parameters, e.g., A and B. The log-rule ($log(k)/k$ starting at $k = 2$) has an advantage in that it does not need any tuning parameters, unlike the $A/(B+k)$, and yet produces values that are better than those of the $1/k$ rule. Finally, in the last row of the table, we present the results of an algorithm in which we use the rule $1/k$ separately for each state-action pair. In other words, a separate step-size is used for each $Q(i, a)$, and the step-size is defined as $1/V(i, a)$, where $V(i, a)$ is the number of times the state-action pair (i, a) was tried. This works well, as is clear from the table, but requires a separate $V(i, a)$ for each state-action pair. In other words, this rule increases the storage burden of the algorithm. In summary, it appears that the log-rule not only works well, but also does not need tuning of parameters.

4.1.2 Policy Iteration

In this section, we will pursue an RL approach based on policy iteration for solving discounted reward MDPs. Direct policy iteration is usually ruled out in the context of RL because in its policy evaluation phase, one has to solve linear equations, which requires the transition probabilities. Then, *modified* policy iteration (see Chap. 6) becomes worth pursuing because in it the policy evaluation phase does not require solving of any equations; in the policy evaluation phase of modified policy iteration, one performs a value iteration (to determine the value function vector of the policy being evaluated). Since value iteration can be done in a simulator using Q-factors, it is possible to derive a policy iteration algorithm based on simulation.

The idea underlying the usage of policy iteration in RL is quite simple. We start with an arbitrary policy; and then we evaluate the Q-factors associated with that policy in the simulator. Thereafter, we perform the policy improvement step, which leads to a new policy. Then we return to the simulator to evaluate the new policy. This continues until no improvement is obtained. We discuss this idea in more detail below.

Q-Factor Policy Iteration. We need to define a Q-factor here *in terms of the policy.* For a state-action pair (i,a), for a policy $\hat{\mu}$, where $a \in \mathcal{A}(i)$,

$$Q_{\hat{\mu}}(i,a) = \sum_{j=1}^{|\mathcal{S}|} p(i,a,j)\left[r(i,a,j) + \lambda J_{\hat{\mu}}(j)\right], \tag{7.13}$$

where $\vec{J}_{\hat{\mu}}$ is the value function vector associated with the policy $\hat{\mu}$. Notice the difference with the definition of the Q-factor in value iteration, which is:

$$Q(i,a) = \sum_{j=1}^{|\mathcal{S}|} p(i,a,j)\left[r(i,a,j) + \lambda J^*(j)\right], \tag{7.14}$$

where $\vec{J}^*$ denotes the value function vector associated with the *optimal* policy.

Using the definition in Eq. (7.13), we can develop a version of policy iteration in terms of Q-factors. Now, from the Bellman equation for a given policy $\hat{\mu}$, which is also called the Poisson equation, we have that:

$$J_{\hat{\mu}}(i) = \sum_{j=1}^{|\mathcal{S}|} p(i,\mu(i),j)\left[r(i,\mu(i),j) + \lambda J_{\hat{\mu}}(j)\right], \quad \forall i. \tag{7.15}$$

From Eqs. (7.15) and (7.13), one has that

$$J_{\hat{\mu}}(i) = Q_{\hat{\mu}}(i,\mu(i)), \quad \forall i. \tag{7.16}$$

Using Eq. (7.16), Eq. (7.13) can be written as:

$$Q_{\hat{\mu}}(i,a) = \sum_{j=1}^{|\mathcal{S}|} p(i,a,j)\left[r(i,a,j) + \lambda Q_{\hat{\mu}}(j,\mu(j))\right], \quad \forall i, a \in \mathcal{A}(i). \tag{7.17}$$

It is clear that Eq. (7.17) is a Q-factor version of the equation used in the policy evaluation phase of policy iteration and is thus useful in devising a Q-factor version of policy iteration, which we present next. It is a critical equation on which much of our subsequent analysis is based. It is in fact the Q-factor version of the Bellman policy equation for discounted reward.

Steps in the Q-Factor Version of Regular Policy Iteration.

Step 1: Set $k = 1$. Select a policy, $\hat{\mu}_k$, arbitrarily.

Step 2: For the policy, $\hat{\mu}_k$, obtain the values of $\vec{Q}_{\hat{\mu}_k}$ by solving the system of linear equations given below:

$$Q_{\hat{\mu}_k}(i,a)=\sum_{j=1}^{|\mathcal{S}|} p(i,a,j)\left[r(i,a,j)+\lambda Q_{\hat{\mu}_k}(j,\mu_k(j))\right] \text{ for all } (i,a)\text{-pairs.}$$

Step 3: Generate a new policy $\hat{\mu}_{k+1}$, using the following relation:

$$\mu_{k+1}(i) \in \arg\max_{u\in\mathcal{A}(i)} Q_{\hat{\mu}_k}(i,u).$$

If possible, set $\hat{\mu}_{k+1} = \hat{\mu}_k$.

Step 4: If the policy $\hat{\mu}_{k+1}$ is identical to policy $\hat{\mu}_k$, the algorithm terminates. Otherwise, set $k \leftarrow k + 1$, and go back to Step 2.

How Step 3 is equivalent to policy improvement in classical DP will be explained below in the context of the RL algorithm based on these ideas. Linear algebra methods, e.g., Gauss-Jordan elimination, are needed in Step 2 of the above algorithm. Instead, one can use the method of successive approximations (value iteration) to solve for the Q-factors. In such a method, one starts with arbitrary values for the Q-factors and then uses an updating scheme derived from the Bellman equation repeatedly until the Q-factors converge. Clearly, the updating scheme, based on the equation in Step 2, would be:

$$Q_{\hat{\mu}_k}(i,a) \leftarrow \sum_{j=1}^{|\mathcal{S}|} p(i,a,j)\left[r(i,a,j)+\lambda Q_{\hat{\mu}_k}(j,\mu_k(j))\right]. \tag{7.18}$$

But the Q-factor can be expressed as an expectation as shown below:

$$\begin{aligned} Q_{\hat{\mu}_k}(i,a) &= \mathsf{E}[r(i,a,j)+\lambda Q_{\hat{\mu}_k}(j,\mu_k(j))] \\ &= \mathsf{E}[\text{SAMPLE}]. \end{aligned} \tag{7.19}$$

Then, using the Robbins-Monro scheme (see (7.6)), Eq. (7.18) can be written, for all state-action pairs (i,a), as:

$$\begin{aligned} Q_{\hat{\mu}_k}(i,a) \quad &\leftarrow \quad (1-\alpha)Q_{\hat{\mu}_k}(i,a)+\alpha[\text{SAMPLE}] \\ &= \quad (1-\alpha)Q_{\hat{\mu}_k}(i,a)+\alpha[r(i,a,j)+\lambda Q_{\hat{\mu}_k}(j,\mu_k(j))], \end{aligned}$$

where the last relationship follows from Eq. (7.19). Thus:

$$Q_{\hat{\mu}_k}(i,a) \leftarrow (1-\alpha)Q_{\hat{\mu}_k}(i,a) + \alpha[r(i,a,j) + \lambda Q_{\hat{\mu}_k}(j,\mu_k(j))]. \quad (7.20)$$

What we have derived above is an RL scheme to evaluate the Q-factors associated with a policy within a simulator. We now discuss how policy iteration may be carried out in a simulator in an iterative style.

We begin with an arbitrary policy. Then we use a simulator to estimate the Q-factors associated with the policy. This, of course, is the policy evaluation phase. We refer to each policy evaluation as an **episode**. When an episode ends, we carry out the *policy improvement* step. In this step, we copy the Q-factors into a new vector called P-factors, and then destroy the old Q-factors. Thereafter a new episode is started to determine the new Q-factors associated with the new policy. The P-factors define the new policy to be evaluated and as such are needed in evaluating the new Q-factors.

Steps in Q-P-Learning. We will next present the step-by-step details of Q-P-Learning, which is essentially an RL algorithm based on policy iteration.

Step 1. Initialize all the P-factors, $P(l,u)$ for all $l \in \mathcal{S}$ and $u \in \mathcal{A}(l)$, to arbitrary values. Set k, the number of episodes, to 1. Let n denote the number of state transitions (iterations) within an episode. Initialize $k_{\max}$ and $n_{\max}$ to large numbers.

Step 2 (Policy Evaluation). Start fresh simulation. Set all the Q-factors, $Q(l,u)$, to 0. Let the current system state be i. Set n, the number of iterations within an episode, to 1.

Step 2a. Simulate action $a \in \mathcal{A}(i)$ with probability $1/|\mathcal{A}(i)|$.

Step 2b. Let the next state encountered in the simulator be j. Let $r(i,a,j)$ be the immediate reward earned in the transition from state i to state j. Update α. Then update $Q(i,a)$ using:

$$Q(i,a) \leftarrow (1-\alpha)Q(i,a) + \alpha\left[r(i,a,j) + \lambda Q\left(j, \arg\max_{b\in\mathcal{A}(j)} P(j,b)\right)\right]. \quad (7.21)$$

Step 2c. Increment n by 1. If $n < n_{\max}$, set $i \leftarrow j$ and return to Step 2a. Otherwise go to Step 3.

Step 3 (Policy Improvement). Set for all $l \in \mathcal{S}, u \in \mathcal{A}(l)$,

$$P(l,u) \leftarrow Q(l,u).$$

Then increment k by 1. If k equals $k_{\max}$, go to Step 4. Otherwise, go to Step 2.

Step 4. For each $l \in \mathcal{S}$, select $d(l) \in \arg\max_{b \in \mathcal{A}(l)} Q(l, b)$. The policy (solution) generated by the algorithm is $\hat{d}$. Stop.

A number of important remarks are in order here.

Remark 1: The update in Step 2b and that in (7.20) appear different on first glance, although they are supposed to be the same. The update in (7.20) was derived from the Q-factor version of the Bellman equation, via the Robbins Monro algorithm. We now show that the updates in Step 2b and (7.20) are equivalent. If the policy evaluated in Step 2b is $\hat{\mu}$, then

$$\mu(j) = \arg\max_{b \in \mathcal{A}(j)} P(j, b), \text{ from which it follows that:}$$

$$Q_{\hat{\mu}}\left(j, \arg\max_{b \in \mathcal{A}(j)} P(j, b)\right) = Q_{\hat{\mu}}(j, \mu(j)); \text{ and so the two are identical.}$$

We need to mention that a well-known class of algorithms in the literature goes by the name **approximate policy iteration** (API) [30, 31], and it is closely related to Q-P-Learning. The main equation that connects Q-P-Learning to API is in fact:

$$Q_{\hat{\mu}}(i, a) = \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) \left[r(i, a, j) + \lambda J_{\hat{\mu}}(j) \right]. \tag{7.22}$$

Note also that since, $Q_{\hat{\mu}}(j, \mu(j)) \equiv J_{\hat{\mu}}(j)$, we have that (7.22) above and the equation underlying Q-P-Learning, i.e., Eq. (7.17), are in fact the same. We will discuss a version of API below.

Remark 2: The policy improvement step in the policy iteration algorithm of classical DP is given by:

$$\mu_{k+1}(i) \in \arg\max_{a \in \mathcal{A}(i)} \left[\sum_{j \in \mathcal{S}} p(i, a, j) \left[r(i, a, j) + \lambda J_{\hat{\mu}_k}(j) \right] \right].$$

We can show that our policy improvement in Step 3 above is equivalent to this, as long as the Q-factors reach their correct values. In Q-P-Learning, the new policy is selected according to the P-factors, which are essentially the Q-factors of the previous episode, i.e., the policy in the newly generated Q-factors at the end of the episode is in fact the improved policy. Note that

$$
\begin{aligned}
\mu_{k+1}(i) \in \arg\max_{a \in \mathcal{A}(i)} & \left[\sum_{j \in \mathcal{S}} p(i,a,j) \left[r(i,a,j) + \lambda J_{\hat{\mu}_k}(j) \right] \right] \\
= \arg\max_{a \in \mathcal{A}(i)} & \left[\sum_{j \in \mathcal{S}} p(i,a,j) \left[r(i,a,j) + \lambda Q_{\hat{\mu}_k}(j, \mu_k(j)) \right] \right] \\
= \arg\max_{a \in \mathcal{A}(i)} & \left[Q_{\hat{\mu}_k}(i,a) \right] \qquad (7.23)
\end{aligned}
$$

Equation (7.23) follows from Eq. (7.17) as long as Q-P-Learning converges to the solution of Eq. (7.17). We thus showed that using the policy in the newly generated Q-factors is equivalent to performing the policy improvement step in classical DP.

Remark 3: In Q-P-Learning, exploration has to be carried out at its maximum rate, i.e., every action has to be tried with the same probability. Without this, there is danger of not identifying an improved policy at the end of an episode. Since the algorithm evaluates a given policy in one episode, it would be incorrect to bias the exploration in favor of that policy. This is because (i) the definition of the Q-factor does not have any such bias and (ii) with such a bias, we may never explore the actions that could potentially become optimal at a later stage; the latter can lead to sub-optimal values of the Q-factors. It is perhaps evident from Remark 2 that incorrect Q-factors, which could result from inadequate exploration, will corrupt the policy improvement step.

Remark 4: A word about the number of iterations within an episode, i.e., $n_{\max}$, is appropriate here. This quantity obviously needs to be large although finite. But how large should it be? Unless this number is large *enough*, inaccuracies in the Q-factors estimated in a given episode can lead to a new policy that is actually worse than the current policy. This phenomenon is part of what is called chattering (or oscillation), which can be undesirable [33]. Setting $n_{\max}$ to 1 or some other very small integer may cause severe chattering and has also been observed in the case of an algorithm called ambitious API [33].

Remark 5: In comparison to Q-Learning, Q-P-Learning will require additional time (especially with large values of $n_{\max}$), since in each episode, Q-P-Learning performs a Q-Learning-like evaluation. In other words, every episode resembles an independent application of Q-Learning. This difference in computational time may become especially acute in the look-up table case. However, policy iteration is known to be more stable with function approximation, which is needed when the number of state-action pairs is very large. Therefore, it appears that there is a significant amount of interest in algorithms based on the Bellman policy equation [31].

SARSA. The Modified Q-Learning algorithm of Rummery and Niranjan [258], also called SARSA [288, 277], is also based on policy iteration but uses an updating equation different than that of Q-P-Learning. Two versions of SARSA can be found in the literature. The first is from [288] (where it is said to follow generalized policy iteration (GPI) in [288]); we will call it the *episodic* version in which a policy (although exploratory) is evaluated within an episode. The exploration is gradually decayed to zero. The second version will be called *non-episodic* because it will not employ the notion of episodes; in this version also, one employs an exploratory policy whose exploration is gradually decayed to zero. The non-episodic version has been analyzed for convergence in [277].

In SARSA (both versions), in order to update a Q-factor of a state, one must not only know the action selected in that state but also the action selected in the next state. Thus, when an action a is selected in a state i and one transitions to state j as a result, one must know which state was visited immediately prior to i and which action was selected in that state. Let the state visited before i be s and let w be the action that was selected in s. Then, *after* we transition from i to j, we update $Q(s, w)$. Since the convergence properties of the non-episodic version are clearly understood, we restrict our discussion to the non-episodic version.

Non-episodic-SARSA. In this algorithm, we will not employ the notion of episodes, and thus the algorithm will resemble Q-Learning in how it functions, although it will still be based on the Bellman policy equation at the start.

Step 1. Initialize the Q-factors, $Q(l, u)$, for all (l, u) pairs. Let n denote the number of state transitions (iterations) of the algorithm. Initialize $n_{\max}$ to a large number. Start a simulation. Set $n = 1$.

Step 2. Let the current system state in the simulation be i. Set s to any state from $\mathcal{S}$, and set w to any action from $\mathcal{A}(s)$. Let $rimm = 0$.

Step 2a. Simulate action $a \in \mathcal{A}(i)$ in state i using an exploratory policy in which the exploration *must be* decayed gradually after every iteration.

Step 2b. Let the next state encountered in the simulator be j. Let $r(i,a,j)$ be the immediate reward earned in the transition from state i to state j. Update α. Then update $Q(s,w)$ using:

$$Q(s,w) \leftarrow Q(s,w) + \alpha \left[rimm + \lambda Q(i,a) - Q(s,w) \right] I(n \neq 1), \tag{7.24}$$

where $I(.)$, the indicator function, equals 1 when the condition inside the brackets is satisfied and 0 otherwise.

Step 2c. Set $w \leftarrow a$, $s \leftarrow i$, and $rimm \leftarrow r(i,a,j)$.

Step 2d. Increment n by 1. If $n < n_{\max}$, set $i \leftarrow j$ and return to Step 2a. Otherwise go to Step 3.

Step 3. For each $l \in \mathcal{S}$, select $d(l) \in \arg\max_{b \in \mathcal{A}(l)} Q(l,b)$. The policy (solution) generated by the algorithm is $\hat{d}$. Stop.

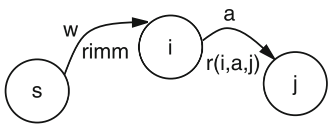

Figure 7.4. A schematic showing the updating in SARSA: The quantity above the *arrow* is the action selected and that below is the immediate reward earned in the transition. Note that we update $Q(s,w)$ after the transition from i to j is complete

A Note About the Delayed Updating in SARSA. Figure 7.4 is a schematic that should help you understand how the updating in SARSA works. It is important to realize that in SARSA, the updating of a Q-factor occurs after *two state transitions* have occurred (this is unlike the one-step updating we see in Q-Learning and in most algorithms in the book, where the updating occurs after *one state transition*). The indicator function in Eq. (7.24) ensures that in the first iteration ($n = 1$), no updating is performed. This is because the updating is performed two state transitions later, and the second state transition has not occurred yet. The notation via the special variables, s and w, allows us to express the update of a Q-factor that occurs after two state transitions.

It is worthwhile pointing out here that the exploration is decayed after every state transition and should tend to zero in the limit. This intuitively suggests that the algorithm update which is at the start based on a given policy (Bellman policy equation) tends to that based on the optimal policy (Bellman optimality equation).

SARSA and Q-P-Learning. One aspect in which SARSA (both epsidoic and non-episodic versions) differs significantly from Q-P-Learning is that in the latter, exploration is *never* decayed. The other major difference of SARSA (both versions) with Q-P-Learning is that the Q-factor of the next state in any update in SARSA is the Q-factor of the next state associated with the action selected by the current exploratory policy, while in Q-P-Learning, we use the Q-factor of the action defined by the policy being evaluated, i.e., the policy stored in the P-factors, which is fixed (and unchanged) within an episode.

CAP-I: Conservative Approximate Policy Iteration. We now present a member from a large class of algorithms loosely referred to in the literature as Approximate Policy Iteration (API). The specific algorithm that we present will be called Conservative API (CAP-I). The name "conservative" is used to distinguish it from its more well-known, but adventurous, cousin, which we call "ambitious" API. The *traditional* version of ambitious API (AAP-I) does not appear to have satisfactory convergence properties [33]. For a discussion on convergence properties of API in general, see Bertsekas and Tsitsiklis [33, pgs. 231-7] where ambitious API is called "partially optimistic TD(0)," used under the noisy conditions of a simulator. The discussion there also refers to a version called "optimistic TD(0)," which should not be confused with ambitious API; optimistic TD(0) is an algorithm of only academic interest, because it requires the transition probabilities and is not intended for simulation-based setting (see however [34, 35] for a more recent version of A-API that is convergent). Hence, we focus on CAP-I (which is called ordinary or modified policy iteration in [33]) that appears to be slow, but steady and convergent.

CAP-I has three stages in each iteration. In the first stage, it evaluates the value function for each state; this is performed by simulating the policy being evaluated. In the second stage, the algorithm evaluates the Q-factors for the policy being evaluated using the value function generated in the first stage. In the third stage, policy improvement is performed. Each of the first two stages requires long simulations, and as such, this algorithm will be slower than Q-P-Learning. Our description is taken from [120].

Steps in CAP-I. We will now present the step-by-step details.

Step 1. Initialize $k_{\max}$, $n_{\max}$, and $m_{\max}$ to large numbers. Let the number of algorithm iterations be denoted by k. Set $k = 1$. Select any policy arbitrarily and call it $\hat{\mu}_k$.

Step 2 (Policy Evaluation: Estimating Value Function.) Start fresh simulation. Initialize $J(l) = 0$ for all $l \in \mathcal{S}$. Let the current system state be i. Set n, the number of iterations within the policy evaluation episode, to 1.

Step 2a. Simulate action $a = \mu_k(i)$ in state i.

Step 2b. Let the next state encountered in the simulator be j. Let $r(i, a, j)$ be the immediate reward earned in the transition from state i to state j. Update α. Then update $J(i)$ using:

$$J(i) \leftarrow (1 - \alpha)J(i) + \alpha \left[r(i, a, j) + \lambda J(j)\right] \text{ where } a = \mu_k(i).$$

Step 2c. Increment n by 1. If $n < n_{\max}$, set $i \leftarrow j$ and return to Step 2a. Otherwise go to Step 3.

Step 3 (Q-factor evaluation). Start fresh simulation. Set $Q(l, u) = 0$ for $l \in \mathcal{S}, u \in \mathcal{A}(l)$. Let the current system state be i. Set m, the number of iterations within an episode for Q-factor evaluations, to 1.

Step 3a. Simulate action $a \in \mathcal{A}(i)$ with probability $1/|\mathcal{A}(i)|$.

Step 3b. Let the next state encountered in the simulator be j. Let $r(i, a, j)$ be the immediate reward earned in the transition from state i to state j. Update β. Then update $Q(i, a)$ using:

$$Q(i, a) \leftarrow (1 - \beta)Q(i, a) + \beta \left[r(i, a, j) + \lambda J(j)\right].$$

Step 3c. Increment m by 1. If $m < m_{\max}$, set $i \leftarrow j$ and return to Step 3a. Otherwise go to Step 4.

Step 4 (Policy Improvement). Increment k by 1. Set for all $i \in \mathcal{S}$, $\mu_k(i) \in \arg\max_{a \in \mathcal{A}(i)} Q(i, a)$. If k equals $k_{\max}$, declare $\hat{\mu}_k$ to be an optimal policy and stop. Otherwise, go to Step 2.

The reader should note that the value of the $J(.)$ function used in Step 3 is that obtained at the end of Step 2. Also, note that the need for Q-factor evaluation (Step 3) arises from the fact that since the transition probabilities are unknown, it is *not* possible to perform policy improvement (see policy improvement step in dynamic programming from Chap. 6) on the basis of the value function ($J(.)$) alone. This is why any model-free API algorithm based on the value function must have the follow-up step (Step 3) of evaluating the Q-factors, since otherwise, it is not possible to perform policy improvement. In the literature, one finds that in the discussion on API, this step is often skipped, because it is assumed. But, as stated above, this step requires another long simulation, making this algorithm slower than many other algorithms based on the Bellman policy equation.

Some additional remarks about the algorithm's specifics are necessary. Both α and β can use the same updating rules. Further, both $n_{\max}$ and $m_{\max}$ should be set to large numbers, but since in the Q-factor evaluation stage, we do not use a recursive equation (in that we use a fixed value of $J(.)$ on the right hand side), it is quite possible for $m_{\max}$ to be significantly smaller than $n_{\max}$. (Clearly, $k_{\max}$ must always be large enough.)

AAP-I. If we set $n_{\max}$ to 1 or some small integer, we generate the class of algorithms loosely called ambitious API (AAP-I). It is not hard to see that such an algorithm will be faster than CAP-I, but as stated above, its convergence is questionable. Note that even if $n_{\max}$ is set to a small value, for any reasonable algorithm, $m_{\max}$ must be sufficiently high to perform a credible policy improvement.

4.2. Average Reward

This section will discuss DP-based RL algorithms for average reward MDPs. Like in the discounted case, we will divide this section into two parts: one devoted to value iteration and the other to policy iteration.

We first present the Q-factor version of the Bellman equation for average reward. For all (i, a) pairs:

$$Q(i,a) = \sum_{j=1}^{|\mathcal{S}|} p(i,a,j)\left[r(i,a,j) - \rho^* + \max_{b\in\mathcal{A}(j)} Q(j,b)\right], \qquad (7.25)$$

where ρ^* denotes the optimal average reward. The algorithms in this section will strive to solve the equation above. Now, using the notions of Robbins-Monro, like in the case of discounted reward, we can derive a Q-Learning algorithm of the following form:

$$Q(i,a) \leftarrow (1-\alpha)Q(i,a) + \alpha\left[r(i,a,j) - \rho^* + \max_{b\in\mathcal{A}(j)} Q(j,b)\right]$$

for all (i, a) pairs. The only difficulty with this algorithm is that ρ^* is not known in advance! Like in DP, we will use the notion of relative value iteration to circumvent this difficulty.

4.2.1 Relative Value Iteration

The notions of relative value iteration in classical DP can be extended to RL. The main idea is to select a state-action pair arbitrarily at the start of the algorithm. This pair will be denoted by (i^*, a^*) and called the **distinguished** state-action pair. Then each Q-factor will be updated via the following rule:

$$Q(i,a) \leftarrow (1-\alpha)Q(i,a) + \alpha \left[r(i,a,j) + \max_{b \in \mathcal{A}(j)} Q(j,b) - Q(i^*, a^*) \right].$$

The above algorithm will be called Relative Q-Learning in this book. It has been analyzed in [2], and it is shown there that $Q(i^*, a^*)$ will converge to ρ^*; as a result, the algorithm will converge to the optimal solution of the Bellman equation for average reward (see Eq. (7.25)). We now present steps in this algorithm.

Steps in Relative Q-Learning. The steps are identical to those of Q-Learning for discounted reward (see Sect. 4.1.1) with the following critical differences. Step 1 needs to perform the following in addition:

Step 1: Select any one state-action pair to be the distinguished pair (i^*, a^*). Step 4 is as follows:

Step 4: Update $Q(i, a)$ using the following:

$$Q(i,a) \leftarrow (1-\alpha)Q(i,a) + \alpha \left[r(i,a,j) + \max_{b \in \mathcal{A}(j)} Q(j,b) - Q(i^*, a^*) \right].$$

It is important to note the following. Relative value iteration (which is a DP algorithm) can diverge asynchronously and produce a sub-optimal solution, i.e., under asynchronous updating within a DP setting, e.g., in a Gauss-Seidel version. However, interestingly, Relative Q-Learning, which is also asynchronous, generates optimal solutions under appropriate conditions [2].

The value of the average reward produced by the algorithm's solution can also be easily measured within a simulator by re-running the simulator using the policy after the learning is complete. This is oftentimes called the *frozen* phase in RL. The following routine shows how to determine the average reward of a given policy. The simulation must be run for a sufficiently long time to obtain a good estimate, or else one must perform numerous replications using the same policy but different seeds (see Chap. 2).

Average Reward Calculation.

Step 1. Set TR, the total reward (cumulative) reward, to 0 and k, the number of transitions, to 0. Set $k_{\max}$ to a large number.

Step 2. Let the current state be i. Select action a where $a \in \arg\max_{u \in \mathcal{A}(i)} Q(i,u)$ i.e., choose the action associated with the maximum Q-factor for that state.

Step 3. Simulate action a. Let the next state be j. Update as follows:

$$TR \leftarrow TR + r(i,a,j); k \leftarrow k+1.$$

Step 4. If $k < k_{\max}$, set $i \leftarrow j$, and go to Step 2. Otherwise, go to Step 5.

Step 5. Calculate the average reward of the policy via $\rho = TR/k$, and stop.

Relative Q-Learning and Example A. Results from Example A (see Sect. 3.3.2 of Chap. 6) with Relative Q-Learning are as follows. We used $(i^*, a^*) = (1,1)$. The Q-factors obtained are:

$$Q(1,1) = 10.07, Q(1,2) = 18.25, Q(2,1) = 17.24, \text{ and } Q(2,2) = 14.99.$$

As a result, the policy learned is $(2,1)$. When this policy is run in a simulator, the average reward obtained is 10.56. Note that $Q(1,1) = 10.07 \simeq 10.56$.

4.3. R-SMART and Other Algorithms

A number of other convergent algorithms exist for solving the average reward MDP. When the transitions occur between adjacent states or the transition probability matrix is sparse, the Relative Q-Learning algorithm (discussed above) can produce disappointing results. There are at least three alternatives: the R-SMART algorithm [119], discussed below in Sect. 5.2.1 (see Remarks in that section) *and known to have robust behavior*, an SSP-based Q-Learning algorithm, (reader should study [2]), and Q-P-Learning, based on policy iteration.

Policy iteration An average reward RL algorithm based on policy iteration can be developed in a manner analogous to the derivation of Q-P-Learning for the discounted case. In the average reward problem, instead of using the Q-Learning algorithm in the policy evaluation phase, we will need to use the Relative Q-Learning algorithm. The algorithm's steps differ from those for the discounted reward case in the following ways.

1. In Step 1, also select any one state-action pair to be the distinguished pair (i^*, a^*).
2. In Step 2b, update $Q(i, a)$ using: $Q(i, a) \leftarrow (1 - \alpha)Q(i, a)+$ $\alpha \left[r(i, a, j) - Q(i^*, a^*) + Q\left(j, \arg\max_{b \in \mathcal{A}(j)} P(j, b)\right)\right].$

5. SMDPs

In this section, we will first discuss the discounted reward case for solving the generalized semi-Markov decision problems (SMDPs), using RL, and then discuss the average reward case. We remind the reader that the SMDP is a more powerful model than the MDP, because it explicitly models the time spent in a transition. In the MDP, the time spent is the same for every transition. The RL algorithms for SMDPs use extensions of Q-Learning and Q-P-Learning for MDPs. Please refer to value and policy iteration for SMDPs and also review definitions from Chap. 6.

5.1. Discounted Reward

The discounted reward SMDP can be solved using either a value-iteration-based approach or else a policy-iteration-based approach.

We would like to remind the reader that in the generalized SMDP (see Chap. 6), we will assume that some of the immediate reward is earned immediately after action a is selected, i.e., at the start of the transition from i to j. This is called the lump sum reward and is denoted by $r_L(i, a, j)$, where the subscript L denotes lump sum. In addition, one can earn reward continuously during the transition at the rate $r_C(i, a, j)$, where the subscript C denotes continuous.

5.1.1 Steps in Q-Learning: Generalized SMDPs

The steps are similar to those in Q-Learning for MDPs with the following exception. In Step 4, update $Q(i, a)$ via: $Q(i, a) \leftarrow (1 - \alpha)Q(i, a)+$

$$\alpha\left[r_L(i,a,j)+\frac{1-e^{-\gamma t(i,a,j)}}{\gamma}r_C(i,a,j)+e^{-\gamma t(i,a,j)}\max_{b\in\mathcal{A}(j)}Q(j,b)\right], \tag{7.26}$$

where $t(i,a,j)$ is the *possibly random* amount of time it takes to transition from i to j under a and γ is the rate of return or rate of interest. A version of this algorithm without the lump sum reward appears in [52], while the version with the lump sum reward and a convergence proof appears in [119].

5.1.2 Steps in *Q*-*P*-Learning: Generalized SMDPs

The steps are similar to those of Q-P-Learning for discounted reward MDPs with the following difference in Step 2b.

In Step 2b, update $Q(i,a)$ using: $Q(i,a) \leftarrow (1-\alpha)Q(i,a)+$

$$\alpha\left[r_L(i,a,j)+\frac{1-e^{-\gamma t(i,a,j)}}{\gamma}r_C(i,a,j)+e^{-\gamma t(i,a,j)}Q\left(j,\arg\max_{b\in\mathcal{A}(j)}P(j,b)\right)\right].$$

This algorithm is described in [118].

5.2. Average Reward

Solving average reward SMDPs via RL is perhaps more difficult than all the other problems discussed above. The reasons will be explained below. However, many real-world problems, including those arising in queueing networks, tend to be SMDPs rather than MDPs. As such, it is important that we develop the theory for it.

We remind the reader that in the average reward MDPs, one can bypass the need to estimate ρ^*, the optimal average reward, via relative value iteration (and Relative Q-Learning in RL). However, in SMDPs, this does not work in value iteration. For an illustration of the difficulties posed by ρ^* in SMDPs, see the Example in Sect. 7.2.4 of Chap. 6. One way out of this difficulty is via discretization of the SMDP to an MDP, discussed in the previous chapter. But, the discretization requires the transition probabilities. Since we do not have the access to transition probabilities in RL, unlike in DP, we must find ways to circumvent this difficulty without discretization. One approach that we will focus on here is to use the two-time-scale framework of Borkar [45] to develop a value iteration algorithm that will estimate Q-factors on one time scale using a learning rate α and the average reward on the other using a learning rate β, where α and β are different and must satisfy some conditions.

We begin by presenting a Q-factor version of the Bellman equation for average reward SMDPs, which can be derived in a manner similar to that for the discounted MDP. For all (i, a) pairs:

$$Q(i,a) = \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) \left[r(i,a,j) - \rho^* \bar{t}(i,a,j) + \max_{b \in \mathcal{A}(j)} Q(j,b) \right]. \quad (7.27)$$

This suggests a value iteration algorithm of the form:

$$Q(i,a) \leftarrow \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) \left[r(i,a,j) - \rho^* \bar{t}(i,a,j) + \max_{b \in \mathcal{A}(j)} Q(j,b) \right] \quad \forall (i,a).$$

Unfortunately, this cannot be used directly to derive an RL algorithm, since the value of ρ^* is unknown. One approach is to use the following update instead

$$Q(i,a) \leftarrow (1-\alpha)Q(i,a) + \alpha \left[r(i,a,j) - \rho t(i,a,j) + \max_{b \in \mathcal{A}(j)} Q(j,b) \right], \ \forall (i,a) \quad (7.28)$$

where ρ here denotes a guessed estimate of ρ^* updated gradually, and one hopes that it will converge to ρ^*. The mechanism to update ρ that we will use is:

$$\rho \leftarrow (1-\beta)\rho + \beta \left[\frac{TR}{TT} \right], \quad (7.29)$$

where the TR and TT denote the total accumulated reward and the total accumulated time, respectively, in the system during the state transitions triggered by *greedy* actions chosen by the algorithm. These updating equations are the basis for the R-SMART family of algorithms [110, 119] that we will focus on. As we will see below, the update in Eq. (7.28) can pose some numerical difficulties, and must be further modified.

It is important to note that in the update defined in Eq. (7.28), $t(i,a,j)$ denotes the *possibly random* transition time in going from i to j under a's influence. The notation $\bar{t}(i,a,j)$, which is used in the updates that involve the transition probabilities, denotes the *mean* transition time in going from i to j under a's influence. The two terms are related as follows:

$$\bar{t}(i,a,j) = \mathsf{E}[t(i,a,j)] = \int_0^\infty \tau f_{i,a,j}(\tau) d\tau,$$

where $f_{i,a,j}(.)$ is the *pdf* of the transition time from i to j under a's influence.

5.2.1 R-SMART for SMDPs

In this subsection, we discuss a class of algorithms collectively referred to as R-SMART (Relaxed-Semi-Markov Average Reward Technique). As the name suggests, this class of algorithms is designed to solve average reward SMDPs; the prefix "Relaxed" indicates that the algorithm uses the so-called "two time scale" version. We will discuss the framework of two time scales later. We discuss two versions of R-SMART algorithms, the first version being called the CF-version, while the second the SSP-version.

CF-Version of R-SMART. This version of R-SMART needs an assumption that we discuss now. Consider the Q-factor version of the following variant of the Bellman optimality equation for average reward SMDP.

$$Q(i,a) = \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) \left[r(i,a,j) - \rho \bar{t}(i,a,j) + \eta \max_{b \in \mathcal{A}(j)} Q(j,b) \right], \tag{7.30}$$

where η is a scalar that satisfies $0 < \eta < 1$. For any given value of ρ, it can be shown that the above equation has a unique solution; this is because it behaves like the Bellman equation for the discounted reward MDP with its immediate reward altered to $r(i,a,j) - \rho\bar{t}(i,a,j)$.

We will call η the contraction factor (CF), because if ρ is fixed, it makes the transformation defined in Eq. (7.30) a so-called contractive transformation, which has some nice convergence properties. We will discuss contractive transformations in detail in Chap. 9. The algorithm that we present below will seek to solve the Bellman equation presented in Eq. (7.30).

Do note that the true version of the Bellman optimality equation for average reward SMDPs will have $\eta = 1$ and $\rho = \rho^*$. The assumption needed by the CF-version of R-SMART is as follows:

Assumption 7.1 *There exists a value $\bar{\eta}$ in the open interval* $(0,1)$ *such that for all $\eta \in (\bar{\eta}, 1)$, the unique solution of Eq. (7.30) obtained after setting $\rho = \rho^*$ produces a policy $\hat{d}$ whose average reward equals ρ^*.*

Note that Assumption 7.1 implies that the policy $\hat{d}$ will produce for each $i \in \mathcal{S}$, an action $d(i) \in \arg\max_{a \in \mathcal{A}(i)} Q(i,a)$, where $Q(i,a)$ denotes the unique solution of Eq. (7.30) when solved with $\rho = \rho^*$. Essentially, what Assumption 7.1 assures us is that for a sufficiently large value of η (i.e., η sufficiently close to 1), a solution of Eq. (7.30) with $\rho = \rho^*$ will yield the optimal policy.

We now derive an RL algorithm based on this idea and call it the contracting factor (CF) version of R-SMART. The algorithm will use an update based on Eq. (7.30) while ρ will be updated via Eq. (7.29) under the so-called two time scale condition that we now explain.

Using two time scales means having one class of iterates, e.g., Q-factors, that is updated using one step size, α, and another class of iterates, e.g., the scalar ρ, that is updated using a different step size, β, such that α and β share the following relationship:

$$\lim_{k\to\infty} \frac{\beta^k}{\alpha^k} = 0; \tag{7.31}$$

note that the superscript, k, used above with α and β, indicates that the step sizes are functions of k. The superscript has been suppressed elsewhere to increase clarity. Step-size rules such as $\alpha^k = log(k)/k$ and $\beta^k = A/(B+k)$ (with suitable values of A and B) satisfy Eq. (7.31); other rules that satisfy the relationship can also be identified in practice. The condition in Eq. (7.31) is a defining feature of the two-time-scale framework. Whenever two time scales are used, the two step sizes must satisfy this condition.

Steps in CF-Version of R-SMART.

Step 1. Set for all $l \in \mathcal{S}, u \in \mathcal{A}(l)$, $Q(l,u) = 0$. Set k, the number of state transitions, to 0. We will run the algorithm for $k_{\max}$ iterations, where $k_{\max}$ is chosen to be a sufficiently large number. Start system simulation at any arbitrary state. Set TR, TT, and ρ to 0.

Step 2. Let the current state be i. Select action a via an exploratory strategy, where the rate of exploration is decayed with iterations. If $a \in \arg\max_{u\in\mathcal{A}(i)} Q(i,u)$, set $\phi = 0$. Otherwise set $\phi = 1$.

Step 3. Simulate action a. Let the next state be j. Let $r(i,a,j)$ be the immediate reward earned in the transition to state j from state i under the influence of action a. Let $t(i,a,j)$ denote the time spent in the same transition. Increment k by 1 and update α.

Step 4a. Update $Q(i,a)$ via:

$$Q(i,a) \leftarrow (1-\alpha)Q(i,a)+\alpha\left[r(i,a,j) - \rho t(i,a,j) + \eta \max_{b\in\mathcal{A}(j)} Q(j,b)\right]; \tag{7.32}$$

Step 4b. If ϕ equals 0 (i.e., a is a greedy action), update the following.

$$TR \leftarrow TR + r(i,a,j); TT \leftarrow TT + t(i,a,j).$$

Then, update ρ via Eq. (7.29) in which β is another step-size that must share the relationship defined in Eq. (7.31) with α.

Step 5. If $k < k_{\max}$, set $i \leftarrow j$, and then go to Step 2. Otherwise, go to Step 6.

Step 6. For each $l \in \mathcal{S}$, select $d(l) \in \arg\max_{b \in \mathcal{A}(l)} Q(l, b)$. The policy (solution) generated by the algorithm is $\hat{d}$. Stop.

It is not difficult to see that the update defined in Eq. (7.32) seeks to solve the version of the Bellman equation given in Eq. (7.30).

We now present some important remarks.

Remark 1: The above algorithm can be used to solve an MDP by setting $t(\cdot, \cdot, \cdot) = 1$, i.e., the time of each transition is set to 1.

Remark 2: Unfortunately, it is not possible to guess $\bar{\eta}$, and hence we must use a value for η that is sufficiently close to 1. If our value for η turns out to be below $\bar{\eta}$, Assumption 7.1 is violated, and we may not obtain the optimal solution. In practice, hence, the user must guesstimate η, and if the solutions produced are worse than those produced by the competition (e.g., a problem-specific heuristic), one must increase η's value. It has been found empirically that even with the value of $\eta = 1$, this algorithm frequently converges to the optimal solution.

SSP-Version of R-SMART. We now present another version of R-SMART. For this version, we first need to present some background related to solving an average reward MDP using a technique different than Relative Q-learning. We will follow this discussion by how this technique for MDPs can be then extended for solving SMDPs.

Associated with an average reward MDP with all Markov chains regular, one can construct an SSP (stochastic shortest-path problem) that has the same solution as the original MDP. The motivation for constructing it is, of course, that it may be more convenient to solve the SSP instead of the original MDP [30, 32]. We will not attempt to explain mathematically how an SSP can be exploited to solve the average reward MDP, but refer the interested reader to Bertsekas [30], where the idea was introduced. We present an intuitive explanation of this idea.

The SSP is a problem in which there is an absorbing state(s), and the goal is to maximize the *total* (un-discounted) reward earned from the starting state until the system is absorbed in the absorbing state. Via an ingenious argument, Bertsekas [30] showed that under some conditions, one can construct an *imaginary* SSP from any average reward MDP easily. This SSP has some nice properties and is easy to solve. What is interesting is that *the solution to this SSP is an optimal solution of the original MDP as well!*

The construction of the associated SSP requires that we consider *any* state in the MDP (provided that every Markov chain is regular, an assumption we make throughout) to be the absorbing state; we call this state the distinguished state and denote it by i^*. In the associated SSP, if the distinguished state is the next state (j) in a transition and the Q-factor of the previous state (i) is being updated after the transition occurs, the value of zero will replace the distinguished state's Q-factor. (Remember that when we update a Q-factor, we need Q-factors from the *next* state as well.) However, when the turn comes to update a Q-factor of the distinguished state, it will be updated just like any other Q-factor. Also, very importantly, the SSP requires that we replace the immediate reward, $r(i, a, j)$, by $r(i, a, j) - \rho^*$. Since ρ^* is unknown at the start, we will update ρ via Eq. (7.29), under the two-time-scale conditions (discussed previously).

Now, clearly, here, we are interested here in solving the SMDP rather than the MDP. It was shown in [119] that similar to the MDP, by using a distinguished (absorbing) state i^*, one can construct an SSP for an SMDP as well—such that the solution of the SSP is identical to that of the original SMDP. For the SMDP, solving the associated SSP requires that we replace the immediate reward, $r(i, a, j)$, by $r(i, a, j) - \rho^*\bar{t}(i, a, j)$, where note that we have the time element $(t(i, a, j))$ in the modified immediate reward. Then, provided the value of ρ^* is known in advance, the following value iteration algorithm can be used to solve the SSP and hence the SMDP:

$$Q(i,a) \leftarrow \sum_{j\in\mathcal{S}} p(i,a,j)\left[r(i,a,j) - \rho^*\bar{t}(i,a,j) + I(j \neq i^*) \max_{b\in\mathcal{A}(j)} Q(j,b)\right], \tag{7.33}$$

where $I(.)$ is an indicator function, and it equals 1 when $j \neq i^*$ and equals 0 when $j = i^*$. Of course, one must update ρ in a manner such that it reaches ρ^* in the limit. We will update ρ on a second time scale—in a manner identical to that used in the CF-version.

Equation (7.33) thus serves as the basis for deriving a Q-Learning algorithm (see Eq. (7.34) below). Convergence for the resulting algorithm under certain conditions is shown in [119]. We now present steps in the algorithm.

Steps in SSP-Version of R-SMART.

Step 1. Set for all $l \in \mathcal{S}, u \in \mathcal{A}(l)$, $Q(l, u) = 0$. Set k, the number of state transitions, to 0. We will run the algorithm for $k_{\max}$ iterations, where $k_{\max}$ is chosen to be a sufficiently large number. Start system simulation at any arbitrary state. Set TR, TT and ρ to 0. Select any state in the system to be the distinguished state i^*.

Step 2. Let the current state be i. Select action a via an exploratory strategy, where the rate of exploration is decayed with iterations. If $a \in \arg\max_{u \in \mathcal{A}(i)} Q(i, u)$, set $\phi = 0$. Otherwise set $\phi = 1$.

Step 3. Simulate action a. Let the next state be j. Let $r(i, a, j)$ be the immediate reward earned in the transition to state j from state i under the influence of action a. Let $t(i, a, j)$ denote the time spent in the same transition. Increment k by 1 and update α.

Step 4a. Update $Q(i, a)$ via:

$$Q(i,a) \leftarrow (1-\alpha)Q(i,a) + \alpha \left[r(i,a,j) - \rho t(i,a,j) + I(j \neq i^*) \max_{b \in \mathcal{A}(j)} Q(j,b) \right]. \tag{7.34}$$

Step 4b. If ϕ equals 0 (i.e., a is a greedy action), update the following.

$$TR \leftarrow TR + r(i,a,j); TT \leftarrow TT + t(i,a,j).$$

Then, update ρ via Eq. (7.29) in which β is another step-size that must share the relationship defined in Eq. (7.31) with α.

Step 5. If $k < k_{\max}$, set $i \leftarrow j$, and then go to Step 2. Otherwise, go to Step 6.

Step 6. For each $l \in \mathcal{S}$, select $d(l) \in \arg\max_{b \in \mathcal{A}(l)} Q(l, b)$. The policy (solution) generated by the algorithm is $\hat{d}$. Stop.

Remark: The above algorithm can be used to solve an MDP by setting $t(\cdot,\cdot,\cdot) = 1$, i.e., the time of each transition is set to 1. We further note that the CF-version produces graceful behavior and can often be more robust than the SSP version in how the Q-factors behave.

5.2.2 *Q*-*P*-Learning for SMDPs

We now discuss an RL approach based on policy iteration for solving SMDPs. The Q-P-Learning algorithm has a policy evaluation phase, which is performed via value iteration. For discounted reward MDPs and SMDPs, regular value iteration (derived from the Bellman policy equation) keeps the iterates bounded and also converges. For average reward MDPs, one uses relative value iteration (derived from the Bellman policy equation). For average reward SMDPs, we cannot use relative value iteration without discretization. But, after a suitable modification, one can use the Bellman equation for value iteration for a given policy. The Bellman equation relevant to this case is the Bellman policy (or Poisson) equation, which can be expressed in terms of Q-factors as follows.

$$Q(i,a) = \sum_{j=1}^{|\mathcal{S}|} p(i,a,j)[r(i,a,j) - \rho_{\hat{\mu}}\bar{t}(i,a,j) + Q(j,\mu(j))].$$

The above is the Bellman policy equation for the policy $\hat{\mu}$ and in general does not have a unique solution, which may pose problems in RL. We now discuss two approaches to bypass this difficulty.

Like in value-iteration-based RL, one approach is to solve the associated SSP, and use a variant of the above equation that applies for the SSP. Hence the first question is, what does the equation look like for the SSP? Using the notion of a distinguished state i^*, we have the following Bellman policy equation:

$$Q(i,a) = \sum_{j=1}^{|\mathcal{S}|} p(i,a,j)[r(i,a,j) - \rho_{\hat{\mu}}\bar{t}(i,a,j) + I(j \neq i^*)Q(j,\mu(j))]. \tag{7.35}$$

Now, the second question is how does one obtain the value of $\rho_{\hat{\mu}}$? This can be resolved as follows: *Before* updating the Q-factors using this equation, one estimates the average reward of the policy $\hat{\mu}$ in a simulator. Thus, in Step 2, we estimate the average reward of the policy, whose Q-factors are evaluated later in Step 3 using an update based on Eq. (7.35).

The second approach is to use the CF-version. We will discuss that later. We first present steps in the SSP-version of Q-P-Learning for average reward SMDPs.

Steps in the SSP-Version.

Step 1: Initialize the vector $P(l,u)$ for all states $l \in S$ and all $u \in \mathcal{A}(l)$ to arbitrary value. Set $k = 1$ (k denotes the number of episodes) and initialize $k_{\max}$ and $n_{\max}$ to large numbers. Set any state in the system to i^*, the distinguished state.

Step 2: Set $Q(l,u) = 0$ for all $l \in S$ and all $u \in \mathcal{A}(l)$. Simulate the system for a sufficiently long time using in state m, the action given by $\arg\max_{b \in A(m)} P(m,b)$. At the end of the simulation, divide the total of immediate rewards by the total of immediate times to obtain an estimate of the average reward ρ. Use averaging with several replications to obtain a good estimate of ρ. Set n, the number of iterations within an episode, to 1.

Step 3 (Policy evaluation) Start fresh simulation. Let the current system state be $i \in \mathcal{S}$.

Step 3a: Simulate action $a \in \mathcal{A}(i)$ with probability $1/|\mathcal{A}(i)|$.

Step 3b: Let the next decision-making state encountered in the simulator be j. Also, let $t(i,a,j)$ be the transition time (from state i to state j) and let $r(i,a,j)$ be the immediate reward.

Step 3c: Calculate α using the step-size rules discussed above. Then update $Q(i,a)$ using: $Q(i,a) \leftarrow (1-\alpha)Q(i,a)+$

$$\alpha \left[r(i,a,j) - \rho t(i,a,j) + I(j \neq i^*) Q\left(j, \arg\max_{b \in \mathcal{A}(j)} P(j,b) \right) \right].$$

Step 3d: Increment n by 1. If $n < n_{max}$ set current state i to new state j and then go to Step 3a; else go to Step 4.

Step 4: (*Q to P conversion – policy improvement*) Set $P(l,u) \leftarrow Q(l,u)$ for all l and $u \in \mathcal{A}(l)$. Set $k \leftarrow k+1$. If k equals $k_{\max}$ go to Step 5; else go back to step 2.

Step 5. For each $l \in \mathcal{S}$, select $d(l) \in \arg\max_{b \in \mathcal{A}(l)} Q(l,b)$. The policy (solution) generated by the algorithm is $\hat{d}$. Stop.

The above description is based on [118, 109].

Steps in the CF-Version. The CF-version is based on using the contraction factor, η, where $0 < \eta < 1$. For this, we will need a counterpart of Assumption 7.1 for a given policy. We state that assumption next. First we present the CF-version of the Bellman equation for a given policy (Poisson equation), $\hat{\mu}$, in an average reward SMDP.

$$Q(i,a) = \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) \left[r(i,a,j) - \rho \bar{t}(i,a,j) + \eta Q(j, \mu(j)) \right], \quad (7.36)$$

where η is a scalar that satisfies $0 < \eta < 1$. For any given value of ρ, it can be shown that the above equation has a unique solution; this is because it behaves like the discounted reward Bellman equation for a given policy with its immediate reward altered to $r(i,a,j) - \rho t(i,a,j)$.

Assumption 7.2 *There exists a value $\bar{\eta}$ in the open interval $(0,1)$ such that for all $\eta \in (\bar{\eta}, 1)$, the unique solution of Eq. (7.36) obtained after setting $\rho = \rho_{\hat{\mu}}$ produces a policy $\hat{d}$ whose average reward equals $\rho_{\hat{\mu}}$.*

Essentially, what Assumption 7.2 assures us is that for a sufficiently large value of η (i.e., η sufficiently close to 1), a solution of Eq. (7.36) with $\rho = \rho_{\hat{\mu}}$ will yield the policy $\hat{\mu}$.

The steps in the algorithm will be identical to those above except that there would be no need to select a distinguished state in Step 1 and that the update in Step 3c would be: $Q(i,a) \leftarrow (1-\alpha)Q(i,a)+$

$$\alpha \left[r(i,a,j) - \rho t(i,a,j) + \eta Q\left(j, \arg\max_{b \in \mathcal{A}(j)} P(j,b) \right) \right].$$

In practice, setting $\eta = 1$ also appears to work, but then the convergence arguments require that boundedness of the Q-factors is assumed apriori. It is to be noted that both of these versions of Q-P-Learning can be used to solve MDPs by setting $t(\cdot,\cdot,\cdot) = 1$, i.e., the time of each transition is set to 1.

6. Model-Building Algorithms

Scattered in the literature on RL, one finds some papers that discuss a class of algorithms called model-**based** algorithms. These algorithms actually build the transition probability model in some form and attempt to use the Bellman equation in its original form, i.e., with its transition probabilities intact. At the beginning of the chapter, we have discussed a mechanism to first build the transition probability model within the simulator and then use DP. Model-based algorithms are similar in spirit, but they do not wait for the model to be built. Rather, they start updating the value function, or the Q-factors, while the model is being simultaneously built. The question that arises is this: What is the advantage underlying taking the additional step of building the model when we already have the model-free Bellman equation in terms of Q-factors (which we have used all along)? The answer is that oftentimes model-free algorithms exhibit unstable behavior, and their performance also depends on the choice of the step-size. Model-based algorithms hold the promise of being more stable than their model-free counterparts [324].

We will call these algorithms model-**building** algorithms to reflect the fact they *build*, directly or indirectly, the transition probability model, while the algorithm works, i.e., during its run time. Strictly speaking, only DP is model-based, because it requires the knowledge of the model *before* the algorithm starts working. We emphasize that model-building algorithms do not require the *transition-probability* model (loosely referred to as a "model" here); like the model-free algorithms of the previous sections, all they need is a *simulation model* of the system.

We have already discussed how to construct the transition-probability model within a simulator in Sect. 2.2. The same principle will be applied here. As stated above, model-building algorithms do not wait for the model to be built but start updates within the simulator even as the model is being built. The number of times each state-action pair is tried in the simulator can be kept track of; we will call this quantity the **visit factor**. $V(i,a)$ will denote the visit factor for the state-action pair (i,a).

6.1. RTDP

We first present model-building algorithms that compute the *value function* rather than the Q-factors. The central idea here is to estimate the immediate reward function, $\bar{r}(i,a)$, and the transition probability function, $p(i,a,j)$. While this estimation is being done, we use a step-size-based asynchronous DP algorithm. The estimates of the immediate reward and transition probabilities will be denoted by $\tilde{r}(i,a)$ and $\tilde{p}(i,a,j)$, respectively.

The following algorithm, called RTDP (Real Time Dynamic Programming), is based on the work in [17]. It is designed for discounted reward MDPs.

Step 1. Set for all $l, m \in \mathcal{S}$ and $u \in \mathcal{A}(l)$,

$$h(l) \leftarrow 0, V(l,u) \leftarrow 0, W(l,u,m) \leftarrow 0, \text{ and } \tilde{r}(l,u) \leftarrow 0.$$

Set k, the number of state transitions, to 0. We will run the algorithm for $k_{\max}$ iterations, where $k_{\max}$ is chosen to be a sufficiently large number. Start system simulation at any arbitrary state.

Step 2. Let the state be i. Select an action in state i using some action selection strategy (see comment below for action selection). Let the selected action be a.

Step 3. Simulate action a. Let the next state be j.

Step 4. Increment both $V(i,a)$ and $W(i,a,j)$ by 1.

Step 5. Update $\tilde{r}(i,a)$ via: $\tilde{r}(i,a) \leftarrow \tilde{r}(i,a) + \left[r(i,a,j) - \tilde{r}(i,a)\right]/V(i,a)$.

Step 6. Calculate for all $l \in \mathcal{S}$, $\tilde{p}(i,a,l) \leftarrow W(i,a,l)/V(i,a)$.

Step 7. Update $h(i)$ using the following equation:

$$h(i) \leftarrow \max_{u \in \mathcal{A}(i)} \left[\tilde{r}(i,u) + \lambda \sum_{l \in \mathcal{S}} \tilde{p}(i,u,l)h(l)\right].$$

Step 8. Increment k by 1. If $k < k_{\max}$, set $i \leftarrow j$, and go to Step 2. Otherwise go to Step 9.

Step 9. For each $m \in \mathcal{S}$, select

$$d(m) \in \arg\max_{u \in \mathcal{A}(m)} \left[\tilde{r}(m,u) + \lambda \sum_{l \in \mathcal{S}} \tilde{p}(m,u,l)h(l)\right].$$

The policy (solution) generated by the algorithm is $\hat{d}$. Stop.

6.2. Model-Building Q-Learning

We now present model-building algorithms of the Q-Learning type, which estimate Q-factors rather than the value function. We begin with the discounted reward case. It has two notable differences with RTDP and H-Learning. (1) It computes Q-factors instead of computing the value function vector ($\vec{h}$) and (2) it uses a step sizes.

Step 1. Set for all $l, m \in \mathcal{S}$ and $u \in \mathcal{A}(l)$,

$$Q(l,u) \leftarrow 0, V(l,u) \leftarrow 0 \text{ and } \tilde{r}(l,u) \leftarrow 0, \text{ and } W(l,u,m) \leftarrow 0.$$

Set the number of state transitions, k, to 0. We will run the algorithm for $k_{\max}$ iterations, where $k_{\max}$ is chosen to be a sufficiently large number. Start system simulation at any arbitrary state.

Step 2. Let the current state be i. Select action $u \in \mathcal{A}(i)$ with probability $1/|\mathcal{A}(i)|$. Let the selected action be a.

Step 3. Simulate action a. Let the next state be j. Increment k by 1. Increment both $V(i,a)$ and $W(i,a,j)$ by 1.

Step 4. Update $\tilde{r}(i,a)$ via: $\tilde{r}(i,a) \leftarrow \tilde{r}(i,a) + [r(i,a,j) - \tilde{r}(i,a)]/V(i,a)$.

Step 5. Calculate for all $l \in \mathcal{S}$, $\tilde{p}(i,a,l) \leftarrow W(i,a,l)/V(i,a)$.

Step 6. Update $Q(i,a)$ using the following equation:

$$Q(i,a) \leftarrow (1-\alpha)Q(i,a) + \alpha\left[\tilde{r}(i,a) + \lambda\sum_{l\in\mathcal{S}}\tilde{p}(i,a,l)\max_{b\in\mathcal{A}(l)} Q(l,b)\right].$$

Step 7. Increment k by 1. If $k < k_{\max}$, set $i \leftarrow j$ and go to Step 2. Otherwise go to Step 8.

Step 8. For each $m \in \mathcal{S}$, select $d(m) \in \arg\max_{b\in\mathcal{A}(m)} Q(m,b)$. The policy (solution) generated by the algorithm is $\hat{d}$. Stop.

The step size, α, can be decayed as in regular Q-Learning. A similar algorithm for the average reward case works as follows. The steps will be identical to those above with the exception of Step 6, where we will use the idea of Relative Q-Learning:

Step 6. Update $Q(i,a)$ using the following equation:

$$Q(i,a) \leftarrow (1-\alpha)Q(i,a) + \alpha\left[\tilde{r}(i,a) + \sum_{l\in\mathcal{S}}\tilde{p}(i,a,l)\max_{b\in\mathcal{A}(l)} Q(l,b) - Q(i^*,a^*)\right].$$

6.3. Indirect Model-Building

We now discuss some model-building algorithms from [115] that do not require the storage of the V terms of the W terms. These algorithms will seek to build the model indirectly, i.e., the transition probabilities will not be sought to be estimated. Rather, the algorithm will estimate the $\tilde{r}$ terms and also estimate the *expected* value of the next Q-factor.

Step 1. Set for all (l,u), where $l \in \mathcal{S}$ and $u \in \mathcal{A}(l)$, $Q(l,u) \leftarrow 0$, $\tilde{r}(l,u) \leftarrow 0$, and $Q_n(l,u) \leftarrow 0$. Note that $Q_n(l,u)$ will denote the estimate of the maximum Q-factor for the *next* state when action u is chosen in state l. Set k, the number of state changes, to 0. Set $k_{\max}$, which denotes the maximum number of iterations for which the algorithm is run, to a sufficiently large number; note that the algorithm runs iteratively between Steps 2 and 6. Select step sizes α and β using the two-time-scale structure discussed previously. Start system simulation at any arbitrary state.

Step 2. Let the current state be i. Select action a with a probability of $1/|\mathcal{A}(i)|$.

Step 3. Simulate action a. Let the next state be j. Increment k by 1.

Step 4. Update $Q(i, a)$ using the following equation:

$$Q(i, a) \leftarrow (1 - \alpha)Q(i, a) + \alpha \left[\tilde{r}(i, a) + \lambda Q_n(i, a)\right].$$

Step 5. Update $Q_n(i, a)$ and $\tilde{r}(i, a)$ as follows:

$$Q_n(i, a) \leftarrow (1-\beta)Q_n(i, a)+\beta \max_{b \in \mathcal{A}(j)} Q(j, b); \ \tilde{r}(i, a) \leftarrow (1-\beta)\tilde{r}(i, a)+\beta r(i, a, j).$$

Step 6. If $k < k_{max}$, set $i \leftarrow j$ and then go to Step 2. Otherwise, go to Step 7.

Step 7. For each $l \in \mathcal{S}$, select $d(l) \in \arg\max_{b \in \mathcal{A}(l)} Q(l, b)$. The policy (solution) generated by the algorithm is $\hat{d}$. Stop.

Note that the update in Step 5 uses samples; but Step 4 uses estimates of the *expected* immediate reward and the *expected* value of the Q-factor of the next state, making the algorithm model-based. Its critical feature is that it avoids estimating the transition probabilities and also computing the expectation with it. The steps for the average reward case are same with the following changes: Any one state-action pair, to be denoted by (i^*, a^*), is selected in Step 1, and the update in Step 4 is changed to the following:

$$Q(i, a) \leftarrow (1 - \alpha)Q(i, a) + \alpha \left[\tilde{r}(i, a) + Q_n(i, a) - Q_n(i^*, a^*)\right].$$

7. Finite Horizon Problems

Finite horizon problems can be solved using an extension of Q-Learning discussed for infinite horizon problems. The reader is advised to read the section on finite horizon in Chap. 6.

We will assume that the problem has a unique starting state. One would have to associate a Q-factor with the following triple: (i, a, s), where i denotes the state, a the action and s the stage. Thus, the Q-factor for choosing action a in the state-stage combination (i, s) will be denoted by $Q(i, s, a)$. Also, the immediate reward earned when action a is selected in (i, s) and the system transitions to $(j, s + 1)$ will be denoted by $r(i, s, a, j, s + 1)$.

We will assume that decision making is to be made in T stages with $T < \infty$; also, we define $\mathcal{T} = \{1, 2, \ldots, T\}$. When the system reaches the terminal stage, $(T + 1)$, we will reset the system, within the simulator, to the starting state, which we assume to be unique (or known with certainty). For the starting state, $s = 1$. The stage, s, will be incremented by 1 after very state transition. The problem solution methodology will be similar in every other manner to the infinite horizon problem. The main step in the total discounted reward (over a time horizon of T stages) algorithm would be:

$$Q(i,s,a) \leftarrow (1-\alpha)Q(i,s,a) + \alpha \left[r(i,s,a,j,s+1) + \lambda \max_{b \in \mathcal{A}(j,s+1)} Q(j,s+1,b) \right]. \tag{7.37}$$

Note that $Q(j,T+1,b) = 0$ for all $j \in \mathcal{S}$ and $b \in \mathcal{A}(j,T+1)$.

For total expected reward, one should set $\lambda = 1$. The algorithm can be analyzed as a special case of the SSP (see [116]; see also [94]). We present a step-by-step description below.

Step 1. Set k, the number of iterations, to 0. Initialize the Q-factors, $Q(i,s,a)$ for all $i \in \mathcal{S}$, all $s \in \mathcal{T}$ and all $a \in \mathcal{A}(i,s)$, to 0. Also set $Q(j,T+1,b) = 0$ for all $j \in \mathcal{S}$ and $b \in \mathcal{A}(j,T+1)$. The algorithm will be run for $k_{\max}$ iterations, where $k_{\max}$ is chosen to be a sufficiently large number. Start system simulation at the starting state, which is assumed to be known with certainty. Set $s = 1$.

Step 2. Let the current state be i and the current stage be s. Select action a with a probability of $1/|\mathcal{A}(i,s)|$. Simulate action a. Let the next state be j. The next stage will be $(s+1)$. Let $r(i,s,a,j,s+1)$ be the immediate reward earned in transiting to $(j,s+1)$ from (i,s) under a's influence.

Step 3. Update the $Q(i,s,a)$ via Eq. (7.37). Then, increment k by 1. Increment s by 1, and if the new value of s equals $(T+1)$, set $s = 1$. If $k < k_{\max}$, set $i \leftarrow j$ and return to Step 2. Otherwise, go to Step 4.

Step 4. For each $l \in \mathcal{S}$ and each $s \in \mathcal{T}$, select $d(l,s) \in \arg\max_{b \in \mathcal{A}(l,s)} Q(l,s,b)$. The policy (solution) generated by the algorithm is $\hat{d}$. Stop.

With this, we end our discussion on RL algorithms using look-up tables. In the next section, we will discuss some function approximation strategies that can be used to solve problems with a large number of state-action pairs.

8. Function Approximation

In a look-up table, each Q-factor is stored individually in a table in the computer's memory. As stated in the beginning of the chapter, for large state-action spaces which require millions of Q-factors to be stored, look-up tables are ruled out. This, we remind the reader, is called the curse of dimensionality.

Consider an MDP with a million state-action pairs. Using model-free algorithms, one can avoid the huge transition probability matrices associated to the MDP. However, one must still find some means of storing the one million Q-factors. We will now discuss some strategies that allow this.

8.1. State Aggregation

State **aggregation** means lumping several states together. This is usually the first strategy tried for dealing with the curse of dimensionality. With aggregation of states, one may obtain a relatively smaller number of state-action pairs for which one can store Q-factors individually, making look-up tables feasible.

When states are lumped together, we may lose the Markov property, the DP solution is not guaranteed to be optimal, and consequently the procedure becomes heuristic. However, empirical evidence suggests that despite this, a DP approach on a lumped state space often outperforms *other* heuristic procedures.

Under some situations, it is even possible to combine states without losing the Markov property (via the *lumpability result* [162]), but it cannot be guaranteed that the solution obtained from the lumped (reduced) state space is identical to that obtained without lumping. In other words, even here, optimality cannot be guaranteed. Still, however, state aggregation is a robust approach for breaking the curse of dimensionality and is often tried as the first attempt at function approximation. A general rule that we have used in our computational experiments is to combine states with "similar" characteristics together. The next example illustrates this idea. We will use $\vec{\mathsf{s}}$ to denote the state in this section.

Example 1: The state of the Markov chain is defined by $\vec{\mathsf{s}} = (\mathsf{s}(1), \mathsf{s}(2), \mathsf{s}(3))$, where $\mathsf{s}(1), \mathsf{s}(2)$ and $\mathsf{s}(3)$ are integers that take values from $\{0, 1, 2, \ldots, 49\}$. Without aggregation, one has 50^3 states here. Let us assume that the attributes $\mathsf{s}(1)$ and $\mathsf{s}(2)$ are more important than $\mathsf{s}(3)$. (How does one know which attribute is more important? Often, the problem's structure can help us answer this question. For instance, in problems associated with failure due to aging of systems, the age of the system is usually more important than other attributes.) Then, one way to aggregate states is to treat all the values of $\mathsf{s}(3)$ ranging from 0 to 24 as one value and the values ranging from 25 to 49 as another. For the other two attributes, $\mathsf{s}(1)$ and $\mathsf{s}(2)$, which are more important, we use a so-called *bucketing* scheme of a fixed length, say 5. With such a scheme, for our example, the values in the following groups are treated as one value:

$$(0 \leq \mathsf{s}(1) \leq 4); (5 \leq \mathsf{s}(1) \leq 9); \cdots ; (45 \leq \mathsf{s}(1) \leq 49).$$

Thus, for instance any value for $\mathsf{s}(1)$ that satisfies $5 \leq \mathsf{s}(1) \leq 9$ is treated as the same value. A similar strategy will be used for $\mathsf{s}(2)$. This will reduce our state space to only $(10)(10)(2) = 200$ states.

In the above example, states with similar values, or values in a given range of a state parameter, were combined. However, "similar" could also mean similar in terms of transition rewards and times. Exploiting similarity in terms of transition rewards can help us successfully extract the information hidden in a state space. Hence, states with very *dissimilar* transition rewards should *not* be combined. Since the TRMs (and TPMs) are unavailable, identifying states with similar transition rewards is usually not possible. However, from the nature of the problem, one can often identify the "good" states, which if entered result in high profits, and the "bad" states, which if entered result in losses. (The "ugly" states are those that are rarely visited! But for the time being, we will assume there are not too many of them, and that their effect on the overall performance of the policy can be disregarded). If it is possible to make such a classification, then aggregation should be done in a manner such that the good and bad states remain separate as much as is possible. Furthermore, sometimes, it is possible to devise a scheme to generate an entire spectrum of values such that its one end is "good" and the other "bad." The following example illustrates this idea.

Example 2. The state space is defined as in Example 1. But, we now define a scalar **feature** as follows:

$$\phi = 2\mathsf{s}(1) + 4\mathsf{s}(2) + 8\mathsf{s}(3), \tag{7.38}$$

where ϕ is a scalar. Let us assume that from the structure of the problem, we know that large values of ϕ imply good states and small values imply bad states. What we now have is a transformed state space into one defined in terms of ϕ. We will call the space generated by features the **feature space.** The process of creating a feature is called *encoding* in the neural network community. For an example of a feature in RL that identifies good and bad states in terms of rewards and losses, see [109].

A scheme such as the one shown above also aggregates states, but not necessarily in a *geometric* sense, as suggested in Example 1. In a geometric lumping, neighboring states are lumped together. Since geometric lumping can lump states with dissimilar transition rewards together, it is not always the best approach.

It is important to note that in the above example, via Eq. (7.38), we have mapped a 3-dimensional state space into a new state space (really a feature space) that has a lower dimension of one. Such mapping of state-spaces into lower dimensions is often very convenient in many forms of function approximation.

Feature-generating functions can be much more complicated than that above, and as a rule, they do not carry us from higher to a dimension of one. Consider the following functions that generate two features:

$$\phi(1) = 2\mathsf{s}^2(1) + 4\mathsf{s}(2) + 5\mathsf{s}(3); \phi(2) = 4\mathsf{s}(1) + 2\mathsf{s}^2(2) + 9\mathsf{s}(3); \quad (7.39)$$

where $\phi(\mathsf{l})$ denotes the feature in the lth dimension (or the lth feature) and is a scalar for every l. The ideas of feature creation are useful in all methods of function approximation, not just state aggregation. We will explore them further below.

Features and architecture. We now seek to generalize the ideas underlying feature creation. From here onwards, our discussion will be in terms of Q-factors or state-action values, which are more commonly needed than the value function in simulation-based optimization. Hence, the basis expression will denoted in terms of the feature *and* the action: $\phi(\mathsf{l}, a)$ for $\mathsf{l} = 1, 2, \ldots, \mathsf{n}$ and for every action a, where n denotes the number of features. We will assume that the state has d dimensions. Then, in general, the feature extraction map can be expressed as:

$$\begin{bmatrix} F_a(1,1) & F_a(1,2) & \cdot & \cdot & F_a(1,d) \\ F_a(2,1) & F_a(2,2) & \cdot & \cdot & F_a(2,d) \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ F_a(\mathsf{n},1) & F_a(\mathsf{n},2) & \cdot & \cdot & F_a(\mathsf{n},d) \end{bmatrix} \cdot \begin{bmatrix} \mathsf{s}(1) \\ \mathsf{s}(2) \\ \cdot \\ \cdot \\ \mathsf{s}(\mathsf{d}) \end{bmatrix} = \begin{bmatrix} \phi(1,a) \\ \phi(2,a) \\ \cdot \\ \cdot \\ \phi(\mathsf{n},a) \end{bmatrix},$$

where the matrix, whose elements are $F_a(.,.)$, denotes the feature extraction map for action a; the feature extraction map converts the state space to the feature space. Then the above matrix representation for Eq. (7.38) is:

$$\begin{bmatrix} 2 & 4 & 8 \end{bmatrix} \cdot \begin{bmatrix} \mathsf{s}(1) \\ \mathsf{s}(2) \\ \mathsf{s}(3) \end{bmatrix} = \phi;$$

Assuming that the representation in Eq. (7.39) is for some action a, the matrix representation for it will be:

$$\begin{bmatrix} 2\mathsf{s}(1) & 4 & 5 \\ 4 & 2\mathsf{s}(2) & 9 \end{bmatrix} \cdot \begin{bmatrix} \mathsf{s}(1) \\ \mathsf{s}(2) \\ \mathsf{s}(3) \end{bmatrix} = \begin{bmatrix} \phi(1,a) \\ \phi(2,a) \end{bmatrix}.$$

Basis functions. In the literature, the elements of the feature matrix, $\phi(.,.)$, go by the name **basis functions**. Thus, the features are essentially equivalent to the so-called basis functions. Although commonly called basis functions, it is important to recognize that these functions are fixed at the very beginning, to be never changed, and thus they are in reality fixed expressions. We will also call them basis *expressions*. Further, since they are expressions involving the state variables, their values depend on the values of the state variables involved.

It is common practice to first identify the expressions to be used for these so-called basis functions. A so-called "architecture" is then selected for the Q-factor-function. The architecture is an attempt to express the Q-factor as a function (usually linear) of the basis functions. The coefficients of basis functions in the architecture are the so-called weights of the architecture. While the weights change as the Q-function changes, the basis functions do not change. Hence, the basis functions have to be rich enough to capture the shape of the Q-function even as it changes. The weights and the basis expressions together define what is known as the architecture of the function approximator. We now illustrate this idea with what constitutes a typical *linear* architecture:

$$\begin{aligned} Q_{\vec{w}}(\vec{\mathsf{s}},a) &= w(1,a)\phi(1,a) + w(2,a)\phi(2,a) + \cdots + w(\mathsf{n},a)\phi(\mathsf{n},a) \\ &= \sum_{\mathsf{l}=1}^{\mathsf{n}} w(\mathsf{l},a)\phi(\mathsf{l},a), \end{aligned} \tag{7.40}$$

where $w(\mathsf{l},a)$ denotes the lth weight for action a. In the above, the weights are the only unknowns (remember the basis expressions are known) and must be estimated (and updated) via some approach (usually neurons or regression). If the architecture, such as the one above, fits the actual Q-function well, then instead of storing all the Q-factors, one now needs only n scalars for each action. Typically, n should be chosen so that it is much smaller than the size of the state space. Herein lies the power of function approximation. Function approximation enables us to replicate the behavior of look-up tables without storing all the Q-factors. Of course, this is easier said than done, because in practice, there are a number of challenges associated to it that we will discuss below. See Fig. 7.5 for a pictorial representation of the ideas underlying generation of an architecture.

With such a representation, any time the Q-factor of a state-action pair is to be updated, one determines the values of the basis expressions associated to the state-action pair, and then plugs them into the equation, e.g., Eq. (7.40), to obtain the Q-factor's value. When the value is updated, the change is transmitted back to update the terms of the weights (the $w(.,.)$-terms). Ideally, the updating of the weights should ensure that every time the Q-factor's value is evaluated via the equation above, the value returned is approximately equal to the value a look-up table would have produced. Whether this actually happens depends on (i) how suitable the architecture chosen was and (ii) how appropriate the updating of weights was. Clearly, there are no guarantees on either aspects.

A linear architecture in feature space is popular for many reasons, including mathematical simplicity. It can also capture a non-linear Q-function. Our next example illustrates this idea.

Example 3. For an MDP with 2 actions in each state, for state $i \in \mathcal{S}$,

$$Q_{\vec{w}}(i,1) = \mathsf{F}(i) \text{ and } Q_{\vec{w}}(i,2) = \mathsf{G}(i).$$

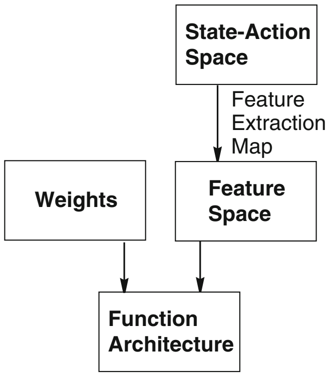

Figure 7.5. Generating the architecture: The state-action space is transformed into the feature space, which, together with the weights, defines the function's architecture

Assuming i to be a scalar, we use the following functions to approximate the Q-factors:

$$\mathsf{F}(i)=w(1,1)+w(2,1)i+w(3,1)i^2; \;\; \mathsf{G}(i)=w(1,2)+w(2,2)i+w(3,2)i^2+w(4,2)i^3,$$

where i^k denotes i raised to the kth power. Here, we will assume that to obtain n features, the one-dimensional state i will be written as:

$$\vec{\mathsf{s}} = [i\ 0\ 0 \ldots\ 0]^T, \text{ with } (\mathsf{n}-1) \text{ zeroes.}$$

As stated above, the non-linear state space of the approximation in the example above can be represented in terms of linear features. And here is how we do it for this example: For $\mathsf{F}(.)$, we can write as follows:

$$\begin{bmatrix} \frac{1}{i} & 0 & 0 \\ 1 & 0 & 0 \\ i & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} i \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} \phi(1,1) \\ \phi(2,1) \\ \phi(3,1) \end{bmatrix},$$

$$\text{i.e., } \phi(1,1) = 1; \quad \phi(2,1) = i; \quad \phi(3,1) = i^2.$$

A similar representation is possible for $\mathsf{G}(.)$. The above leads to a linear architecture in terms of features:

$$\mathsf{F}(i) = w(1,1)\phi(1,1) + w(2,1)\phi(2,1) + w(3,1)\phi(3,1).$$

Thus, a linear feature space may hide a non-linear Q-function. Obtaining a linear feature space from a non-linear state space is often advantageous from the computational perspective. Another approach for capturing a non-linear function within a linear feature space is the use of the so-called *radial basis function* in RL (see e.g., [267]).

8.2. Function Fitting

We must now employ a method for fitting the function with the architecture and feature space selected, i.e., determine the values of the weights. In general, we have two choices for the method: regression or neural networks. Each method has its advantages and drawbacks. We have already discussed these topics in some depth in Chap. 4.

If the Q-function is linear or approximately linear in the feature space, it may be approximated either by incremental linear regression or a neuron. There is no way of knowing apriori if the Q-function is linear. If it is indeed linear, a linear function approximation should lead us to good results with our RL algorithm. If the results produced turn out to be inferior to those from some other benchmarking heuristic, it is an indication that perhaps the features identified are unsuitable or else the Q-function is non-linear. Then, one should attempt to find either a new definition for features or a non-linear approximation.

Approximating a non-linear Q-function can be done in any one of the following ways:

1. Using *one* non-linear neural network over the entire state space with *backpropagation* as our main tool.

2. Using a pre-specified non-linear function to approximate the state space, e.g., Example 3, using regression as the main tool.

3. Using a *piecewise* linear function to approximate the state space with a *neuron* or linear regression within each piece. Example 4 (presented later) will illustrate this approach.

4. Using a *piecewise* non-linear function to approximate the state space with a *non-linear neural network* within each piece. This will amount to using several non-linear neural networks.

In general for function approximation to be successful, it must reproduce the behavior that RL with look-up tables would have produced (when the state space is large, think of an *imaginary* look-up table). In other words, any given Q-factor **must** remain at roughly the same value that look-up tables would have produced. The solution with the look-up tables, remember, is an optimal or a near-optimal solution.

Difficulties. We now describe the main challenges faced in function fitting.

1. **Changing Q-functions:** The Q-factors keep changing with every iteration (i.e., are non-stationary) in the look-up table. This implies that the weights must **also** change. Thus, it is not as though, if we somehow obtain the values of these weights during one iteration, we can then use them for every subsequent iteration. In fact, we must keep changing them *appropriately* as the algorithm progresses. In RL, whenever a Q-factor is updated, one data piece related to the value function (Q-function) becomes available. Recall from Chap. 4 that to approximate a function, whose data become available piece by piece, one can use incremental regression or incremental neural network methods. However, it is the case that for these incremental methods to work, all the data pieces must come from the **same** function. Unfortunately, the Q-function in RL keeps changing (getting updated) and the data pieces that we get are not really from the same function. Unfortunately, the structure or architecture of the value function is never known beforehand, and hence the data pieces arrive from a changing function.

2. **Spill over effect:** The spill-over effect arises from the fact that instead of updating $Q(i, a)$ (the look-up table case), we update

the weights that represent $Q(i,a)$. E.g., consider Eq. (7.40). The Q-factor is thus a function of the weights. When a state-action pair (i,a) is visited, the weights get updated. Unfortunately, since *every* Q-factor is a function of the weights, the update in weights causes *all* Q-factors to change, not just the (i,a)-th pair that was visited. Hence the new (updated) Q-factors may not equal the Q-factors that would have been produced by a look-up table, which would have changed only the (i,a)th Q-factor.

We refer to this kind of behavior, where the updates in one area of the state space spill over into areas where they are not supposed to, as the **spill-over effect**. With a look-up table, on the other hand, only *one* Q-factor changes in a given iteration, and the others remain unchanged. A function-approximation-coupled algorithm is supposed to imitate that coupled with a look-up table and should change *only one* Q-value in one iteration, leaving the others unchanged, at least approximately.

3. **Noise due to single sample:** In model-free RL, we use a single sample, instead of using the expectation, that tends to create noise in the updating algorithm. While this poses no problems with look-up tables, the noise can create a significant difficulty with function fitting (see [326, 317]). Unless a model is available apriori, this kind of noise cannot be avoided; see, however, [115] for some model-building algorithms that can partially avoid this noise. With the incremental least squares (regression) algorithm that we will discuss below, Bertsekas and Tsitsiklis [33] state that for Q-Learning, convergence properties are unknown. We also note that in general non-linear neural networks can get trapped in local optima and may display unstable behavior.

Incremental least squares and Bellman error. Incremental least squares or regression methods have always been popular in function approximation and are closely related to the incremental versions of Widrow-Hoff and backpropagation. We only present the central (heuristic) idea here.

Review the Q-factor version of the Bellman equation, i.e., Eq. (7.4). It is then clear that if $Q(i,a)$ is to be updated using function approximation, we would use the following definition for $Q(i,a)$:

$$Q(i,a) = \sum_{j=1}^{|\mathcal{S}|} p(i,a,j)\left[r(i,a,j) + \lambda \max_{b\in\mathcal{A}(j)} Q_{\vec{w}}(j,b)\right].$$

Now, the so-called Bellman error, BE, is defined to be the following:

$$BE = \frac{1}{2} \sum_{i \in \mathcal{S}, a \in \mathcal{A}(i)} \left[Q(i,a) - Q_{\vec{w}}(i,a) \right]^2 .$$

Clearly, the Bellman error denotes the sum of the squared differences between the actual Q-factors obtained from a look-up table and those given by the function approximator. Now, using the above definition of the Q-factor in the Bellman-error expression, we will compute the partial derivative of the Bellman error with respect to the weights. For easing notation, we will assume that the weights, which essentially form a matrix, can be stored in a vector, which will be denoted by $\vec{w}$. Thus, a Q-factor expressed in terms of this vector of weights will be denoted by $Q_{\vec{w}}(.,.)$. Then, we will use the derivative in a steepest-descent algorithm.

$$\begin{aligned}
\frac{\partial BE}{\partial w(\mathsf{l},a)} &= -\frac{\partial Q_{\vec{w}}(i,a)}{\partial w(\mathsf{l},a)} \cdot \sum_{i,a} \left[Q(i,a) - Q_{\vec{w}}(i,a) \right] \\
&= -\frac{\partial Q_{\vec{w}}(i,a)}{\partial w(\mathsf{l},a)} \times \\
&\quad \sum_{i,a} \left[\sum_{j=1}^{|\mathcal{S}|} p(i,a,j) \left[r(i,a,j) + \lambda \max_{b \in \mathcal{A}(j)} Q_{\vec{w}}(j,b) \right] - Q_{\vec{w}}(i,a) \right] \\
&= -\frac{\partial Q_{\vec{w}}(i,a)}{\partial w(\mathsf{l},a)} \times \\
&\quad \sum_{i,a} \left[\sum_{j=1}^{|\mathcal{S}|} p(i,a,j) \left[r(i,a,j) + \lambda \max_{b \in \mathcal{A}(j)} Q_{\vec{w}}(j,b) - Q_{\vec{w}}(i,a) \right] \right] .
\end{aligned}$$

We can now remove the terms $p(i,a,j)$ that need summation over j, thereby replacing an expectation with a single sample; this is similar to the use of Robbins-Monro algorithm for deriving Q-Learning. The end result is the following definition for the partial derivative:

$$\frac{\partial BE}{\partial w(\mathsf{l},a)} = -\left(\frac{\partial Q_{\vec{w}}(i,a)}{\partial w(\mathsf{l},a)} \sum_{i,a} \left[r(i,a,j) + \lambda \max_{b \in \mathcal{A}(j)} Q_{\vec{w}}(j,b) - Q_{\vec{w}}(i,a) \right] \right) .$$

The above will lead to a batch-updating algorithm that requires samples over all the Q-factors. If we use an incremental approach, where we use only one Q-factor, we can further simply this to:

$$\frac{\partial BE}{\partial w(\mathsf{l},a)} = -\left(\frac{\partial Q_{\vec{w}}(i,a)}{\partial w(\mathsf{l},a)} \left[r(i,a,j) + \lambda \max_{b \in \mathcal{A}(j)} Q_{\vec{w}}(j,b) - Q_{\vec{w}}(i,a) \right] \right) . \tag{7.41}$$

Now, the above can be combined with the following steepest descent algorithm:

$$w(\mathsf{l}, a) \leftarrow w(\mathsf{l}, a) - \beta \frac{\partial BE}{\partial w(\mathsf{l}, a)} \text{ for all } (\mathsf{l}, a), \tag{7.42}$$

where β is the step size. For using the above, we must determine the expressions for $\frac{\partial Q_{\vec{w}}(i,a)}{\partial w(\mathsf{l},a)}$, which can be done easily from the architecture. In general,

$$\frac{\partial Q_{\vec{w}}(i, a)}{\partial w(\mathsf{l}, a)} = \phi(\mathsf{l}, a),$$

and herein lies the convenience of defining an architecture linear in terms of the weights. In other words, the derivatives equal the basis functions, whose values are readily calculable. For example, in Example 3, we will obtain the following expressions for $\frac{\partial Q_{\vec{w}}(i,a)}{\partial w(\mathsf{l},a)}$ for $a = 1$:

$$\frac{\partial Q_{\vec{w}}(i, a)}{\partial w(1, a)} = 1; \quad \frac{\partial Q_{\vec{w}}(i, a)}{\partial w(2, a)} = i; \quad \frac{\partial Q_{\vec{w}}(i, a)}{\partial w(3, a)} = i^2.$$

The combination of Eqs. (7.41) and (7.42) is a popular algorithm in function approximation for RL. The central idea under the above derivation is to minimize the Bellman error, which is from Werbös [314] (see also [315]), where it was presented for a single policy for policy iteration. The above formula (7.41) is closely related to that found in [13, 51], where it is based on temporal difference formula in [285]—the so-called least squares temporal difference (LSTD) formula for one-step updates. The framework of temporal differences can be extended from one-step updates to multi-step updates. An example of an algorithm that employs a single-step update is Q-Learning in which the immediate reward from a single step is used for updating purposes. Multi-step updates have not been covered in this book, and hence we do not pursue this topic any further, but refer the reader to [285, 33] for extensions of the above formula to multi-step updates.

It should be clear that the equations above can be easily adapted for average reward, Q-P-Learning, and SMDP algorithms. For example, for the CF-version of R-SMART, the term $\lambda \max_{b\in\mathcal{A}(j)} Q_{\vec{w}}(j, b)$ in Eq. (7.41) should be replaced by

$$\eta \max_{b\in\mathcal{A}(j)} Q_{\vec{w}}(j, b) - \rho t(i, a, j),$$

where ρ is a separate scalar updated as shown in R-SMART's steps. For the CF-version of Q-P-Learning for average reward SMDPs, the same term would be replaced by

$$\eta Q\left(j, \arg\max_{b \in \mathcal{A}(j)} P_{\vec{w}'}(j,b)\right) - \rho t(i,a,j);$$

where $\vec{w}'$ denotes the weights of the P-factors and ρ denotes average reward of the policy contained in the P-factors and is computed prior to updating the Q-factors. Similarly, for the discounted reward Q-P-Learning algorithm for MDPs, that term would be replaced by

$$\lambda Q\left(j, \arg\max_{b \in \mathcal{A}(j)} P_{\vec{w}'}(j,b)\right),$$

where, again, $\vec{w}'$ denotes the weights of the P-factors.

Q-Learning coupled with function fitting. The mechanism used to couple a neural network or incremental regression with an RL algorithm will now be illustrated with the Q-Learning algorithm (for discounted MDPs). For other algorithms, it is very analogous to what we present below. The reader should review the material related to neural networks from Chap. 4 at this stage.

If the feature is defined as an n-tuple, a neural network or the incremental regression method needs n inputs and one additional input for the bias. In the following discussion, although we will refer to the state or feature as i or j, it will be understood to be an n-tuple whenever needed.

Remark: It is preferable in practice to use a separate function approximator *for each action.* Also, usually, an incremental style of updating is used, since the information for the Q-factor becomes available one data piece at a time.

We now discuss the steps that have to be followed in general for using RL in combination with a neural network. Following the steps, we provide a simple example to illustrate those steps.

Step 1. Initialize the weights of the neural network for any given action to small random numbers. Initialize the corresponding weights of all the other neural networks to identical numbers. (This ensures that all the Q-factors for a given state have the same value initially.) Set k, the number of state transitions, to 0. Start system simulation at any arbitrary state.

Step 2. Let the state be i. Simulate action a with a probability of $1/|\mathcal{A}(i)|$. Let the next state be j.

Step 3. Determine the output of the neural network for i and a. Let the output be called Q_{old}. Also find, using state j as the input, the outputs of all the neural networks of the actions allowed in state j. Call the maximum of those outputs: Q_{next}.

Step 3a. Update Q_{old} as follows (this is the standard Q-Learning algorithm).

$$Q_{\text{new}} \leftarrow (1-\alpha)Q_{\text{old}} + \alpha\left[r(i,a,j) + \lambda Q_{\text{next}}\right].$$

Step 3b. Then use Q_{new} and Q_{old} to update the weights of the neural network associated with action a via an incremental algorithm. In the neural network, Q_{new} will serve as the "target" value (y_p in Chap. 4), Q_{old} as the output (o_p in Chap. 4), and i as the input.

Step 4. Increment k by 1. If $k < k_{\max}$, set $i \leftarrow j$; then go to Step 2. Otherwise, go to Step 5.

Step 5. The policy learned is stored in the weights of the neural network. To determine the action associated with a state, find the outputs of the neural networks associated with the actions that are allowed in that state. The action(s) with the maximum value for the output is the action dictated by the policy learned. Stop.

We reiterate that the neural network in **Step 3b** is updated in an incremental style. Sometimes in practice, just one (or two) iteration(s) of training is sufficient in the updating process of the neural network. In other words, when a Q-factor is to be updated, only one iteration of updating is done *within the neural network.* Usually too many iterations in the neural network can lead to "over-fitting." Over-fitting implies that the Q-factor values in parts of the state space other than those being trained are incorrectly updated (spill-over effect). It is important that training be localized and limited to the state in question.

Steps with incremental regression. Here the steps will be similar to the steps above with the understanding that the weights of the neural network are now replaced by those of the regression model; Step 3a will not be needed Step 3b will be different. In Step 3b, we will first compute the values of the Q-factors for (i,a) and those for all actions associated to state j by plugging in the input features into the regression model. Then a combination of Eqs. (7.41) and (7.42) will be used to update the weights.

Simple example with a neuron. We now discuss a simple example of Q-Learning coupled with a neuron using incremental updating on an MDP with two states and two actions. We will use $\mathsf{n} = 2$ where for both actions: $\phi(1, a) = 1$, and $\phi(2, a)$ will equal the state's value;

$$\text{i.e., conceptually } Q_{\vec{w}}(i,1){=}w(1,1){+}w(2,1)i; \quad Q_{\vec{w}}(i,2){=}w(1,2){+}w(2,2)i.$$

Remember that the above (basis function) representation is conceptual; we do not store $Q_{\vec{w}}(i,1)$ or $Q_{\vec{w}}(i,2)$ in the computer's memory. Only four scalars will be stored in the memory: $w(1,1)$, $w(2,1)$, $w(1,2)$, and $w(2,2)$.

Step 1. Initialize the weights of the neuron for action 1, i.e., $w(1,1)$ and $w(2,1)$, to small random numbers, and set the corresponding weights for action 2 to the same values. Set k, the number of state transitions, to 0. Start system simulation at any arbitrary state. Set $k_{\max}$ to a large number.

Step 2. Let the state be i. Simulate action a with a probability of $1/|\mathcal{A}(i)|$. Let the next state be j.

Step 3. Evaluate the Q-factor for state-action pair, (i, a), which we will call Q_{old}, using the following:

$$Q_{\text{old}} = w(1, a) + w(2, a)i.$$

Now evaluate the Q-factor for state j associated to each action, i.e.,

$$Q_{\text{next}}(1) = w(1,1) + w(2,1)j; \quad Q_{\text{next}}(2) = w(1,2) + w(2,2)j.$$

$$\text{Now set } Q_{\text{next}} = \max\left\{Q_{\text{next}}(1), Q_{\text{next}}(2)\right\}.$$

Step 3a. Update the relevant Q-factor as follows (via Q-Learning).

$$Q_{\text{new}} \leftarrow (1-\alpha)Q_{\text{old}} + \alpha\left[r(i,a,j) + \lambda Q_{\text{next}}\right]. \tag{7.43}$$

Step 3b. The current step in turn may contain a number of steps and involves the neural network updating. Set $m = 0$, where m is the number of iterations used within the neural network. Set $m_{\max}$, the maximum number of iterations for neuronal updating, to a suitable value (we will discuss this value below).

Step 3b(i). Update the weights of the neuron associated to action a as follows:

$$w(1,a) \leftarrow w(1,a) + \mu(Q_{\text{new}} - Q_{\text{old}})1; \quad w(2,a) \leftarrow w(2,a) + \mu(Q_{\text{new}} - Q_{\text{old}})i. \tag{7.44}$$

Step 3b(ii). Increment m by 1. If $m < m_{\max}$, return to Step 3b(i); otherwise, go to Step 4.

Step 4. Increment k by 1. If $k < k_{\max}$, set $i \leftarrow j$; then go to Step 2. Otherwise, go to Step 5.

Step 5. The policy learned, $\hat{d}$, is virtually stored in the weights. To determine the action prescribed in a state i where $i \in \mathcal{S}$, compute the following:

$$d(i) \in \arg\max_{a \in \mathcal{A}(i)} \left[w(1,a) + w(2,a)i \right].$$

Some important remarks need to be made in regards to the algorithm above.

Remark 1. Note that the update in Eq. (7.44) is the update used by an incremental neuron that seeks to store the Q-factors for a given action.

Remark 2. The step-size μ is the step size of the neuron, and it can be also be decayed with every iteration m (see Chap. 4).

Remark 3. The value of $m_{\max}$ equals the number of iterations for which the neuron should be run after each iteration of the Q-Learning algorithm. One should not set very large values for $m_{\max}$ in order to avoid overfitting (see Chap. 4). In practice, $m_{\max}$ can be as low as 2.

Remark 4. It is **strongly recommended** that the state's value (i or j) be normalized to the interval $(0,1)$ in all the calculations above.

Remark 5. We used the linear architecture in which $n = 1$ in Eq. (7.40) with $f(1) = i$. It is certainly possible to use a more involved function approximation architecture in the above.

Remark 6. Note that Eq. (7.43) yields

$$Q_{\text{new}} - Q_{\text{old}} = \alpha \left[r(i,a,j) + \lambda Q_{\text{next}} - Q_{\text{old}} \right],$$

$$\text{i.e., } \mu \left[Q_{\text{new}} - Q_{\text{old}} \right] = \mu\alpha \left[r(i,a,j) + \lambda Q_{\text{next}} - Q_{\text{old}} \right].$$

If we set $\beta = \mu\alpha$, the update in (7.44) becomes equivalent to that in Eqs. (7.41) and (7.42), providing yet another perspective on the Bellman error update: a link between the incremental neuron and the Bellman error update.

Unfortunately, function fitting (i.e., neural networks or regression) has not always worked well in a number of well-publicized experiments in RL; see e.g., [50]. We will now discuss an alternative to function fitting.

Example with incremental regression. We now present show how incremental regression (or Bellman error) can be used on the same simple example studied above in which the basis functions for a state i are: $\phi(1,a) = 1$ and $\phi(2,a) = i$ for both actions. This implies that:

$$\frac{\partial Q_{\vec{w}}(i,a)}{\partial w(1,a)} = 1; \quad \frac{\partial Q_{\vec{w}}(i,a)}{\partial w(2,a)} = i \tag{7.45}$$

The steps will be similar to those described for the neuron with the following difference in Step 3.

Step 3. Evaluate Q_{old} and Q_{next} as discussed above in Step 3 of the neuron-based algorithm. Then, update the weights as follows:

$$w(1,a) \leftarrow w(1,a) + \mu \left(r(i,a,j) + \lambda Q_{\text{next}} - Q_{\text{old}} \right) 1;$$
$$w(2,a) \leftarrow w(2,a) + \mu \left(r(i,a,j) + \lambda Q_{\text{next}} - Q_{\text{old}} \right) i.$$

Remark: It is to be noted that the above update is derived from using the definition of the basis functions in (7.45) in combination with Eqs. (7.41) and (7.42).

Some final thoughts. We conclude this section with some advice based on our limited computational experience. We have found the following strategy to be robust for function approximation. First, identify a suitable feature space. Divide the feature space into compartments, and place a neuron (or a non-linear neural network) within each compartment. (See Fig. 7.6.) Even with neurons, this will most likely lead to a non-linear approximation of the Q-function, which is actually piecewise linear, of the feature space. Because of the separation, updating is localized to within the compartment, thus minimizing the damage produced by the spill-over effect. Choosing the size of the compartment requires trial and error. The following example illustrates these ideas.

Example 4. Consider a one-dimensional state space defined by i where $i \in \{1, \ldots, 9\}$. We now use a very simple partitioning of the state space into two compartments for action a. Then, a piecewise linear representation with two compartments separated at $i = 5$ would be

$$Q(i,a) = w_1(1,a) + w_1(2,a)i \text{ for } i \leq 5; \; Q(i,a) = w_2(1,a) + w_2(2,a)i \text{ for } i > 5,$$

where $w_c(.,a)$ denotes the weight in the cth compartment for action a. The training for points in the zone $i \leq 5$ would not spill over into the training for points in the other zone. Clearly, one can construct multiple compartments, and compartments do not have to be of the same size.

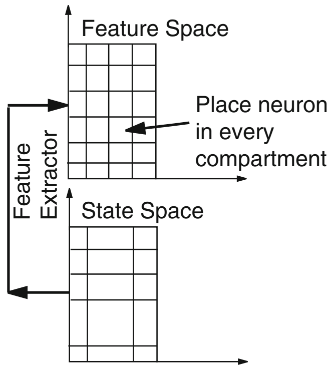

Figure 7.6. A feature extraction mapping that transforms the actual 2-dimensional state space (bottom) into a more regular feature space: Within each compartment in the feature space, a neuron can be placed

9. Conclusions

This chapter was meant to serve as an introduction to the fundamental ideas related to RL. Many RL algorithms based on Q-factors were discussed. Their DP roots were exposed and step-by-step details of some algorithms were presented. Some methods of function approximation of the Q-function were discussed. Brief accounts of model-building algorithms and finite horizon problems were also presented

At this point, we summarize the relationship between RL algorithms and their DP counterparts. The two main algorithms of DP, value and policy iteration, are based on the Bellman equation that contains the elements of the **value function** as the unknowns. One can, as discussed in this chapter, derive a Q-factor version of the Bellman equation. Most RL algorithms, like Q-Learning and Q-P-Learning, are based on the Q-factor version of the Bellman equation. Table 7.3 shows the DP roots of some of the RL algorithms that we have discussed in this chapter.

Bibliographic Remarks: In what follows, we have attempted to cite some important references related to research in RL. In spite of our best efforts, we fear that this survey is incomplete in many respects, and we would like to apologize to researchers whom we have not been able to acknowledge below.

Early works. Barto et al. [18], Barto and Anandan [16], Watkins [312], Werbös [314], and Rummery and Niranjan [258] are some of the initial works in RL. The work of Werbös [314] and Sutton [285] played an influential role in formulating the ideas of policy iteration, function approximation, and temporal differences within RL. The history can be traced to earlier works. The idea of learning can be found

Table 7.3. The relationship between DP and RL

DP	RL
Bellman optimality equation (Q-factor version)	Q-Learning
Bellman policy equation (Q-factor version)	Q-P-Learning
Value iteration	Q-Learning
Relative value iteration	Relative Q-Learning
Modified policy iteration	Q-P-Learning and CAP-I

in Samuel [260] and Klopf [176]. Holland [140] is also an early work related to temporal differences. Some other related research can be found in Holland [141] and Booker [44].

Textbooks. Two textbooks that appeared before the one you are reading and laid the foundation for the science of RL are [33] and [288].

Neuro-dynamic programming (NDP) [33], an outstanding book, strengthened the connection between RL and DP. This book is strongly recommended to the reader for foundational concepts on RL. It not only discusses a treasure of algorithmic concepts likely to stimulate further research in coming years, but also presents a detailed convergence analysis of many RL algorithms. The name NDP is used to emphasize the connection of DP-based RL with function approximation (neural networks).

The book of Sutton and Barto [288] provides a very accessible and intuitive introduction to reinforcement learning, including numerous fundamental ideas ranging from temporal differences, through Q-Learning and SARSA, to actor-critics and function approximation. The perspective of machine learning, rather than operations research, is used in this text, and the reader should find numerous examples from artificial intelligence for illustration.

The reader is also referred to Chapter 6 in Vol II of Bertsekas [30], which focusses on Approximate Dynamic Programming (ADP) and discusses a number of recent advances in this field, particularly in the context of function approximation and temporal differences. The acronym ADP, which is also used to mean Adaptive Dynamic Programming, is often used to refer to simulation-based DP and RL schemes for solving MDPs that employ regression-based function approximators, e.g., linear least squares. It was coined in Werbös [318] and is being used widely in the literature now.

More recently, a number of books have appeared on related topics, and we survey a subset of these. Chang et al. [62] discuss a number of recent paradigms, including those based on stochastic policy search and MRAS for simulation-based solutions of MDPs. Szepesvári [289] presents a very crisp and clear overview of many RL algorithms. Numerous other books contain related material, but emphasize specific topics: function approximation [56], stochastic approximation [48], sensitivity-based learning [57], post-decision-making [237], and knowledge-gradient learning [238].

Reinforcement Learning. The Q-Learning algorithm for discounted reward MDPs is due to Watkins [312]. This work appears to have established the link between DP and RL for the first time in the context of value iteration. However, Werbös [314] had argued for this in an interesting earlier work in 1987 in the context of policy iteration. Indeed, the work in [314] was influential in that it led to widespread use of the gradient within RL function approximation for RL and also to approximate policy iteration.

Modified Q-Learning for discounted reward has its origins in the algorithm of Rummery and Niranjan [258]. A closely related algorithm is known as SARSA [287]; see also Sutton and Barto [288] where the policy evaluation phase (episode) of SARSA is imbedded within a so-called generalized policy iteration and hence may be composed of many iterations of value iteration. These ideas been formalized and made mathematically rigorous in the treatment given in [33], where it is called *approximate policy iteration.* Our presentation of Q-P-Learning for discounted reward [118] is consistent with the description in [33] when Q-factors are used. The Relative Q-Learning algorithm for average reward MDPs is due to Abounadi et al. [2]. The Q-P-Learning algorithm for average reward MDPs, which uses relative value iteration in the policy evaluation phase, is new material in this book.

R-SMART was first presented in Gosavi [110] (see also A. Gosavi, An algorithm for solving semi-Markov decision problems using reinforcement learning: convergence analysis and numerical results. Ph.D. dissertation, Department of Industrial and Management Systems Engineering, University of South Florida, Tampa, 1999, Unpublished). The SSP and CF versions presented here are however from Gosavi [119]. SMART, which does not use learning rates for updating ρ, is due to [72]. R-Learning, one of the first algorithms in average reward RL, is due to Schwartz [268]. An overview of early RL algorithms can be found in [157, 195]. Some recent reviews can be found in Gosavi [114, 122].

A SMDP algorithm based on value iteration that uses a continuous rate of reward and disregards lump sum reward can be found in Bradtke and Duff [52]. Our discussion here includes the lump sum reward and is based on Gosavi [119]. Q-P-Learning for average reward SMDPs first appeared in Gosavi [109]. However, the algorithms presented here for the SSP-version (average reward case) and the discounted reward case (that considers both the lump sum and continuous reward rate) are from Gosavi [118].

The model-building RTDP algorithm for discounted reward and its average reward counterpart (called h-Learning) can be found in [17] and Tadepalli and Ok [292], respectively. Other works on model building include [286, 211, 160, 53, 283, 79, 307]. Model-building algorithms of the Q-Learning variety (which learn Q-factors rather than the value function) presented in this book were new material in the first edition; they are now covered in Gosavi [115, 117]; see also Gosavi et al. [125] for a model-building adaptive critic. Model-building algorithms have been used for robotic soccer [324], helicopter control [218, 1, 179], function magnetic imaging resonance (fMRI) studies of brain [331, 150], and vision [204].

Function Approximation. A number of papers related to function approximation with classical DP are: [25, 301, 156, 228]. For function approximation in RL and ADP, see [314, 13, 51, 50, 75, 219, 322, 196, 321]. For a general (not including the RL context) discussion on function approximation schemes, including nearest neighbor methods, kernel methods, and neural networks, see [131, 78]. In the context of RL, function approximation based on neural networks requires encoding; see Hinton [135] (coarse coding), Albus [4] (CMAC coding), and Kanerva [158] (kanerva coding).

Stochastic Adaptive Search. Recently, there has been much interest in using static optimization techniques, based on stochastic adaptive search/meta-heuristics, to solve MDPs via simulation. Some noteworthy works include [63, 64, 61, 145] (based on simulated annealing, genetic algorithms and learning automata, etc.) and [146, 147] (based on MRAS). Much of this work has been summarized in [62].

Other references. Some other works related to RL are the work of Jalali and Ferguson [153], temporal differences in value iteration [312, 224, 51, 123], and temporal differences in policy iteration [187].

Finally, we must cite some work that laid the foundation of RL. The pioneering early work of Robbins and Monro [247] forms the underlying basis for almost all RL algorithms. The Robbins-Monro algorithm has been combined with value iteration (the work of Bellman [24]) to derive Q-Learning and with policy iteration (Howard [144]) and modified policy iteration (van Nunen [305]) to derive approximate policy iteration or Q-P-Learning.

Case study on total productive maintenance. This case study has been described at the end of Chap. 6. Here we illustrate the use of the CF-version of R-SMART using the same case study. The following values were used in the simulator: $\eta = 0.99$; $\alpha^k = \frac{1{,}000}{5{,}000+k}$; $\beta^k = \frac{1{,}000}{k(5{,}000+k)}$, where $k \geq 1$. The action selection probability was defined as shown below for all i:

$$\mathsf{p}^k(i) = 0.5\frac{log(k+1)}{k+1} \text{ where } k \geq 1.$$

The system was simulated for 200,000 days. From the Q-factors, one can identify the policy delivered by the algorithm. The algorithm always generated the optimal solution of producing for values of i less than or equal to 5 and maintaining from there onwards.

Note that to identify the policy delivered by the algorithm, one is only interested in the lowest value of i at which the algorithm selects the maintain action. If we denote this value of i by i^*, the algorithm will not permit the system to enter any state beyond that, i.e., any value of $i > i^*$. Hence, actions selected for states $i > i^*$ are not of any interest to us when examining the Q-factors generated by the RL algorithm. In the example above, i^* turned out to be 6.

Chapter 8

CONTROL OPTIMIZATION WITH STOCHASTIC SEARCH

1. Chapter Overview

In this chapter, we discuss an approach for solving Markov decision problems (MDPs) and Semi-Markov decision problems (SMDPs) using an approach that employs the so-called action-selection probabilities instead of the Q-factors required in reinforcement learning (RL). The underlying philosophy of this approach can be explained as follows. The action-selection probabilities, which are stored in some form either directly or indirectly, are used to guide the search. As a result, we have a stochastic search in which each action is considered to be equally good at the start, but using feedback from the system about the effectiveness of each action, the algorithm updates the action-selection probabilities—leading the system to the optimal policy at the end. It should be clear to the reader that like RL, this approach also uses feedback from the system, but unlike RL, it stores action-selection probabilities.

The two methods in the class of "stochastic search" that we cover are widely known as learning automata (or automata theory) and actor critics (or adaptive critics). Automata theory for solving MDPs/SMDPs will be referred to as **M**arkov **C**hain **A**utomata **T**heory (MCAT) in this book. MCAT does not use the Bellman equation of any kind, while actor critics use the Bellman policy equation (Poisson equation). In Sect. 2, we discuss MCAT, and in Sect. 3, we discuss actor critics, abbreviated as **AC**s. MCAT will be treated for average

A. Gosavi, *Simulation-Based Optimization*, Operations Research/Computer Science Interfaces Series 55,
DOI 10.1007/978-1-4899-7491-4_8

reward MDPs and SMDPs, while ACs will be treated for discounted and average reward MDPs and average reward SMDPs. It is recommended that the reader become familiar with the basics of MDPs and SMDPs from Chap. 6 before reading this chapter.

2. The MCAT Framework

MCAT provides a simulation-based methodology to solve MDPs and SMDPs; furthermore, as stated above, it does not use the dynamic programming framework, unlike RL.

We will focus on one specific type of MCAT algorithm that is used to solve average reward MDPs and SMDPs. We introduce some standard notation now.

One associates with each state-action pair, (i, a), the so-called action-selection probability to be denoted by $p(i, a)$. Let $\mathcal{A}(i)$ denote the set of actions allowed in state i. The set of states is denoted by $\mathcal{S}$.

As mentioned above, in the beginning of the learning process, each action is equally likely. Hence, for each $i \in \mathcal{S}$ and each $a \in \mathcal{A}(i)$,

$$p(i, a) = \frac{1}{|\mathcal{A}(i)|},$$

where $|\mathcal{A}(i)|$ denotes the number of actions allowed in state i. The updating process employs the following intuitive idea. The simulator simulates a trajectory of states. In each state, actions are selected using the action-selection probabilities. If the performance of an action is considered to be *good* based on the rewards generated, the probability of that action is *increased*. Of course, this "updating" scheme must always ensure that the sum of the probabilities of all the actions in a given state is 1, i.e.,

$$\sum_{a \in \mathcal{A}(i)} p(i, a) = 1 \qquad i \in \mathcal{S}.$$

Under some conditions on the MDP/SMDP, the probability of one action, the optimal action, converges to 1, while that of each of the other actions converges to 0. If m actions are optimal, the probability of each of the optimal actions should converge to $1/m$.

We now provide an overview of the feedback mechanism. Details are provided in the steps of the algorithm. For the algorithm to work, each state must be visited infinitely often. Consider the scenario in which state i has been revisited, i.e., it was visited at least once before. The algorithm needs to know the value of the average reward earned by the system since its last visit to i. This value is also called the **response** of the system to the action selected in the last visit to state i.

The response, as we will see later, is used to generate the feedback used to update the action selection probabilities. The response is calculated as the total reward earned since the last visit to i divided by the number of state transitions since the last visit to i. Thus, for MDPs, the response for state i is given by

$$s(i) = \frac{R(i)}{N(i)},$$

where $R(i)$ denotes the *total* reward earned since the last visit to state i and $N(i)$ denotes the number of state transitions that have occurred since the last visit to i. For the SMDP, $N(i)$ is replaced by $T(i)$, where $T(i)$ denotes the *total* time spent in the simulator since the last visit to i. Thus, for SMDPs, the response for state i is:

$$s(i) = \frac{R(i)}{T(i)}.$$

The response is then **normalized** to convert it into a scalar quantity that lies between 0 and 1. The normalized response is called **feedback**. The normalization is performed via:

$$\phi(i) = \frac{s(i) - s_{\min}}{s_{\max} - s_{\min}}, \tag{8.1}$$

where $s_{\min}$ is the *minimum* response possible in the system and $s_{\max}$ is the *maximum* response possible in the system.

As stated above, the feedback is used to update the action-selection probability $p(i, x)$ where x denotes the action selected in the last visit to i. Without normalization, we will see later, the updated action-selection probabilities can exceed 1 or become negative.

Many schemes have been suggested in the literature to update the action-selection probabilities. All schemes are designed to punish the bad actions and reward the good ones. A popular scheme, known as the Reward-Inaction scheme, will be covered in detail because it appears to be one of the more popular ones [215].

Reward-Inaction Scheme. As discussed above, a trajectory of states is simulated. An iteration is said to be performed when one transitions from one state to another. Consider the instant at which the system visits a state i. At this time, the action-selection probabilities for state i are updated. To this end, the feedback $\phi(i)$ has to be computed as shown in Eq. (8.1). Let $L(i)$ denote the action taken in the *last* visit to state i and α denote the step size or learning rate. Let $p^k(i, x)$ denote the action-selection probability of action x in state i in the kth iteration of the algorithm. For the sake of simplicity, let us assume for the time being that no more than two actions per state are allowed. At the very beginning, before starting the simulation, one sets x to any of the two actions. The value of x is not changed later during the algorithm's progress. Then, using the Reward-Inaction scheme, the action-selection probability of an action x is updated via the rule given below:

$$p^{k+1}(i,x) \leftarrow p^k(i,x) + \alpha\phi(i)I(L(i)=x) - \alpha\phi(i)p^k(i,x), \qquad (8.2)$$

where $I(.)$ in the indicator function that equals 1 if the condition inside the brackets is satisfied and is 0 otherwise. Note that the above scheme is used to update an action x out of the two actions numbered 1 and 2. Clearly, after the above update, the other action will have to be set to $\big(1 - p^{k+1}(i,x)\big)$.

The reward-inaction scheme is so named because a good action's effects are rewarded while those of a poor action are ignored (inaction). How this is ensured can be explained as follows. Assume that x is the action that was selected in the last visit to i. Then, the change in the value of $p(i,x)$ will equal $\phi \times \alpha[1 - p(i,x)]$. Thus, if the response is strong (feedback close to 1), the probability will be increased significantly, while if the response is weak (feedback close to 0), the increase will not be very significant. Likewise, if the action x was not selected in the last visit, the change will equal $-\phi \times \alpha p(i,x)$. Hence, if the action was not selected, but the response was strong, its probability will be reduced significantly, but if the response was weak, its probability will not see a significant change.

Now, we present step-by-step details of an MCAT algorithm.

2.1. Step-by-Step Details of an MCAT Algorithm

Before presenting the steps, we would like to make the following comments. Our presentation is for average reward SMDPs using the reward-inaction scheme. The MDP is a special case of the SMDP; one can replace time spent in each transition by 1 in the SMDP to obtain the MDP. The algorithm description presented below assumes that there are two possible actions in each state. We will discuss a generalization to multiple actions later. Also, we will drop the superscript k from $p^k(i,x)$ to increase clarity.

Steps. Set the number of iterations, k, to 0. Initialize action probabilities $p(i,a) = \frac{1}{|\mathcal{A}(i)|}$ for all $i \in \mathcal{S}$ and $a \in \mathcal{A}(i)$. Set the cumulative reward earned and the cumulative time spent until the last visit to i,

$C_r(i)$ and $C_t(i)$, respectively, to 0 for all $i \in \mathcal{S}$. Also, initialize to 0 the following quantities: the total reward earned in system from the start, TR, and the total time spent in the system from the start, TT. Initialize $s_{\min}$ and $s_{\max}$ as described above. Set x to any action allowed in the system. The value of x is never changed. Start system simulation. Let the starting state be denoted by i. Select a small (suitable) value for α.

1. If this is the first visit to state i, go to Step 3. Otherwise, compute R, the total reward earned in the system since the last visit to i, and T, the total time spent in the system since the last visit to i, as follows:

$$R = TR - C_r(i); \quad T = TT - C_t(i).$$

Then, compute the response and feedback as follows:

$$s = \frac{R}{T}; \quad \phi = \frac{s - s_{\min}}{s_{\max} - s_{\min}}.$$

2. Let $L(i)$ denote the action that was selected in the last visit to i. Update $p(i,x)$ using:

$$p(i,x) \leftarrow p(i,x) + \alpha\phi I(L(i) = x) - \alpha\phi p(i,x),$$

where $I(.)$ is the indicator function that equals 1 when the condition within brackets is satisfied and equals 0 otherwise. Then, update the other action (action other that x) so that sum of the probabilities of actions for state i is 1.

3. With probability $p(i,a)$, select an action a from the set $\mathcal{A}(i)$.

4. Set $L(i) \leftarrow a$, $C_r(i) \leftarrow TR$, and $C_t(i) \leftarrow TT$. Then, simulate action a. Let the next system state be j. Also let $t(i,a,j)$ denote the (random) transition time, and $r(i,a,j)$ denote the immediate reward earned in the transition resulting from selecting action a in state i.

5. Set $TR \leftarrow TR + r(i,a,j)$; $TT \leftarrow TT + t(i,a,j)$.

6. Set $i \leftarrow j$ and $k \leftarrow k+1$. If $k < MAX\_STEPS$, return to Step 1; otherwise STOP.

Remark 1. The algorithm requires knowledge of $s_{\max}$ and $s_{\min}$, which can sometimes be obtained from those values of the transition reward matrix that are known. When these values are unknown, one must come up with conservative estimates of bounds. Unfortunately, if the bounds used are too conservative, the rate of updating is slowed down since the resulting feedback values are far from 1 even when the response is good, or from 0 when the response is weak.

Remark 2. Termination criterion: One can run the algorithm until all the probabilities have converged to 0 or 1 if one is certain that there is a unique optimal action. This may take a very long time, however. A second approach is to run the algorithm until one of the probabilities in each state exceeds a pre-fixed "safe" value, e.g., 0.9. This action is then declared to be the optimal action for that state. A third approach, popular in practice, runs the algorithm for a large number of iterations and selects the most likely action as the optimal in each state.

Remark 3. The step size (learning rate), α, is fixed and need not be diminished with iterations. This is a remarkable feature of this algorithm. Convergent constant step size algorithms [319] are rare in stochastic approximation.

Remark 4. Selection of x: Note that in the description above, we use the same value of x for every state. However, it is acceptable to have a different x for each state. Then, x would be replaced by $x(i)$. Thus for instance in a system with two actions in each state, one could have $x(1) = 2$; and $x(2) = 1$.

2.2. An Illustrative 3-State Example

In this section, we will show how the MCAT algorithm, discussed above, works on a simple 3-state SMDP. The example comes from Gosavi et al. [124]. For the sake of simplicity, we will first assume that each state has two possible actions.

Obviously, we will not require the values of the transition probabilities of the underlying Markov chain; a simulator of the system will be sufficient. Using the simulator, a trajectory of states is generated, and updating is performed for a state when the simulator visits it. Let the first few states in the trajectory be defined as:

$$2 \to 1 \to 3 \to 1 \to 2.$$

We will now show how updating is done for these states. In the beginning, all the action probabilities are supposed to be equal. Since there are two actions in each state, we set

$$p(1,1) = p(1,2) = p(2,1) = p(2,2) = p(3,1) = p(3,2) = 0.5.$$

The updating required by the algorithm is performed (within thesimulator), and the action chosen is simulated. The next state is generated by the simulator. Again, an action is chosen in the next state, the necessary updates are performed, and we move on to the next state. This continues until the termination criterion for the algorithm is met.

We begin by setting $C_r(i) = 0$ and $C_t(i) = 0, \forall i$. We also set TR and TT to 0. Also, we will assume that $x = 1$, $s_{\max} = 10$, $s_{\min} = -5$, and $\alpha = 0.1$. The updating calculations performed in each state are listed below.

- **State 2:** This is the first visit to state 2. Hence the action probabilities are not updated. Let the action selected, a, be 1. Then, following Step 4, we set $L(2) = 1$, $C_r(2) \leftarrow TR(= 0)$ and $C_t(2) \leftarrow TT(= 0)$. The next state is 1. Here $r(i, a, j) = 4.5$ and $t(i, a, j) = 2.34$ for $i = 2, j = 1$. (Notice that both values, 4.5 and 2.34, are generated by the simulator.) Next, following Step 6, we update TR and TT. New values for these variables are: $TR = 4.5$ and $TT = 2.34$. We are now in state 1.

- **State 1:** This is the first visit to state 1, and the calculations performed will be similar to those shown above. Let the action selected be 2. So, following Step 4, we set $L(1) = 2$, $C_r(1) \leftarrow TR(= 4.5)$ and $C_t(1) \leftarrow TT(= 2.34)$. The next state is 3. From the simulator, $r(i, a, j) = 3.5$, and $t(i, a, j) = 0.11$ for $i = 1, j = 3$. Following Step 6, next we must update TR and TT. The new values for these variables are: $TR = 4.5 + 3.5 = 8$ and $TT = 2.34 + 0.11 = 2.45$. We are now in state 3.

- **State 3:** This is the first visit to state 3 and once again the calculations performed will be similar to those shown above. Let the action selected be 2. Again, following Step 4, we set $L(3) = 2$, $C_r(3) \leftarrow TR(= 8)$ and $C_t(3) \leftarrow TT(= 2.45)$. The next state is 1. From the simulator, $r(i, a, j) = -1$, and $t(i, a, j) = 1.55$ for $i = 3, j = 1$. We next update TR and TT following Step 6. The new values for these variables are: $TR = 8 - 1 = 7$ and $TT = 2.45 + 1.55 = 4$. We are now in state 1.

- **State 1 (again):** This is a re-visit (second visit) to state 1. Therefore, we first need to execute Step 1. We compute: $R = TR - C_r(1) = 7 - 4.5 = 2.5$; $T = TT - C_t(1) = 4 - 2.34 = 1.66$,; $s = R/T = 2.5/1.66 = 1.5060$; $\phi = \frac{s - s_{\min}}{s_{\max} - s_{\min}} = \frac{1.5060+5}{10+5} = 0.4337$. Now, $L(1) = 2$; but since $L(1) \neq x = 1$, we update $p(1, 1)$:

  $$p(1, 1) \leftarrow p(1, 1) - \alpha\phi p(1, 1) = 0.5 - 0.1(0.4337)(0.5) = 0.4783$$

  Let the action selected in this state be 1. Therefore, $L(1) = 1$, $C_r(1) = 7$, and $C_t(1) = 4$. From the simulator, $r(i, a, j) = 2.4$, and $t(i, a, j) = 1.23$, for $i = 3, j = 1$. The new values are: $TR = 7 + 2.4 = 9.4$ and $TT = 4 + 1.23 = 5.23$. We are now in state 2.

- **State 2 (again):** This is a re-visit (second visit) to state 2. Hence we first execute Step 1. We set $R = TR - C_r(2) = 9.4 - 0 = 9.4$; $T = TT - C_t(2) = 5.23 - 0 = 5.23$; $s = R/T = 9.4/5.23 = 1.7973$; $\phi = \frac{s - s_{\min}}{s_{\max} - s_{\min}} = \frac{1.7973+5}{10+5} = 0.4531$. Now, $L(2) = 1$; but since $L(2) = x = 1$, we update $p(2, 1)$:

  $p(2,1) \leftarrow p(2,1)+\alpha\phi(1-p(2,1))=0.5+0.1(0.4531(1-0.5)) = 0.5226.$

  Let the action selected in this state be 2. Then, $L(2) = 2$, $C_r(2) = 9.4$, and $C_t(2) = 5.23$. Let the next state be 3. From the simulator, $r(i, a, j) = -1.9$ and $t(i, a, j) = 4.8$, where $i = 2, j = 3$. The new values are: $TR = 9.4 - 1.9 = 7.5$ and $TT = 5.23 + 4.8 = 10.03$.

Updating will continue in this fashion for a large number of iterations. The most likely action (the action with the largest probability) in each state will be considered to be the best action for that state.

2.3. Multiple Actions

Note that so far we have assumed that only two actions are allowed in each state. A number of ways have been suggested in the literature to address the case with more than two actions. We will describe one approach below.

If state i is the current state, update $p(i, x)$ as described earlier for the case with two actions. Now, let us denote the change in the probability, $p(i, x)$, by Δ, which implies that:

$$\Delta = p^{k+1}(i, x) - p^k(i, x).$$

Clearly, Δ may be positive or negative. Now, if the total number of actions in state i is m, where $m > 2$, then update all probabilities other than $p(i, x)$ using the following rule:

$$p^{k+1}(i, a) \leftarrow p^k(i, a) - \frac{\Delta}{m - 1}, \text{ where } a \neq x.$$

Unfortunately, this penalizes or rewards several actions equally, which is a deficiency. More intelligent schemes have also been suggested in the literature (see [215]).

3. Actor Critics

Actor critics (ACs), also called adaptive critics, have a rather long history [328, 18, 316]. Like MCAT, ACs use action-selection probabilities to guide the search, but like RL, they also use the Bellman equation—the Bellman policy equation (Poisson equation) in particular. For each state-action pair, (i, u), one stores $H(i, u)$, a surrogate for the action-selection probability. An action a is selected in state i with the probability $p(i, a)$ where

$$p(i,a) = \frac{e^{H(i,a)}}{\sum_{b \in \mathcal{A}(i)} e^{H(i,b)}}. \tag{8.3}$$

The above is called the Gibbs softmax method of action selection. The terms $H(.,.)$ hence dictate the action-selection probabilities. Like in MCAT, $H(i, a)$ is updated on the basis of the feedback from the system related to choosing action a in state i, but unlike in MCAT and like in RL, the feedback exploits a Bellman equation. Unfortunately, $H(i, a)$ can become unbounded, and since it is used as the power of an exponential in the Gibbs softmax method, it cannot be allowed to become too large, since its exponential can cause computer overflow. In practice, one keeps these values artificially bounded by projection onto the interval $[-\bar{\mathsf{H}}, \bar{\mathsf{H}}]$ where $\bar{\mathsf{H}} > 0$. The concept of projection is explained below Eq. (8.5). The value of $\bar{\mathsf{H}}$ should be greatest possible value such that the computer does not overflow when it attempts to compute $e^{\bar{\mathsf{H}}}$. It is recommended that the reader be familiar with RL before reading any further.

3.1. Discounted Reward MDPs

We now present AC algorithm from [18, 178]. The algorithm will use the two timescale framework that was discussed in the context of R-SMART and the SSP-version of Relative Q-Learning. In addition to the $H(,.,)$ terms that dictate the action selection, the algorithm also needs the value function, $J(i)$, for each state $i \in \mathcal{S}$.

Step 1. Initialize all J-values and H-values to 0, i.e., for all l, where $l \in \mathcal{S}$ and $u \in \mathcal{A}(l)$, set $J(l) \leftarrow 0$ and $H(l, u) \leftarrow 0$. Set k, the number of iterations, to 0. The algorithm will be run for $k_{\max}$ iterations, where $k_{\max}$ is chosen to be sufficiently large. Set a computer-permissible large positive value to $\bar{\mathsf{H}}$ (as discussed above). Start system simulation at any arbitrary state.

Step 2. Let the current state be i. Select action a with a probability of $p(i, a)$ defined in Eq. (8.3).

Step 3. (**Critic Update**) Simulate action a. Let the next state be j. Let $r(i, a, j)$ be the immediate reward earned in going to j from i under a. Update $J(i)$ via the following equation using the step-size, α:

$$J(i) \leftarrow (1-\alpha)J(i) + \alpha\left[r(i,a,j) + \lambda J(j)\right]. \tag{8.4}$$

Step 4. (**Actor Update**) Update $H(i,a)$ using a step size, β:

$$H(i,a) \leftarrow H(i,a) + \beta \left[r(i,a,j) + \lambda J(j) - J(i) \right]. \qquad (8.5)$$

If $H(i,a) > \bar{\mathsf{H}}$, set $H(i,a) \leftarrow \bar{\mathsf{H}}$. If $H(i,a) < -\bar{\mathsf{H}}$, set $H(i,a) \leftarrow -\bar{\mathsf{H}}$.

Step 5. If $k < k_{\max}$, increment k by 1, set $i \leftarrow j$, and then go to Step 2. Otherwise, go to Step 6.

Step 6. For each $l \in \mathcal{S}$, select $d(l) \in \arg\max_{b \in \mathcal{A}(l)} H(l,b)$. The policy (solution) generated by the algorithm is $\hat{d}$. Stop.

In the above, the step sizes, α and β, should be updated in a style such that they satisfy the two time scale condition, i.e.,

$\lim_{k \to \infty} \frac{\beta^k}{\alpha^k} = 0$; note that we suppressed superscript k above for clarity's sake.

An example of step size rules that satisfy the above condition is:

$$\alpha^k = \frac{log(k)}{k}; \quad \beta^k = \frac{A}{B+k}.$$

Note that β converges to 0 faster than α, and hence the time scale that uses β is called the slower time scale while the time scale that uses α is called the faster time scale. Since β converges to 0 faster, it is as if the faster time scale sees the slower time scale as moving very slowly, i.e., as if the values on the slower time scale are fixed.

3.2. Average Reward MDPs

We now discuss how the actor critic may be used to solve the average reward MDP. Remember that Relative Q-Learning (average reward) was an extension of Q-Learning (discounted reward), and the extension needed a distinguished state, i^*, to be set at the start. In an analogous manner, the average reward actor critic for the MDP follows from its discounted reward counterpart. The differences in the steps are described next.

Step 1: Choose any state from the $\mathcal{S}$ to be the distinguished state i^*.

Step 3: Use the following equation to update $J(i)$:

$$J(i) \leftarrow (1-\alpha)J(i) + \alpha \left[r(i,a,j) - J(i^*) + J(j) \right].$$

Step 4: Update $H(i,a)$ using the following equation:

$$H(i,a) \leftarrow H(i,a) + \beta \left[r(i,a,j) + J(j) - J(i) - J(i^*) \right].$$

If $H(i,a) > \bar{\mathsf{H}}$, set $H(i,a) \leftarrow \bar{\mathsf{H}}$. If $H(i,a) < -\bar{\mathsf{H}}$, set $H(i,a) \leftarrow -\bar{\mathsf{H}}$.

Convergence results assure us that the value of $J(i^*)$ should converge at the end to the vicinity of ρ^*, the optimal average reward.

3.3. Average Reward SMDPs

We now present an extension of the discounted reward algorithm that for the average reward SMDP under Assumption 7.1 of Chap. 7. We call this algorithm S-MACH (Semi-Markov Actor Critic Heuristic). Consider the following equation: For all $i \in \mathcal{S}$, if ρ^* denotes the optimal average reward of the SMDP:

$$J(i) = \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) - \rho \bar{t}(i,a) + \eta \sum_{j \in \mathcal{S}} p(i,a,j) J(j) \right], \text{ where } \rho = \rho^*. \tag{8.6}$$

Note that in the discounted reward Bellman equation for MDPs, if we replace (i) the immediate reward, $r(i,a,j)$, by $r(i,a,j) - \rho^* \bar{t}(i,a,j)$ and (ii) λ by η, we obtain the above equation. Under Assumption 7.1 from Chap. 7, it can be shown that the unique solution of the above equation will yield an optimal solution to the average reward SMDP when $\eta \in (\bar{\eta}, 1)$. Hence, our next algorithm (from [183]) seeks to solve the above equation in order to solve the SMDP. To this end, the algorithm will use three time scales: $J(.)$ and $H(.,.)$ terms will be updated on the fastest and medium time scales, respectively, while ρ will be updated on the slowest time scale. It is shown in [183] that under suitable conditions, ρ will converge to ρ^*, and we will obtain the optimal solution in the limit. The updates on the faster time scales can be obtained directly from those of the AC presented above for discounted reward MDPs. The algorithm differs from the discounted reward AC in the following ways:

Step 1: Set $\rho = 0$. Also, set both TR and TT, the total reward and the total time spent in the simulator, respectively, to 0. Select a positive value for η that is large enough but less that 1.

Step 3: If $t(i,a,j)$ denotes the random transition time, update $J(i)$ via:

$$J(i) \leftarrow (1-\alpha)J(i) + \alpha \left[r(i,a,j) - \rho t(i,a,j) + \eta J(j) \right].$$

Step 4: Update $H(i,a)$ using the following equation:

$$H(i,a) \leftarrow H(i,a) + \beta \left[r(i,a,j) - \rho t(i,a,j) + \eta J(j) - J(i) \right].$$

If $H(i,a) > \bar{\mathsf{H}}$, set $H(i,a) \leftarrow \bar{\mathsf{H}}$. If $H(i,a) < -\bar{\mathsf{H}}$, set $H(i,a) \leftarrow -\bar{\mathsf{H}}$. Then update ρ, TR and TT as follows:

$$TR \leftarrow TR + r(i,a,j);\ TT \leftarrow TT + t(i,a,j); \rho \leftarrow (1-\bar{\gamma})\rho + \bar{\gamma}R/T;$$

where $\bar{\gamma}$ is the step size on the slowest time scale. The step sizes must satisfy the following two rules: $\lim_{k\to\infty} \frac{\bar{\gamma}^k}{\beta^k} = 0$ and $\lim_{k\to\infty} \frac{\beta^k}{\alpha^k} = 0$. Selecting a suitable value for η may need experimentation (see [183]).

In conclusion, we would like to note that ACs have a noteworthy drawback: they need to compute the exponential of a term that can get unbounded. Large powers for the exponential lead to unpleasant computer overflows and numerical instability. Also, the forcible bounding of the $H(.,.)$ terms leads to only ϵ-convergence [178]; an unfortunate consequence of this is that one may often obtain sub-optimal policies in practice.

4. Concluding Remarks

This short chapter was meant to introduce you to solving MDPs/SMDPs via stochastic search, i.e., stochastic policies in which the action selection probabilities (or their surrogates) are directly updated. Our discussion was limited to some of the earliest advances, i.e., learning automata and actor critics. We discuss some of the more recent developments in the bibliographic remarks.

Bibliographic Remarks. MCAT is due to Wheeler and Narendra [319]. Narendra and Thathachar [215] discuss these topics in considerable detail. Several other references are provided in [215]. Our discussion in this chapter, especially that in the context of SMDPs, follows Gosavi et al. [124]. ACs were studied in [328, 18, 316]. The two time scale framework for the actor critic was introduced in [178] and ϵ-convergence was established. A number of other algorithms that bypass the inconvenient exponential term in the action selection have been proposed in [178]. The SMDP algorithm for average reward is from [183]. Other algorithms that use action-selection probabilities include policy gradients [21, 278], simultaneous perturbation [241], and MRAS [62]. See also [327, 167, 168, 225, 217].

Chapter 9

CONVERGENCE: BACKGROUND MATERIAL

1. Chapter Overview

This chapter introduces some fundamental mathematical notions that will be useful in understanding the analysis presented in the subsequent chapters. The aim of this chapter is to introduce elements of the mathematical framework needed for analyzing the convergence of algorithms discussed in this book. Much of the material presented in this chapter is related to mathematical analysis, and hence a reader with a good grasp of mathematical analysis may skip this chapter. To follow Chap. 10, the reader should read all material up to and including Theorem 9.2 in this chapter. All the ideas developed in this chapter will be needed in Chap. 11.

So far in the book we have restricted ourselves to an *intuitive* understanding of why algorithms generate optimal solutions. In this chapter, our aim is to make a transition from the nebulous world of intuition to a more solid mathematical world. We have made every attempt to make this transition as gentle as possible. Apart from the obvious fact that an algorithm's usefulness is doubted unless mathematical arguments show it, there are at least two other reasons for studying mathematical arguments related to an algorithm's ability to generate optimal solutions: (i) mathematical analysis leads to the identification of conditions under which optimal solutions can be generated and (ii) mathematical analysis provides insights into the working mechanism of the algorithm. The reader not interested in everything in this

DOI 10.1007/978-1-4899-7491-4_9

chapter is advised to read ahead, and then come back to this chapter as and when the proof of a result needs to be understood. We will begin this chapter with a discussion on vectors and vector spaces.

2. Vectors and Vector Spaces

We assume that you have a clear idea of what is meant by a **vector**. We will now attempt to tie this concept to the dependable framework of vector **spaces**. We will begin with simple concepts that most readers are familiar with.

A set is a collection of objects; we refer to these objects as elements. A set may contain either a finite or an infinite number of elements. We will use the calligraphic letter (e.g., $\mathcal{A}, \Re$) to denote a set. A finite set can be described explicitly i.e., via a definition of each element. Consider for example:

$$\mathcal{A} = \{1, 4, 1969\}.$$

Here $\mathcal{A}$ denotes a finite set because it contains a finite number of elements; note each of the elements, $1, 4$, and 1969, is clearly defined in this definition of the set.

An infinite set is often described using a *conditional* notation which can take one of the two equivalent forms:

$$\mathcal{B} = \{x : 1 \le x \le 2\} \text{ or } \mathcal{B} = \{x | 1 \le x \le 2\}.$$

In this notation, whatever follows ":" or "|" is the condition; the implication here is that $\mathcal{B}$ is an infinite set that contains all real numbers between 1 and 2, including 1 and 2. In other words, $\mathcal{B} = [1, 2]$ is a *closed* interval.

The set $\Re$ is the set of real numbers—geometrically, it is the real number line. Algebraically, the idea lifts easily to n dimensions. We will see how via the following example.

Example 1. A set $\mathcal{X}$ which is defined as: $\mathcal{X} = \{1, 2, 7\}$. There are three members in this set and each member is actually drawn from the real line—$\Re$. Thus $\mathcal{X}$ is a subset of the set $\Re$. Now consider the next example.

Example 2. A set $\mathcal{Y}$ which is defined as:

$$\mathcal{Y} = \{(1.8, 3), (9.5, 1969), (7.2, 777)\}.$$

Each member of this set is actually a pair of two numbers. If each number in the pair is drawn from the real line, each pair is said to be a member of $\Re^2$. The pair is also called a 2-tuple. Thus each member of the set $\mathcal{Y}$ is a two-tuple. Now consider the following example.

Example 3. Each member of a set $\mathcal{Z}$ assumes the following form:

$$(x(1), x(2), \ldots, x(n)).$$

Then each of these members is said to belong to the set $\Re^n$. Each member in this case is referred to as an n-tuple. An n-tuple has n elements.

The n-tuple is said to be an n-dimensional **vector** if the n-tuple satisfies certain properties. We use the notation $\rightarrow$ above a letter to denote a vector and the notation $\wedge$ to denote the n-tuple. For instance, $\vec{x}$ will denote a vector, but $\hat{y}$ will denote an n-tuple that may or may not be a vector.

The properties that an n-tuple should possess to be called a vector are related to its addition and scalar multiplication. We describe these properties, next.

Addition Property. The addition property requires that the sum of the two n-tuples $\hat{x} = (x_1, x_2, \ldots, x_n)$ and $\hat{y} = (y_1, y_2, \ldots, y_n)$ be defined by

$$\hat{x} + \hat{y} = (x_1 + y_1, x_2 + y_2, \ldots, x_n + y_n).$$

Scalar Multiplication Property. The scalar multiplication property implies that a real number c times the n-tuple $\hat{x}$—denoted by $c\hat{x}$—be defined by

$$c\hat{x} = (cx_1, cx_2, \ldots, cx_n).$$

These definitions qualify the n-tuples to be called **vectors**, and many other properties follow from these two key properties. (See any standard text on linear algebra for more.)

Let us show some examples to demonstrate these properties. The sum of $\hat{a} = (1, 2)$ and $\hat{b} = (4, -7)$ has to be:

$$((1 + 4), (2 - 7)) = (5, -5),$$

if $\hat{a}$ and $\hat{b}$ are vectors. So also the multiplication of $\hat{a}$ by a scalar such as 10 should be given by:

$$10(1, 2) = (10, 20).$$

When endowed with these two key properties (addition and scalar multiplication), we refer to the set $\Re^n$ as a **vector space**. As mentioned above, the framework of vector spaces will be very useful in studying the convergence properties of the algorithms that we have seen in this book.

Some examples of vector spaces. $\Re^2$, $\Re^{132}$, and $\Re^{1969}$.

The next section deals with the important notion of vector norms.

3. Norms

A **norm**, in a given vector space V, is a scalar quantity associated with a given vector in that space. To give a rough geometric interpretation, it denotes the **length** of a vector. There are many ways to define a norm, and we will discuss some standard norms and their properties.

The so-called **max norm**, which is also called **infinity norm** or **sup norm**, is defined as follows.

$$||\vec{x}||_\infty = \max_i |x(i)|.$$

In the above definition, $||\vec{x}||_\infty$ denotes the max norm of the vector $\vec{x}$, and $x(i)$ denotes the ith element of the vector $\vec{x}$. The following example will illustrate the idea.

Example. $\vec{a} = (12, -13, 9)$ and $\vec{b} = (10, 12, 14)$ are two vectors in $\Re^3$. Their max norms are:

$$||\vec{a}||_\infty = \max\{|12|, |-13|, |9|\} = 13 \text{ and}$$

$$||\vec{b}||_\infty = \max\{|10|, |12|, |14|\} = 14.$$

There are other ways to define a norm. The Euclidean norm, for instance, is defined as follows:

$$||\vec{x}||_2 = \sqrt{\sum_i (x(i))^2}.$$

It denotes the Euclidean length of a vector. The Euclidean norm of vectors $\vec{a}$ and $\vec{b}$ are hence:

$$||\vec{a}||_2 = \sqrt{(12)^2 + (-13)^2 + (9)^2} = 19.84943 \text{ and}$$

$$||\vec{b}||_2 = \sqrt{(10)^2 + (12)^2 + (14)^2} = 20.9761.$$

Some important properties of norms are discussed next.

3.1. Properties of Norms

A scalar must satisfy the following properties in order to be called a norm. The notation $||\vec{x}||$ is a norm of a vector $\vec{x}$ belonging to the vector space V if

1. $||\vec{x}|| \geq 0$ for all $\vec{x}$ in V, where $||\vec{x}|| = 0$ if and only if $\vec{x} = \vec{0}$.

2. $||a\vec{x}|| = |a|\ ||\vec{x}||$ for all $\vec{x}$ in V and all a.

3. $||\vec{x} + \vec{y}|| \leq ||\vec{x}|| + ||\vec{y}||$ for all $\vec{x}, \vec{y}$ in V.

The last property is called the triangle inequality . It is not hard to show that all the three conditions are true for both max norms and Euclidean norms. It is easy to verify the triangle inequality in $\Re$ with the norm of x (or y) being the absolute value of x (or y).

4. Normed Vector Spaces

A vector space equipped with a norm is called a **normed vector space.** Study the following examples.

Example 1. The vector space defined by the set $\Re^2$ along with the Euclidean norm. The norm for this space is:

$$||\vec{x}||_2 = \sqrt{\sum_{i=1}^{2}[x(i)]^2 \text{ if the vector } \vec{x} \text{ is in } \Re^2}.$$

Example 2. The vector space defined by the set $\Re^{69}$ equipped with the max norm. The norm for this space is:

$$||\vec{x}||_\infty = \max_{1\leq i\leq 69} |x(i)| \text{ where the vector } \vec{x} \text{ is in } \Re^{69}.$$

We will return to the topic of vector spaces after discussing sequences.

5. Functions and Mappings

It is assumed that you are familiar with the concept of a function from high school calculus courses. In this section, we will present some definitions and some examples of functions.

5.1. Domain and Range

A simple example of a function is: $y = x + 5$, where x can take on any real value. We often denote this as: $f(x) = x + 5$. The rule $f(x) = x + 5$ leaves us in no doubt about what the value of y will be when the value of x is known. A function is thus a **rule**. You are familiar with this idea. Now, let us interpret a function as a set of pairs; one value in the pair will be any legal value of x, and the other value will be the corresponding value of y. For instance, the function considered above can be defined as a set of (x, y) pairs; some examples of these pairs are $(1, 6), (1.2, 6.2)$ etc.

Notice that the values of x actually come from $\Re$, the set of real numbers. And the values of y, as a result of how we defined our function, also come from the same set $\Re$. The set from which the values of x come is called the **domain** of the function, and the set from which the values of y come is called the **range** of the function. The domain and range of a function can be denoted using the following notation.

$$f : \mathcal{A} \rightarrow \mathcal{B},$$

where $\mathcal{A}$ is the domain and $\mathcal{B}$ is the range. This is read as: "a function f from $\mathcal{A}$ to $\mathcal{B}$."

The example given above is the simplest form of a function. Let us consider some more complicated functions. Consider the following function.

$$y = 4 + 5x_1 + 3x_2 \tag{9.1}$$

in which each of x_1 and x_2 takes on values from the set $\Re$. Now, a general notation to express this function is $y = f(x_1, x_2)$. This function (given in Eq. (9.1)) clearly picks up a **vector** such as $(1, 2)$ and assigns a value of 15 to y. Thus, the function (9.1) can be represented as a set of ordered triples (x_1, x_2, y)—examples of which are: $(1, 2, 15)$ and $(0.1, 1.2, 8.1)$, and so on. This makes it possible to view this function as an operation, whose input is a vector of the form (x_1, x_2) and whose output is a scalar. In other words, the domain of the function is $\Re^2$ and its range is $\Re$.

Functions that deal with vectors are also called **mappings** or **maps** or **transformations**. It is not hard to see that we can define a function from $\Re^2$ to $\Re^2$. An example of such a function is defined by the following **two** equations:

$$\begin{aligned} x'_1 &= 4x_1 + 5x_2 + 9, \text{ and} \\ x'_2 &= 5x_1 + 7x_2 + 7. \end{aligned}$$

In the context of the above function, consider the vector with $x_1 = 1$ and $x_2 = 3$. It is a member of the set $\Re^2$. When the function or transformation is applied on it (i.e., used as an input in the right hand sides of the equations above), what we get is another vector belonging to the set $\Re^2$. In particular, we get the vector $x'_1 = 28$, and $x'_2 = 33$. One of the reasons why a vector function is also called a transformation is: a vector function generates a new vector from the one that is supplied to it.

The function shown above is actually defined by two linear equations. It is convenient to represent such a function with vector notation. The following illustrates the idea.

$$\vec{x'} = \mathbf{A}\vec{x} + \mathbf{B}. \tag{9.2}$$

Here

$$\vec{x} = (x_1, x_2)^T \text{ and } \vec{x'} = (x'_1, x'_2)^T$$

are the two-dimensional vectors in question and $\mathbf{A}$ and $\mathbf{B}$ are the matrices. ($\vec{x}^T$ denotes the transpose of the vector $\vec{x}$.) For the example under consideration, the matrices are:

$$\mathbf{A} = \begin{bmatrix} 4 & 5 \\ 5 & 7 \end{bmatrix}, \text{ and } \mathbf{B} = \begin{bmatrix} 9 \\ 7 \end{bmatrix}.$$

In general, functions or transformations can be defined from $\Re^{n_1}$ to $\Re^{n_2}$, where n_1 and n_2 are positive integers that may or may not be equal.

5.2. The Notation for Transformations

In general, we will use the following compact notation for transformations on vectors:

$$F\vec{a} \equiv F(\vec{a}).$$

Here F denotes a transformation (or a mapping or a function) that transforms the vector $\vec{a}$ while $F\vec{a}$ denotes the **transformed vector**. This implies that $F\vec{a}$ is a vector that may be different from $\vec{a}$—the vector that was transformed by F.

We will be using operators of the form F^k frequently in the remainder of this book. The meaning of this operator needs to be explained. Let us examine some examples now.

F^1 will mean the same thing as F. But $F^2(\vec{a})$, in words, is the vector that is obtained after applying the transformation F to the vector $F(\vec{a})$. In other words, $F^2(\vec{a})$ denotes the vector obtained after applying the transformation F **two times** on the vector $\vec{a}$. Mathematically:

$$F^2(\vec{a}) \equiv F\left(F(\vec{a})\right).$$

In general, F^k means the following:

$$F^k\vec{a} \equiv F\left(F^{k-1}(\vec{a})\right).$$

We will, next, discuss the important principle of mathematical induction.

6. Mathematical Induction

The principle of mathematical induction will be used on many occasions in this book. As such, it is important that you understand it clearly. Before we present it, let us define $\mathcal{J}$ to be the set of positive integers, i.e., $\mathcal{J} = \{1, 2, 3, 4, \ldots\}$. The basis for mathematical induction is the well-ordering principle that we state below without proof.

Well-ordering principle. Every non-empty subset of $\mathcal{J}$ has a member that can be called its smallest member.

To understand this principle, consider some non-empty subsets of $\mathcal{J}$.

$$\mathcal{A} = \{1, 3, 5, 7, \ldots\}, \mathcal{B} = \{3, 4, 5, 6\}, \text{ and } \mathcal{C} = \{34, 1969, 4, 11\}.$$

Clearly, each of these sets is a subset of $\mathcal{J}$ and has a smallest member. The smallest members of $\mathcal{A}, \mathcal{B}$, and $\mathcal{C}$ are respectively $1, 3$, and 4.

THEOREM 9.1 **(Principle of Mathematical Induction)** *If $R(n)$ is a statement containing the integer n such that*

(i) $R(1)$ is true and

(ii) After **assuming** *that $R(k)$ is true, $R(k+1)$ can be shown to be true for every k in $\mathcal{J}$,*

then $R(n)$ is true for all n in $\mathcal{J}$.

This theorem implies that the relation holds for $R(2)$ from the fact that $R(1)$ is true, and from the truth of $R(2)$ one can show the same for $R(3)$. In this way, it is true for all n in $\mathcal{J}$. All these are intuitive arguments. We now present a rigorous proof.

Proof We will use the so-called contradiction logic. It is likely that you have used this logic in some of your first experiences in mathematics in solving geometry riders. In this kind of a proof, we will begin by **assuming** that the result we are trying to prove is **false**. Using the falseness of the result, we will go on to show that something else that we assumed to be true cannot be true. Hence our hypothesis—that the "result is false"—cannot be true. As such, the result is true.

Let us assume that:

(i) $R(1)$ is true, and

(ii) If $R(k)$ is true, $R(k+1)$ is true for every k in $\mathcal{J}$.

In addition, let us assume that

(iii) $R(n)$ is **not** true for **some** values of n.

Let us define a set $\mathcal{S}$ to be a set that contains **all** values of n for which $R(n)$ is not true. Then from assumption (iii), $\mathcal{S}$ is a non-empty subset of $\mathcal{J}$. From the well-ordering principle, one has that $\mathcal{S}$ must have an element that can be described as its smallest element. Let us say that the value of n for this smallest element is p. Now $R(1)$ is true and so p must be greater than 1. This implies that $(p-1) > 0$, which means that $(p-1)$ belongs to $\mathcal{J}$.

Since p is the smallest element of $\mathcal{S}$, the relation $R(n)$ must be true for $(p-1)$ since $(p-1)$ does not belong to $\mathcal{S}$—the set of all the elements for which $R(n)$ is false. Thus we have that $R(p-1)$ is true but $R(p)$ is not. Now this cannot be right since in (ii) we had assumed that if $R(k)$ is true then $R(k+1)$ must be true. Thus setting $k = p-1$, we have a contradiction, and hence our initial hypothesis must be wrong. In other words, the result must be true for **all** values of n in $\mathcal{J}$. ▮

Some general remarks are in order here. We have started our journey towards establishing that the algorithms discussed in previous chapters, indeed, produce optimal or near-optimal solutions. To establish these facts, we will have to come up with proofs written in the style used above.

The proof presented above is our first "real" proof in this book. Any "mathematically rigorous" proof has to be worked out to its last detail, and if any detail is missing, the proof is not acceptable.

Let us see, next, how the theorem of mathematical induction is used in practice. Consider the following problem.

Prove that

$$1 + 3 + 5 + \cdots + (2p-1) = (p)^2.$$

(We will use LHS to denote the left hand side of the relation and RHS the right hand side. Notice that it is easy to use the formulation for the arithmetic progression series to prove the above, but our intent here is to demonstrate how induction proofs work.)

Now, when $p = 1$, $LHS = 1$ and the $RHS = 1^2 = 1$, and thus the relation (the equation in this case) is true when $p = 1$.

Next let us assume that it is true when $p = k$, and hence

$$1 + 3 + 5 + \cdots + (2k-1) = k^2.$$

Now when $p = (k+1)$ we have that:

$$\begin{aligned} LHS &= 1 + 3 + 5 + \cdots + (2k-1) + (2(k+1)-1) \\ &= k^2 + 2(k+1) - 1 \\ &= k^2 + 2k + 2 - 1 \\ &= (k+1)^2 \\ &= RHS \text{ when } p = k+1. \end{aligned}$$

Thus we have proved that the relation is true when $n = k+1$ **if** the relation is true when $p = k$. Then, using the theorem of mathematical induction, one has that the relation is true.

As you have probably realized that to prove a relation using induction, one must **guess** what the relation should be. But then the obvious question that arises is: how should one guess? For instance, in the example above, how may one guess that the RHS should be p^2? We will address this question in a moment. But keep in mind that once we have a guess, we can use induction to prove that the relation is true (that is if the guess is right in the first place).

We now show some *technology* that may help in making good guesses. Needless to say, guessing randomly rarely works. Often, knowing how a relationship behaves for **given** values of p comes in handy and from that it may be possible to recognize a general pattern. Once a pattern is identified, one can generalize to obtain a plausible expression. All this is part of **rough work** that we omit from the final proof. We will illustrate these ideas using the result discussed above.

The RHS denotes the sum of the first p terms, which we will denote by S^p. So let us see what values S^p takes on for small values of p. The values are:

$$S^1 = 1, S^2 = 4, S^3 = 9, S^4 = 16, S^5 = 25, \text{ and so on.}$$

You have probably noticed that there is a relationship between

$$1, 4, 9, 16, 25, \ldots.$$

Yes, you've guessed right! They are the squares of

$$1, 2, 3, 4, 5 \ldots.$$

And hence a logical guess for S^p is $S^p = p^2$. We can then, as we have done above, use induction to show that it is indeed true. The example that we considered above is very simple but the idea was to show you that mathematical results are never the consequence of "divine inspiration" or "superior intuition" [95] but invariably the consequence of such rough work.

7. Sequences

It is assumed that you are familiar with the notion of a sequence. A familiar example of a sequence is:

$$a, ar, a(r)^2, a(r)^3, \ldots$$

We associate this with the geometric progression in which a is the first term in the sequence and r is the common ratio.

A sequence, by its definition, has an infinite number of terms. We will be dealing with sequences heavily in this book. Our notation for a sequence is:

$$\{x^p\}_{p=1}^{\infty}.$$

Here x^p denotes the pth term of the sequence. For the geometric sequence shown above:

$$x^p = (a)^{p-1}.$$

A sequence can be viewed as a function whose *domain* is the set of positive integers $(1, 2, \ldots)$ and the *range* is the set that includes all possible values that the terms of the sequence can take on.

We are often interested in finding the value of the **sum** of the first m terms of a sequence. Let us consider the geometric sequence given above. The sum of the first m terms of this sequence, it can proved, is given by:

$$S^m = \frac{a(1-(r)^m)}{(1-r)}. \tag{9.3}$$

The sum itself forms the sequence

$$\{S^1, S^2, S^3, \ldots, \}$$

We can denote this sequence by $\{S^m\}_{m=1}^{\infty}$. We will prove (9.3) using mathematical induction.

Proof Since the first term is a, S^1 should be a. Plugging in 1 for m in the (9.3), we can show that. So clearly the relation is true for $m = 1$. Now let us assume that it is true for $m = k$; that is,

$$S^k = \frac{a(1-(r)^k)}{(1-r)}.$$

The $(k+1)$th term is (from the way the sequence is defined) ar^k. Then the sum of the first $(k+1)$ terms is the sum of the first k terms and the $(k+1)$th term. Thus:

$$\begin{aligned} S^{k+1} &= \frac{a(1-(r)^k)}{(1-r)} + a(r)^k \\ &= \frac{a(1-(r)^k) + a(r)^k(1-r)}{1-r} \\ &= \frac{a - a(r)^k + a(r)^k - a(r)^{k+1}}{1-r} = \frac{a(1-(r)^{k+1})}{1-r}. \end{aligned}$$

This proves that the relation is true when $m = k+1$. Then, as a result of the theorem of mathematical induction, the relation given in (9.3) is true. ∎

We will next discuss sequences that share an interesting property—namely that of **convergence.**

7.1. Convergent Sequences

Let us illustrate the idea of convergence of a sequence using familiar ideas. As you probably know, the sum of the GP series **converges** if the common ratio r lies in the interval $[0, 1)$. Convergence of a sequence,$\{x^p\}_{p=1}^{\infty}$, usually, means that as p starts becoming large, the terms of the sequence start approaching a **finite quantity**. The reason why the GP series converges is that the limit of the sum when p tends to infinity is a finite quantity. This is shown next.

$$\lim_{p\to\infty} S^p = \lim_{p\to\infty} a(1-(r)^p)/(1-r) = a(1-0)/(1-r) = a/(1-r). \quad (9.4)$$

The above uses the fact that

$$\lim_{p\to\infty} (r)^p = 0 \text{ when } 0 \leq r < 1.$$

Another simple example of a convergent sequence is one whose pth term is defined by $1/p$. The sequence can be written as:

$$\frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \ldots.$$

It is **intuitively** obvious that as p increases, the term $1/p$ keeps getting smaller. Also, we know from our knowledge of high school calculus that:

$$\lim_{p\to\infty} \frac{1}{p} = 0.$$

From this stage, we will gradually introduce rigor into our discussion on sequences. In particular, we need to define some concepts such as convergence and Cauchy sequences. We will use the abbreviation **iff** to mean "if and only if." The implication of 'iff' needs to be understood. When we say that the condition A holds iff B is true, the following is implied: A is true if B is true **and** B is true if A is true.

DEFINITION 9.1 **A Convergent Sequence:** *A sequence $\{a^p\}_{p=1}^{\infty}$ is said to converge to a real number A iff for any $\epsilon > 0$, there exists a positive integer N such that for all $p \geq N$, we have that*

$$|a^p - A| < \epsilon.$$

Let us explain this definition with an example. Consider the sequence whose pth term is given by $\frac{1}{p}$. It is not hard to see why this sequence converges to 0. If you supply us with a value for ϵ, and if the sequence indeed converges to 0, then we should be able to come up with a value for N such that for any integer, p, which is greater than N, $|a^p - 0|$ must be less than ϵ. For instance, let us assume that $\epsilon = 0.01$. Then,

$$|a^p - 0| = |\frac{1}{p} - 0| = \frac{1}{p} < \epsilon.$$

This implies that

$$p > \frac{1}{\epsilon} = \frac{1}{0.01} = 100.$$

This means that for $p \geq 101$, the condition $|a^p - 0| < \epsilon$ must be satisfied. It is not hard to see from this discussion that the value of N **depends** on the value of ϵ, but if the sequence converges, then for any value of ϵ, one can come up with a suitable finite value for N.

We next define some special properties of sequences.

7.2. Increasing and Decreasing Sequences

Definition 9.2 *A sequence $\{a^p\}_{p=1}^{\infty}$ is said to be decreasing, if for all p,*

$$a^{p+1} \leq a^p.$$

For the same reason, it is called an increasing sequence, if for all p,

$$a^{p+1} \geq a^p.$$

7.3. Boundedness

We next define the concepts of "bounded above" and "bounded below," in the context of a sequence. A sequence $\{a^p\}_{p=1}^{\infty}$ is said to be **bounded below**, if there exists a finite value L such that:

$$a^p \geq L$$

for all values of p. L is then called a **lower bound** for the sequence.

Similarly, a sequence is said to be **bounded above**, if there exists a finite value U such that:

$$a^p \leq U$$

for all values of p. U is then called an **upper bound** for the sequence.

The sequence:

$$\{1, 2, 3, \ldots\}$$

is bounded below, but not above. Notice that 1 is a *lower* bound for this sequence and so are 0 and -1 and -1.5. But 1 is the highest lower bound (often called the infimum).

In the sequence:

$$\{1, 1/2, 1/3, \ldots\},$$

1 is an upper bound and so are 2 and 3 etc. Here 1 is the lowest of the upper bounds (also called the supremum).

A sequence that is bounded both above and below is said to be a **bounded** sequence.

We will next examine a useful result related to decreasing (increasing) sequences that are bounded below (above). The result states that a decreasing (increasing) sequence that is bounded below (above) converges.

Intuitively, it should be clear that a decreasing sequence, that is, a sequence in which each term is less than or equal to the previous term, should converge to a finite value because the values of the terms cannot go below a finite value M. So the terms keep decreasing and once they reach a point below which they cannot go, they stop decreasing; so the sequence should converge. Now, let us prove this idea using precise mathematics. One should remember that to prove convergence, one must show that the sequence satisfies Definition 9.1.

THEOREM 9.2 *A decreasing (increasing) sequence converges if it is bounded below (above).*

Proof We will work out the proof for the decreasing sequence case. For the increasing sequence case, the proof can be worked out in a similar fashion. Let us denote the sequence by $\{a^p\}_{p=1}^{\infty}$. Let L be the highest of the lower bounds on the sequence. Then, for any p, since L is a lower bound,

$$a^p \geq L. \tag{9.5}$$

Choose a strictly positive value for the variable ϵ. Then $\epsilon > 0$. Then $L + \epsilon$, which is greater than L, is **not** a lower bound. (Note that L is the **highest** lower bound.) Then, it follows that there exists an N, such that

$$a^N < L + \epsilon. \tag{9.6}$$

Then, for $p \geq N$, since it is a decreasing sequence,

$$a^p \leq a^N.$$

Combining the above, with Inequations (9.5) and (9.6), we have that for $p \geq N$:

$$L \leq a^p \leq a^N < L + \epsilon.$$

The above implies that for $p \geq N$:

$$L \leq a^p < L + \epsilon.$$

Using the above and the fact that $L - \epsilon < L$, we have that for $p \geq N$:

$$L - \epsilon < a^p < L + \epsilon.$$

The above means that for $p \geq N$: $|a^p - L| < \epsilon$ where $\epsilon > 0$. From Definition 9.1, this means that $\{a^p\}_{p=1}^{\infty}$ is convergent. ▌

The definition of convergence (Definition 9.1) requires the knowledge of the limit (A) of the sequence. Next, we discuss a concept that will lead us to a method that can determine whether a sequence is convergent without attempting to identify its limit.

DEFINITION 9.3 **A Cauchy sequence:** *A sequence $\{a^p\}_{p=1}^{\infty}$ is called a Cauchy sequence iff for each $\epsilon > 0$, there exists a positive integer N such that if one chooses* **any** *m greater than or equal to N and* **any** *k also greater than or equal to N, that is, $k, m \geq N$, then*

$$|a^k - a^m| < \epsilon.$$

This is an important definition that needs to be understood clearly. What it says is that a sequence is Cauchy if for any given value of ϵ, there exists a finite integer N such that the absolute value of the difference between any two terms of the sequence beyond and including the Nth term is less than ϵ. This means that beyond and including that point N in the sequence, **any** two terms are ϵ-close, i.e., the absolute value of their difference is less than ϵ. We illustrate this idea with an example.

Let us assume that $\{1/p\}_{p=1}^{\infty}$ is a Cauchy sequence. (Every convergent sequence is Cauchy, and so this assumption can be justified.) Select $\epsilon = 0.0125$, pick *any* two terms that are beyond the 79th term ($N = 80 = 1/0.0125$ for this situation, and clearly depends on the value of ϵ), and calculate the absolute value of their difference. The value will be less than 0.025. The reason is at the 80th term, the value will be equal to $1/80 = 0.0125$, and hence the difference is bound to be less than 0.0125. Note that:

$$\frac{1}{80} - \frac{1}{p} < 0.0125 = \epsilon \text{ for a strictly positive value of } p.$$

In summary, the value of N will depend on ϵ, but the definition says that for a Cauchy sequence, one can always find a value for N beyond which terms are ϵ-close.

It can be proved that every convergent sequence is Cauchy. Although this is an important result, its converse is even more important. Hence, we omit its proof and turn our attention to its converse.

The converse states that every Cauchy sequence is convergent (we present that as Theorem 9.5 on page 298). The converse will prove to be a very useful result as it will provide us with a mechanism to establish the convergence of a sequence. The reason is that by showing that a given sequence is Cauchy, we will be able to establish that the sequence is convergent.

To prove this important result, we need to discuss a few important topics, which are the boundedness of Cauchy sequences, accumulation points, neighborhoods, and the Bolzano-Weierstrass Theorem. The next few paragraphs will be devoted to discussing these topics.

For a bounded sequence, each of its terms is a finite quantity. As a result, the sequence $\{a^p\}_{p=1}^{\infty}$ is bounded if there is a **positive** number M such that $|a^p| \leq M$ for all p. This condition implies that:

$$-M \leq a^p \leq M.$$

THEOREM 9.3 *Every Cauchy sequence is bounded.*

This is equivalent to saying that every term of a Cauchy sequence is a finite quantity. The proof follows.

Proof Assume that $\{a^p\}_{p=1}^{\infty}$ is a Cauchy sequence. Let us set $\epsilon = 1$. By the definition of a Cauchy sequence, there exists an N such that, for all $k, m \geq N$, we have $|a^k - a^m| < 1$. Let $m = N$. Then for $k \geq N$,

$$\begin{aligned} |a^k| &= |a^k - a^N + a^N| \\ &\leq |a^k - a^N| + |a^N| < 1 + |a^N|. \end{aligned}$$

Since N is finite, a^N must be finite. Hence $|a^N|$ must be a positive finite number. Then it follows from the above inequality that $|a^k|$ is less than a positive finite quantity when $k \geq N$. In other words, a^k is bounded for $k \geq N$. Now, since N is finite, all values of a^k from $k = 1, 2, \ldots, N$ are also finite; that is a^k is finite for $k \leq N$. Thus we have shown that a^k is finite for all values of k. The proof is complete. ∎

In the proof above, if you want to express the boundedness condition in the form presented just before the statement of this theorem, define M as

$$M = \max\{a^1, a^2, \ldots, a^{N-1}, a^N, a^N + 1\}.$$

Then $|a^k| \leq M$ for all values of k. Thus $|a^k|$ is bounded for **all** values of k.

Definition 9.4 *A neighborhood of a real number x with a positive radius r is the open interval $(x - r, x + r)$.*

An open interval (a, b) is a set that contains **infinitely many** points from a to b but the points a and b themselves are **not** included in it. Examples are: $(2, 4)$ (neighborhood of 3) and $(1.4, 79.2)$. In the scalar world, a neighborhood of x is an open interval centered at x.

Definition 9.5 *A point x is called the accumulation point of a set $\mathcal{S}$ iff* **every** *neighborhood of x contains infinitely many points of $\mathcal{S}$.*

Although this is a very elementary concept, one must have a very clear idea of what constitutes and what does not constitute an accumulation point of a given set.

Example 1. Let a set $\mathcal{S}$ be defined by the open interval $(1, 2.4)$. Then every point in $\mathcal{S}$ is an accumulation point. Consider any point in the interval—say 2. Whichever neighborhood of 2 is chosen, you will find that one of the following is true:

- The neighborhood is a subset of $\mathcal{S}$ (for example: if the neighborhood is the interval $(1.8, 2.2)$)
- The neighborhood contains a subset of $\mathcal{S}$ (for example: the neighborhood is the interval $(1.2, 2.8)$)
- The set $\mathcal{S}$ is a subset of the neighborhood (for example: the neighborhood is the interval $(0.5, 3.5)$).

In any case, the neighborhood will contain infinitely many points of the set $\mathcal{S}$.

Example 2. We next consider an example of a point that **cannot** be an accumulation point of the interval $(1, 2.4)$. Consider the point 5. Now, the interval $(2, 8)$ is a neighborhood of 5. This neighborhood contains infinitely many points of the interval $(1, 2.4)$. Yet, 5 is not an accumulation point of the interval $(1, 2.4)$, since one can always construct a neighborhood of 5 that does not contain even a single point from the interval $(1, 2.4)$. An example of such a neighborhood is the interval $(4, 6)$.

Notice that the definition of an accumulation point of a set says that **every** neighborhood of the set must contain infinitely many points of the set.

Example 3. Consider the sequence $\{\frac{1}{p}\}_{p=1}^{\infty}$. Recall that the range of a sequence is the set of all the different values that the sequence can assume. The range of this sequence is an infinite set since one can keep increasing the value of p to get positive terms that keep getting smaller. The terms of course approach 0, but we can never find a finite value for p for which $a(p)$ will **equal** 0. Hence 0 is not a point in the range of this sequence. And yet 0 is an accumulation point of the range. This is because any neighborhood of 0 contains infinitely many points of the range of the sequence.

We are now at a point to discuss the famous Bolzano-Weierstrass theorem. We will not present its proof, so as to not distract the reader from our major theme which is the convergence of a Cauchy sequence. The proof can be found in any undergraduate text on mathematical analysis such as Gaughan [95] or Douglass [81]. This theorem will be needed in proving that Cauchy sequences are convergent.

THEOREM 9.4 **(Bolzano-Weierstrass Theorem:)** *Every bounded infinite set of real numbers has at least one accumulation point.*

The theorem describes a very important property of bounded infinite sets of real numbers. It says that a bounded infinite set of real numbers has at least one accumulation point. We will illustrate the statement of the theorem using an example.

Consider the interval $(1, 2)$. It is bounded (by 1 below and 2 above) and has infinitely many points. Hence it must have at least one accumulation point. We have actually discussed above how *any* point in this set is an accumulation point.

A finite set does not have any accumulation point because, by definition, it has a finite number of points, and hence none of its neighborhoods can contain an infinite number of points.

The next result is a key result that will be used in the convergence analysis of reinforcement learning algorithms.

THEOREM 9.5 *A Cauchy sequence is convergent.*

Proof Let $\{a^p\}_{p=1}^{\infty}$ be a Cauchy sequence. The range of the sequence can be an infinite or a finite set. Let us handle the finite case first, as it is easier.

Let $\{s_1, s_2, \ldots, s_r\}$ consisting of r terms denote a finite set representing the range of the sequence $\{a^p\}_{p=1}^{\infty}$. Now if we choose

$$\epsilon = \min\{|s_i - s_j| : i \neq j; i, j = 1, 2, \ldots, r\},$$

then there is a positive integer N such that for any $k, m \geq N$ we have that $|a^k - a^m| < \epsilon$. Now it is the case that $a^k = s_c$ and $a^m = s_d$ for some c and some d belonging to the set $\{1, 2, \ldots, r\}$. Thus:

$$|s_c - s_d| = |a^k - a^m| < \epsilon = \min\{|s_i - s_j| : i \neq j; i, j = 1, 2, \ldots, r\}.$$

From the above, it is clear that the absolute value of the difference between s_c and s_d is strictly less than the minimum of the absolute value of the differences between the terms. As a result, $|a^k - a^m| = 0$, i.e., $a^k = a^m$ for $k, m \geq N$. This implies that from some point (N) onwards, the sequence values are constant. This means that at this point the sequence converges to a finite quantity. Thus the convergence of the Cauchy sequence with a finite range is established.

Next, let us assume that the set we refer to as the range of the sequence is infinite. Let us call the range $\mathcal{S}$. The range of a Cauchy sequence is bounded by Theorem 9.3. Since $\mathcal{S}$ is infinite and bounded, by the Bolzano-Weierstrass theorem (Theorem 9.4), $\mathcal{S}$ must have an accumulation point; let us call it x. We will prove that the sequence converges to x.

Choose an $\epsilon > 0$. Since x is an accumulation point of $\mathcal{S}$, by its definition, the interval $(x - \frac{\epsilon}{2}, x + \frac{\epsilon}{2})$ is a neighborhood of x that contains infinitely many points of $\mathcal{S}$. Now, since $\{a^p\}_{p=1}^{\infty}$ is a Cauchy sequence, there is a positive integer N such that for $k, m \geq N$, $|a^k - a^m| < \epsilon/2$.

Since $(x - \frac{\epsilon}{2}, x + \frac{\epsilon}{2})$ contains infinitely many points of $\mathcal{S}$, and hence **infinitely** many terms of the sequence $\{a^p\}_{p=1}^{\infty}$, there exists a t such that $t \geq N$ and that a^t is a point in the interval $(x - \frac{\epsilon}{2}, x + \frac{\epsilon}{2})$. The last statement implies that:

$$|a^t - x| < \epsilon/2.$$

Now, since $t > N$ and by the selection of N above, we have from the definition of a Cauchy sequence that

$$|a^p - a^t| < \epsilon/2.$$

Then, we have that

$$|a^p - x| \leq |a^p - a^t| + |a^t - x| = \frac{\epsilon}{2} + \frac{\epsilon}{2} = \epsilon.$$

This implies that $\{a^p\}_{p=1}^{\infty}$ converges to x. ∎

7.4. Limit Theorems and Squeeze Theorem

In this section, we provide some important results that will be used in the book. We begin with the algebraic limit theorem, which we state without proof. The reader is referred to any text on analysis (e.g., [81]) for a proof.

THEOREM 9.6 (**Algebraic Limit Theorem**) *Consider two sequences* $\{a^p\}_{p=1}^{\infty}$ *and* $\{b^p\}_{p=1}^{\infty}$ *such that* $\lim_{p\to\infty} a^p = A$ *and* $\lim_{p\to\infty} b^p = B$. *Then,* *(i)* $\lim_{p\to\infty} Ca^p = CA$ *for all* $C \in \Re$*;*

(ii) $\lim_{p\to\infty}(a^p + b^p) = A + B$*;*

(iii) $\lim_{p\to\infty}(a^p b^p) = AB$*;*

(iv) $\lim_{p\to\infty}(a^p/b^p) = A/B$ *provided* $B \neq 0$*.*

We now present an important theorem that will be used on several occasions in Chap. 11.

Theorem 9.7 (**Order Limit Theorem**) *Consider two sequences* $\{a^p\}_{p=1}^{\infty}$ *and* $\{b^p\}_{p=1}^{\infty}$ *such that* $\lim_{p\to\infty} a^p = A$ *and* $\lim_{p\to\infty} b^p = B$*. Then,*

(i) If $a^p \geq 0$ *for all* $p \in \mathcal{J}$*, then* $A \geq 0$*.*

(ii) If $a^p \leq b^p$ *for all* $p \in \mathcal{J}$*, then* $A \leq B$*.*

(iii) If there exists a scalar $C \in \Re$ *for which* $C \leq b^p$ *for all* $p \in \mathcal{J}$*, then* $C \leq B$*. In a similar manner, if* $a^p \leq C$ *for all* $p \in \mathcal{J}$*, then* $A \leq C$*.*

Proof (i) We use contradiction logic. Assume that $A < 0$. From the definition of convergence we have that for any $\epsilon > 0$, there exists an N such that for all $p \geq N$, $|a^p - A| < \epsilon$. If $\epsilon = |A|$, then we have that $|a^p - A| < |A|$, which implies that $a_N < 0$. But the sequence is defined so that $a_N \geq 0$, and hence we have a contradiction. Therefore $A \geq 0$.

(ii) Since $a_p \leq b_p$, we have that $b_p - a_p \geq 0$. Theorem 9.6 implies that the sequence $\{b_p - a_p\}_{p=1}^{\infty}$ converges to $B - A$. Using part (i) of this theorem, we have that $B - A \geq 0$, i.e., $A \leq B$.

(iii) Assume that every element of the sequence $\{a^p\}_{p=1}^{\infty}$ equals C, i.e., $a_p = C$ for all $p \in \mathcal{J}$. Then applying part (ii) of this theorem, we have the result. The other part can be proved similarly. ∎

We now present another important result that will be used later in the book.

Theorem 9.8 (**Squeeze Theorem**) *Consider three sequences* $\{x^p\}_{p=1}^{\infty}$*,* $\{y^p\}_{p=1}^{\infty}$*, and* $\{z^p\}_{p=1}^{\infty}$ *such that* $x^p \leq y^p \leq z^p$ *for all* $p \in \mathcal{J}$*. Now if* $\lim_{p\to\infty} x^p = \lim_{p\to\infty} z^p = A$*, then* $\lim_{p\to\infty} y^p = A$ *as well.*

Proof Applying, Theorem 9.7 for the sequences $\{x^p\}_{p=1}^{\infty}$ and $\{y^p\}_{p=1}^{\infty}$, we have that $A \leq \lim_{p\to\infty} y^p$. Similarly, applying Theorem 9.7 for the sequences $\{y^p\}_{p=1}^{\infty}$ and $\{z^p\}_{p=1}^{\infty}$, we have that $\lim_{p\to\infty} y^p \leq A$. Together the two bounds on $\lim_{p\to\infty} y^p$ imply that $\lim_{p\to\infty} y^p = A$. ∎

8. Sequences in $\Re^n$

Thus far, our discussions have been limited to **scalar** sequences. A scalar sequence is one whose terms are scalar quantities. Scalar sequences are also called sequences in $\Re$. The reason for this is that scalar quantities are members of the set $\Re$.

A sequence in $\Re^n$ is a sequence whose terms are vectors. We will refer to this sequence as a **vector sequence**. For example, consider a sequence $\{\vec{a}^p\}_{p=1}^{\infty}$ whose pth term is defined as:

$$\vec{a}^p = (\frac{1}{p}, p^2 + 1).$$

This sequence, starting at $p = 1$, will take on the following values:

$$\{(1, 2), (\frac{1}{2}, 5), (\frac{1}{3}, 10), \ldots\}.$$

The above is an example of a sequence in $\Re^2$. This concepts nicely extends to any dimension. Each of the individual scalar sequences in such a sequence is called a **coordinate** sequence. Thus, in the example given above, $\{\frac{1}{p}\}_{p=1}^{\infty}$ and $\{p^2 + 1\}_{p=1}^{\infty}$ are the coordinate sequences of $\{\vec{a}^p\}_{p=1}^{\infty}$.

Remark. We will use the notation $a^p(i)$ to denote the pth term of the ith coordinate sequence of the vector sequence $\{\vec{a}^p\}_{p=1}^{\infty}$. For instance, in the example given above,

$$\vec{a}^p = \{a^p(1), a^p(2)\},$$

where

$$a^p(1) = \frac{1}{p} \text{ and } a^p(2) = p^2 + 1.$$

We will next define what is meant by a Cauchy sequence in $\Re^n$.

9. Cauchy Sequences in $\Re^n$

The concept of Cauchiness also extends elegantly from sequences in $\Re$ (where we have seen it) to sequences in $\Re^n$. The Cauchy condition in $\Re^n$ is defined next.

DEFINITION 9.6 *A sequence in $\Re^n$, denoted by $\{\vec{a}^p\}_{p=1}^{\infty}$, is said to be a Cauchy sequence in $\Re^n$, if for any given $\epsilon > 0$, there exists an N such that for any $m, k \geq N$, $||\vec{a}^k - \vec{a}^m|| < \epsilon$, where $||.||$ denotes any norm.*

You've probably noticed that the definition of Cauchiness in $\Re^n$ uses a norm whereas the same in $\Re$ uses the absolute value of the difference between two points. The definition of Cauchiness for $\Re$ is of course a special case of Definition 9.6. We next state a result that relates the Cauchiness of the individual coordinate sequences within a vector sequence to the Cauchiness of the parent (vector) sequence.

LEMMA 9.9 *If a vector sequence $\{\vec{a}^p\}_{p=1}^{\infty}$ in $\Re^n$ is Cauchy, then each coordinate sequence in the vector sequence is a Cauchy sequence in $\Re$. In other words, if the vector sequence is Cauchy, then for any given $\epsilon > 0$, there exists an N such that for all $k, m \geq N$,*

$$|a^m(i) - a^m(i)| < \epsilon$$

for $i = 1, 2, \ldots, n$.

Proof We will prove the result for the max norm. The result can be proved for any norm. Since $\{\vec{a}^p\}_{p=1}^{\infty}$ is Cauchy, we have that for a given value of $\epsilon > 0$, there exists an N such that for any $m, k \geq N$,

$$||\vec{a}^k - \vec{a}^m|| < \epsilon.$$

From the definition of the max norm, we have that for any $k, m \geq N$,

$$||\vec{a}^k - \vec{a}^m|| = \max_i |a^k(i) - a^m(i)|.$$

Combining the information in the preceding equation and the inequation before it, one has that for any $k, m \geq N$,

$$\max_i |a^k(i) - a^m(i)| < \epsilon.$$

Now in this equation, the $<$ relation holds for $\max_i$ in the left hand side. Hence the result must be true for all i. Thus, for any $k, m \geq N$,

$$|a^k(i) - a^m(i)| < \epsilon$$

is true for all i. This implies that each coordinate sequence is Cauchy in $\Re$. ▮

We next need to understand an important concept that plays a central role in the convergence of discounted reinforcement learning algorithms. It is important that you digest the main idea around which the next section revolves.

10. Contraction Mappings in $\Re^n$

This section is about a special type of mapping (or transformation)—a so-called **contraction** mapping or contractive mapping. Let us begin with some geometric insight into this idea. To this end, consider two distinct vectors, $\vec{a}$ and $\vec{b}$, in $\Re^2$ space. Let us define $\vec{c}$ as follows:

$$\vec{c} = \vec{a} - \vec{b}.$$

Then $\vec{c}$ denotes a difference of the two vectors. Now apply a mapping F to both $\vec{a}$ and $\vec{b}$. We will be applying this transformation repeatedly.

Let us define the notation to be used when the mapping F is applied repeatedly. We will use $\vec{x}^k$ to denote the vector obtained after k applications of F to the vector $\vec{x}$. As such $\vec{a}^0$ will stand for $\vec{a}$, $\vec{a}^1$ will stand for $F\vec{a}^0$, $\vec{a}^2$ will stand for $F^2\vec{a}^0$, and so on. The difference between the transformed vectors $\vec{a}^k$ and $\vec{b}^k$ will be denoted by $\vec{c}^k$. Thus:

$$\vec{c}^1 \equiv F(\vec{a}) - F(\vec{b}), \text{ and } \vec{c}^2 \equiv F^2(\vec{a}) - F^2(\vec{b}), \text{ and so on.}$$

If the length (norm) of the vector $\vec{c}^1$ is strictly smaller than that of $\vec{c}^0$, then the difference vector can be said to have become smaller. In other words, the difference vector can be said to have "shrunk." If this is true of every occasion the mapping is applied, then we go on to declare that the mapping is **contractive** in that space. What is more interesting is that the vectors $\vec{a}$ and $\vec{b}$ keep **approaching each other** with every application of such a mapping. This happens because the difference between the two vectors keeps getting smaller and smaller. Eventually, the vectors will become **identical**. This vector—that will be obtained ultimately—is called a **fixed point** of the mapping. Essentially, this implies that given any vector ($\vec{a}$ or $\vec{b}$ or any other vector), if one keeps applying the transformation F, one eventually obtains the same vector. (The final vector obtained in the limit is usually called the fixed point.) We next provide a technical definition of a fixed point.

Definition 9.7 *$\vec{x}$ is said to be a fixed point of the transformation F if*

$$F\vec{x} = \vec{x}.$$

Note that the definition says nothing about contractions, and in fact, F may have a fixed point even if it is not contractive.

Let us illustrate the idea of generating a fixed point with a contractive mapping using an example. Consider the following vectors.

$$\vec{a}^0 = (4, 2), \text{ and } \vec{b}^0 = (2, 2).$$

We will apply the transformation G in which $x^k(i)$ will denote the ith component of the vector $\vec{x}^k$. Let us assume that G is contractive (this can be proved) and is defined as:

$$x^{k+1}(1) = 5 + 0.2x^k(1) + 0.1x^k(2),$$

$$x^{k+1}(2) = 10 + 0.1x^k(1) + 0.2x^k(2).$$

Table 9.1 shows the results of applying transformation G repeatedly. A careful observation of the table will reveal the following. With every iteration, the difference vector $\vec{c}^k$ becomes smaller. Note that when $k = 12$, the vectors $\vec{a}^k$ and $\vec{b}^k$ have become one, if one ignores anything beyond the fifth place after the decimal point. The two vectors will become one, generally speaking, when $k = \infty$. In summary, one starts with two different vectors but eventually goes to a **fixed point** which, in this case, seems to be very close to $(7.936507, 13.492062)$.

What we have demonstrated above is an important property of a contractive mapping. One may start with **any** vector, but successive applications of the mapping transforms the vector into a unique vector.

See Figs. 9.1–9.3 to get a geometric feel for how a contraction mapping keeps "shrinking" the vectors in $\Re^2$ space, and carries any given vector to a unique fixed point. The figures are related to the data given in Table 9.1. The contraction mapping is very much like a dog that carries every bone (read vector) that it gets to its own hidey hole (read a unique fixed point), regardless of the size of the bone or where the bone has come from.

Let us now examine a more mathematically precise definition of a contraction mapping.

Definition 9.8 *A mapping (or transformation) F is said to be a contraction mapping in $\Re^n$ if there exists a λ such that $0 \leq \lambda < 1$ and*

$$||F\vec{v} - F\vec{u}|| \leq \lambda||\vec{v} - \vec{u}||$$

for all $\vec{v}, \vec{u}$ in $\Re^n$.

As is clear from the definition, the norm represents what was referred to as "length" in our informal discussion on contraction mappings.

By applying F repeatedly, one obtains a **sequence** of vectors. Consider a vector $\vec{a}$ on which the transformation F is applied repeatedly. This will form a sequence of vectors, which we will denote by $\{\vec{a}^k\}_{k=0}^{\infty}$. Here $\vec{a}^k$ denotes the kth term of the sequence. It must be noted that each term is itself a vector. The relationship between the terms is given by

$$\vec{v}^{k+1} = F\vec{v}^k.$$

Table 9.1. Table showing the change in values of the vectors $\vec{a}$ and $\vec{b}$ after repeated applications of G

k	$a^k(1)$	$a^k(2)$	$b^k(1)$	$b^k(2)$	$c^k(1)$	$c^k(2)$
0	4.000000	2.000000	2.000000	2.000000	2.000000	0.000000
1	6.000000	10.800000	5.600000	10.600000	0.400000	0.200000
2	7.280000	12.760000	7.180000	12.680000	0.100000	0.080000
3	7.732000	13.280000	7.704000	13.254000	0.028000	0.026000
4	7.874400	13.429200	7.866200	13.421200	0.008200	0.008000
5	7.917800	13.473280	7.915360	13.470860	0.002440	0.002420
6	7.930888	13.486436	7.930158	13.485708	0.000730	0.000728
7	7.934821	13.490376	7.934602	13.490157	0.000219	0.000219
8	7.936002	13.491557	7.935936	13.491492	0.000066	0.000066
9	7.936356	13.491912	7.936336	13.491892	0.000020	0.000020
10	7.936462	13.492018	7.936456	13.492012	0.000006	0.000006
11	7.936494	13.492050	7.936492	13.492048	0.000002	0.000002
12	7.936504	13.492059	7.936503	13.492059	0.000001	0.000001
13	**7.936507**	**13.492062**	**7.936507**	**13.492062**	0.000000	0.000000

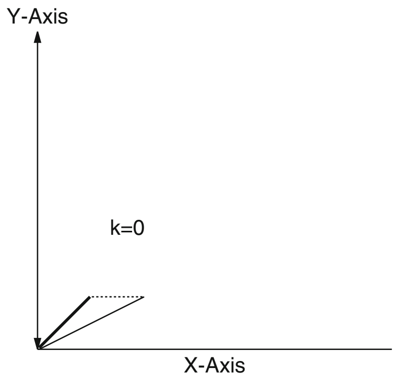

Figure 9.1. The *thin line* represents vector $\vec{a}$, the *dark line* represents the vector $\vec{b}$, and the *dotted line* the vector $\vec{c}$. This is before applying G

Thus the sequence can also be denoted as: $\{\vec{v}^0, \vec{v}^1, \vec{v}^2, \ldots\}$. It is clear that $\vec{v}^1 = F\vec{v}^0, \vec{v}^2 = F\vec{v}^1$, and so on. Now if $||F\vec{v}^0 - F\vec{u}^0|| \leq \lambda||\vec{v}^0 - \vec{u}^0||$ for all vectors in $\Re^n$, then

$$||F^2\vec{v}^0 - F^2\vec{u}^0|| \quad \leq \quad \lambda||F\vec{v}^0 - F\vec{u}^0|| \leq \lambda^2||\vec{v}^0 - \vec{u}^0||.$$

$$\text{Similarly, } ||F^3\vec{v}^0 - F^3\vec{u}^0|| \leq \lambda||F^2\vec{v}^0 - F^2\vec{u}^0|| \leq \lambda^2||F\vec{v}^0 - F\vec{u}^0|| \leq \lambda^3||\vec{v}^0 - \vec{u}^0||.$$

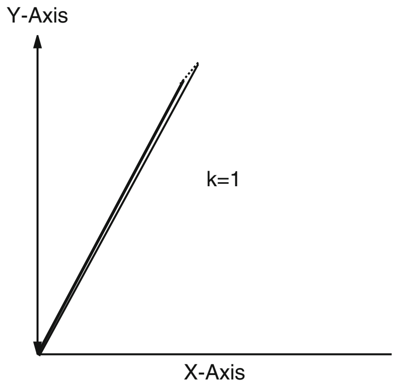

Figure 9.2. The *thin line* represents vector $\vec{a}$, the *dark line* represents the vector $\vec{b}$, and the *dotted line* the vector $\vec{c}$. This is the picture after one application of G. Notice that the vectors have come closer

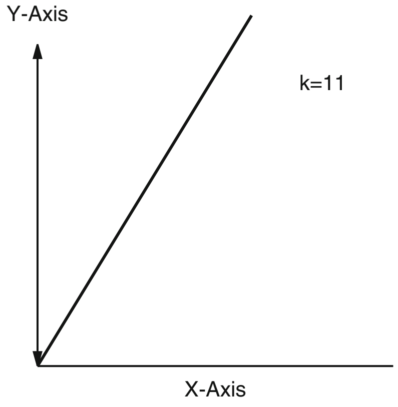

Figure 9.3. This is the picture after 11 applications of G. By now the vectors are almost on the top of each other, and it is difficult to distinguish between them

$$\text{In general, } ||F^m\vec{v}^0 - F^m\vec{u}^0|| \leq \lambda^m||\vec{v}^0 - \vec{u}^0||. \tag{9.7}$$

Inequality (9.7) can be proved from Definition 9.8 using induction. This is left as an exercise.

Remark. Consider F to be a contraction mapping in $\Re^n$. As such for any two vectors $\vec{x}$ and $\vec{y}$ in $\Re^n$, one has that:

$$||F\vec{x} - F\vec{y}|| \leq \lambda||\vec{x} - \vec{y}||. \tag{9.8}$$

Now consider a vector sequence $\{\vec{a}^k\}_{k=0}^{\infty}$ in which $\vec{a}^1 = F\vec{a}^0, \vec{a}^2 = F\vec{a}^1$, and so on, where $\vec{a}^k$ is a member of $\Re^n$. Then, one has that

$$\begin{aligned} ||\vec{a}^1 - \vec{a}^2|| &= ||F\vec{a}^0 - F\vec{a}^1|| \\ &\leq \lambda ||\vec{a}^0 - \vec{a}^1||. \end{aligned} \tag{9.9}$$

Inequality (9.9) follows from Inequality (9.8).

We are now ready to prove a major theorem that will prove very useful in the convergence analysis of some dynamic programming algorithms.

THEOREM 9.10 **(Fixed Point Theorem)** *Suppose F is a contraction mapping in $\Re^n$. Then*

- *There exists a unique vector, which we denote by $\vec{v}_*$, in $\Re^n$ such that $F\vec{v}_* = \vec{v}_*$ and*
- *For any $\vec{v}^0$ in $\Re^n$, the sequence $\{\vec{v}^k\}_{k=0}^{\infty}$ defined by*

$$\vec{v}^{k+1} = F\vec{v}^k = F^k\vec{v}^0 \tag{9.10}$$

converges to $\vec{v}_*$.

Here $\vec{v}_*$ denotes the fixed point of F.

Proof For any $m' > m$,

$$\begin{aligned} ||\vec{v}^m - \vec{v}^{m'}|| = \; & ||F^m\vec{v}^0 - F^{m'}\vec{v}^0|| && (9.11) \\ \leq \; & \lambda^m ||\vec{v}^0 - \vec{v}^{m'-m}|| && (9.12) \\ = \; & \lambda^m ||\vec{v}^0 - \vec{v}^1 + \vec{v}^1 - \vec{v}^2 + \ldots \\ + \; & \ldots + \vec{v}^{m'-m-1} - \vec{v}^{m'-m}|| \\ \leq \; & \lambda^m \; [||\vec{v}^0 - \vec{v}^1|| + ||\vec{v}^1 - \vec{v}^2|| + \ldots \\ + \; & \ldots + ||\vec{v}^{m'-m-1} - \vec{v}^{m'-m}||] && (9.13) \\ = \; & \lambda^m ||\vec{v}^0 - \vec{v}^1|| \cdot \\ & [1 + \lambda + \lambda^2 + \ldots + \lambda^{m'-m-1}] && (9.14) \\ = \; & \lambda^m ||\vec{v}^0 - \vec{v}^1|| \left[\frac{1 - \lambda^{m'-m}}{1 - \lambda}\right] && (9.15) \\ < \; & \lambda^m ||\vec{v}^0 - \vec{v}^1|| \left[\frac{1}{1 - \lambda}\right] && (9.16) \end{aligned}$$

In the above:

- Line (9.11) follows from the definition of F (see Eq. (9.10)).
- Line (9.12) follows from Inequation (9.7) by setting $\vec{u}^0 = \vec{v}^{m'-m}$, which can be shown to imply that:
$$F^m \vec{u}^0 = F^{m'} \vec{v}^0.$$
- Line (9.13) follows from Property 3 of norms given in Sect. 3.
- Line (9.14) follows from the Remark related to Inequation (9.9).
- Line (9.15) follows from the sum of a finite GP series given in Eq. (9.3).
- Line (9.16) follows from the fact that $\lambda^{m'-m} > 0$ since $\lambda > 0$ and $m' > m$.

From (9.16), one can state that by selecting a large enough value for m, $||\vec{v}(m) - \vec{v}(m')||$ can be made as small as needed. In other words, for a given value of ϵ, which satisfies $\epsilon > 0$, one can come up with a finite value for m such that

$$||\vec{v}^m - \vec{v}^{m'}|| < \epsilon.$$

Since $m' > m$, the above ensures that the vector sequence $\{\vec{v}^k\}_{k=0}^{\infty}$ is Cauchy. From Lemma 9.9, it follows that if the vector sequence satisfies the Cauchy condition (see Definition 9.6), then each coordinate sequence is also Cauchy. From Theorem 9.5, a Cauchy sequence converges and thus each coordinate sequence converges to a finite number. Consequently, the vector sequence converges to a finite valued vector. Let us denote the limit by $\vec{v}_*$. Hence we have that

$$\lim_{k\to\infty} ||\vec{v}^k - \vec{v}_*|| = 0. \tag{9.17}$$

Now we need to show that $F\vec{v}_* = \vec{v}_*$. From norms' properties (Sect. 3), it follows that

$$\begin{aligned}
0 \leq\ & ||F\vec{v}_* - \vec{v}_*|| \\
=\ & ||F\vec{v}_* - \vec{v}^k + \vec{v}^k - \vec{v}_*|| \\
\leq\ & ||F\vec{v}_* - \vec{v}^k|| + ||\vec{v}^k - \vec{v}_*|| \\
=\ & ||F\vec{v}_* - F\vec{v}^{k-1}|| + ||\vec{v}^k - \vec{v}_*|| \\
\leq\ & \lambda||\vec{v}_* - \vec{v}^{k-1}|| + ||\vec{v}^k - \vec{v}_*|| \qquad (9.18)
\end{aligned}$$

From (9.17), both terms of the right hand side of (9.18) can be made arbitrarily small by choosing a large enough k. Hence

$$||F\vec{v}_* - \vec{v}_*|| = 0.$$

This implies that $F\vec{v}_* = \vec{v}_*$, and so $\vec{v}_*$ is a fixed point of F.

What remains to be shown is that the fixed point $\vec{v}_*$ is unique. To prove this, let us assume that there are **two** vectors $\vec{a}$ and $\vec{b}$ that satisfy $\vec{x} = F\vec{x}$. Hence

$$\vec{a} = F\vec{a}, \text{ and } \vec{b} = F\vec{b}.$$

Then using the contraction property, one has that:

$$\begin{aligned} \lambda||\vec{b} - \vec{a}|| &\geq ||F\vec{b} - F\vec{a}|| \\ &= ||\vec{b} - \vec{a}|| \end{aligned}$$

which implies that:

$$\lambda||\vec{b} - \vec{a}|| \geq ||\vec{b} - \vec{a}||.$$

Since a norm is always non-negative, and since $\lambda < 1$, the above must imply that $||\vec{b} - \vec{a}|| = 0$. As a result, $\vec{a} = \vec{b}$, and uniqueness follows. ∎

11. Stochastic Approximation

Stochastic approximation is the science underlying the usage of step-size-based schemes that exploit random numbers for estimating quantities such as means and gradients. Examples of stochastic approximation schemes are the simultaneous perturbation algorithm and most reinforcement learning algorithms. In this section, we collect together some important definitions and an important result related to stochastic approximation theory. We will first present an important idea related to convergence of sequences, which will be needed subsequently.

11.1. Convergence with Probability 1

The notion of convergence with probability 1 needs to be understood at this stage. This concept is somewhat different from the idea of convergence of a sequence discussed above.

Definition 9.9 *A sequence $\{x^k\}_{k=0}^{\infty}$ of random variables is said to converge to a random number x_* with probability 1 if for a given $\epsilon > 0$ and a given $\delta > 0$, there exists an integer N such that*

$$\mathsf{P}\left[|x^k - x_*| < \epsilon\right] > 1 - \delta \textit{ for all } k \geq N,$$

$$\text{i.e., } \mathsf{P}\left[\lim_{k\to\infty} x^k = x_*\right] = 1.$$

In the above definition, note that N may depend on both ϵ and δ. The definition implies that if a sequence tends to a limit with probability 1, then you can make δ as small as you want, and still obtain a finite value for N, at which the sequence is arbitrarily close (ϵ-close) to the limit. The definition given above could be extended to a vector via replacement of $|x^k - x_*|$ by the norm $||\vec{x}^k - \vec{x}_*||$. This type of convergence is also called *almost sure* convergence.

When we have convergence of this kind, for a given value of $\epsilon > 0$, the probability with which the absolute value of the difference between the value of the sequence and its limit is greater than ϵ can be made as small as one wants. However, there will always be that small probability (at most δ) with which the absolute value of the difference may differ from the limit by a distance *greater* than ϵ. Note that in our previous definition of convergence, there was no such probability. Further note that in the definition above, the sequence does *not* necessarily have to be a sequence of *independent* random variables. Although the strong law of large numbers uses this concept of convergence, the sequence in it has to one of independent random variables.

11.2. Ordinary Differential Equations

It can be shown under certain conditions that underlying a stochastic approximation scheme, e.g., that used in reinforcement learning and simultaneous perturbation, there exists an ordinary differential equation (ODE). Before we study the ODE underlying a stochastic approximation scheme, we need to study some important properties related to ODEs in general. In this subsection, we will consider properties that we will need later in the book.

11.2.1 Equilibrium Points

Consider an ODE in two variables x and t:

$$\frac{d\vec{x}}{dt} = f(\vec{x}, t), \tag{9.19}$$

where $f(\vec{x}, t)$ is a function of $\vec{x}$ and t, where $\vec{x} \in \Re^n$ and $t \in \Re$. We now present an important definition.

Definition 9.10 *The equilibrium point, also called a critical point, of the ODE in Eq. (9.19) is defined to be a vector $\vec{x}_*$ which satisfies the following:*

$$f(\vec{x}_*, t) = 0.$$

In other words, the function $f(\vec{x}, t)$ equals 0 at the equilibrium point. We need to understand that the equilibrium point of an ODE does not have to be unique, i.e., there may be more than one equilibrium point for an ODE.

11.2.2 A Ball

We define an *open ball* in $\Re^n$ that is centered on $\vec{x}$ and has a radius of $r \in \Re$, where $r > 0$, to be the following set:

$$B(\vec{x}, r) = \{\vec{y} \in \Re^n : ||\vec{x} - \vec{y}|| < r\},$$

where $||.||$ denotes any norm. For instance, for $n = 1$, the ball is simply the open interval: $(x - r, x + r)$. Similarly, if $n = 2$, then the ball can be viewed geometrically as an open circle whose radius is r and center is $\vec{x}$.

11.2.3 Solution of an ODE

The solution of the ODE in (9.19) will be denoted by $\vec{x}(t)$ to indicate that it is a function of t.

Example: Consider the ODE given by

$$\frac{dx}{dt} = -\frac{x}{1+t}. \tag{9.20}$$

By separation of variables, we have that the solution is

$$x = \frac{x_0(1+t_0)}{1+t},$$

where x_0 and t_0 are obtained from the boundary condition to eliminate the constant of integration. Here, we can denote the solution in general as:

$$\phi(t) = \frac{x_0(1+t_0)}{1+t}.$$

Important note: The function, $\phi(t)$, is the **solution** of the ODE, and is a very important tool that will be used in our analysis. Notation employed commonly in the literature on stochastic approximation uses $x(t)$ to denote the solution, $\phi(t)$, and also to denote the variable, x, itself, but we will refrain from using $x(t)$ in this book—in order to avoid confusion. In general, when we are dealing with vectors, the solution will be denoted by $\vec{\phi}(t)$, while $\vec{x}$ will denote the variable.

11.2.4 Stable Equilibrium

DEFINITION 9.11 *An equilibrium point $\vec{x}_*$ is said to be a stable equilibrium point of the ODE in Eq. (9.19) if and only if for all values of $\epsilon > 0$, there exists a scalar $r(\epsilon) > 0$, where $r(\epsilon)$ may possibly depend on ϵ, such that if $\vec{\phi}(0) \in B(\vec{x}_*, r(\epsilon))$, where $B(.)$ is a ball of radius $r(\epsilon)$ centered on $\vec{x}_*$ and $\vec{\phi}(t)$ is the solution of the ODE concerned, then for all t, $\vec{\phi}(t) \in B(\vec{x}_*, \epsilon)$.*

What this definition implies is that if a stable solution of the ODE starts at a point close to the equilibrium point (i.e., within a distance of $r(\epsilon)$ from the equilibrium point), then it always remains ϵ-close to the equilibrium point. Thus, when the equilibrium point is stable, if one starts at a solution close enough to the equilibrium point, then one remains close to it throughout the trajectory pursued by the solution (i.e., even when t tends to infinity). It is easy to see that the solution $x = 0$ for the ODE in (9.20) is an equilibrium point, and it can also be shown to be a stable equilibrium.

11.2.5 Asymptotically Stable Equilibrium

DEFINITION 9.12 *An equilibrium point $\vec{x}_*$ of the ODE in Eq. (9.19) is said to be an asymptotically stable equilibrium point if it is stable and in addition:*

$$\lim_{t\to\infty} \vec{\phi}(t) = \vec{x}_*, \text{ where } \vec{\phi}(t) \text{ denotes the solution of the ODE.}$$

The implication is that the equilibrium point is not only stable but in addition, eventually the ODE's solution will converge to the equilibrium point. In the ODE of Eq. (9.20), where $x = 0$ is the unique equilibrium point, it is easy to see that:

$$\lim_{t\to\infty} \phi(t) = 0,$$

and hence $x = 0$ is an asymptotically stable equilibrium.

When the solution $\vec{\phi}(t)$ converges to $\vec{x}_*$ for *any* initial condition, the asymptotically stable equilibrium is called a **globally** asymptotically stable equilibrium. For example consider the ODE in Eq. (9.20). For all values of x_0 and t_0, the solution $\phi(t)$ will converge to 0 (i.e., $x = 0$, the equilibrium) as t tends to ∞. Thus, regardless of the initial conditions, the solution converges to the critical point, and hence here, $x = 0$ is a globally asymptotically stable equilibrium point for that ODE.

11.3. Stochastic Approximation and ODEs

We have already seen the so-called "stochastic approximation" scheme, which is step-size-based and has random noise in it, in the context of simultaneous perturbation and reinforcement learning. In this section, we will study some important mathematical properties of this scheme under the so-called "synchronous conditions" of updating (explained below). We will present a framework that captures the behavior of such a scheme under such conditions. A scheme that is step-size-based but noise-free can also be analyzed via the framework that we will present below.

The specific framework that we are interested in here is often called the ordinary differential equation (ODE) framework of stochastic approximation. It essentially shows that the behavior of the stochastic approximation scheme (algorithm) can be modeled by an ODE. We first present some notation that will be useful in describing this framework.

We begin with some definitions that we need.

DEFINITION 9.13 *The function $f(x)$ is said to Lipschitz continuous on the set $\mathcal{X}$ if a finite number $K \in \Re$, where $K \geq 0$, exists such that for all $x_1, x_2 \in \mathcal{X}$*

$$||f(x_2) - f(x_1)|| \leq K||x_2 - x_1||.$$

The above also applies when x is replaced by a vector $\vec{x}$. In order to establish Lipschitz continuity of a function, frequently the following test is applied. One computes the derivative of the function. If the derivative can be shown to be bounded, the function is Lipschitz continuous. Hence all linear functions are Lipschitz continuous. Here is how we can illustrate this. Consider the function:

$$f(\vec{x}) = 3 + 2x(1) - 9x(2).$$

It is Lipschitz continuous in $\vec{x}$ since $\frac{\partial f(\vec{x})}{\partial x(1)} = 2$ and $\frac{\partial f(\vec{x})}{\partial x(2)} = -9$.

A example of a function that is not Lipschitz continuous is $f(x) = 2 + 4x^2$ where $x \in \Re$. The derivative is $8x$, which is unbounded when $x \in \Re$. However, for the case where the domain of x is a bounded set, e.g., $(5, 42)$, clearly, $8x$ is bounded, and hence $f(x)$ is Lipschitz over that domain.

A stochastic approximation scheme (algorithm) that works "synchronously" is defined as follows.

DEFINITION 9.14 *Consider the following algorithm in which the values of $\vec{X}$ belonging to $\Re^N$ are updated iteratively. The value of $\vec{X}$*

in the kth iteration of the algorithm will be denoted by $\vec{X}^k$. Then, the synchronous stochastic approximation (updating) scheme (or algorithm) is given as follows. For $l = 1, 2, \ldots, N$,

$$X^{k+1}(l) = X^k(k) + \alpha^k \left[F(\vec{X}^k)(l) + w^k(l) \right], \qquad (9.21)$$

where α^k is a positive scalar called step size, whose value is generally less than 1, $F(.)$ is a continuous function from $\Re^N$ to $\Re^N$, $F(\vec{X})(l)$ denotes the lth component of the vector, $F(\vec{X})$, and $w^k(l)$ denotes the noise term involved in the kth iteration of the algorithm while updating $x^k(l)$.

The descriptor "synchronous" in the above indicates that when the lth element of $\vec{X}^k$ is being updated, all the values used in the update (9.21) belong to the kth iteration. Thus, in the above updating equation, (9.21), we find that all values in the right-hand side belong to the kth iteration. For a synchronous algorithm to work, one must update *all* elements of $\vec{X}$ in a given iteration *before* moving on to the next iteration.

Examples: We have seen synchronous algorithms with step sizes in simultaneous perturbation. In reinforcement learning, we use step sizes, but the updating is asynchronous. Imagine for the time being that we are using reinforcement learning in a synchronous manner. Then, for an example of $\vec{X}^k$, think of the Q-factors in reinforcement learning. Each Q-factor can then be viewed as a component of the iterate. Thus, the lth component of $\vec{X}^k$, i.e., $X^k(l) = Q^k(i, a)$ where $l = (i, a)$.

Noise: The meaning of the so-called "noise term" will become clearer when we analyze specific algorithms. For the time being, assume it to be an additional term that is carried within the algorithm.

In what follows, we will use the term $\mathcal{F}^k$ to denote the history of the algorithm from the first iteration up to and including the kth iteration. We now define it formally.

Definition 9.15 *The history of the stochastic approximation algorithm until and including the kth iteration, $\mathcal{F}^k$, is defined as:*

$$\mathcal{F}^k = \{\vec{X}^1, \vec{X}^2 \ldots, \vec{X}^k, \vec{w}^1, \vec{w}^2 \ldots, \vec{w}^k\}.$$

We now make some assumptions about our algorithm before stating an important result related to it. These assumptions are essentially conditions on our algorithm. For the result that we will present below, all of these conditions should hold.

ASSUMPTION 9.11 *The function $F(.)$ is Lipschitz continuous.*

The precise implication of the following condition will become clearer later. Essentially what the following condition ensures is that the effect of the noise vanishes in the limit, i.e., as if it never existed!

ASSUMPTION 9.12 *For $l = 1, 2, \ldots, N$ and for every k, the following should be true about the noise terms:*

$$\mathsf{E}\left[w^k(l)|\mathcal{F}^k\right] = 0;$$

$$\mathsf{E}\left[\left(w^k(l)\right)^2\middle|\mathcal{F}^k\right] \leq z_1 + z_2||\vec{X}^k||^2; \tag{9.22}$$

where z_1 and z_2 are scalar constants and $||.||$ could be any norm.

It is not hard to see that the first condition within the assumption above essentially states that the conditional expectation of the noise (the condition being that the history of the algorithm is known to us) is 0. The second condition in (9.22) states that the second (conditional) moment of the noise is bounded by a function of the iterate. If the iterate is bounded, this condition holds.

ASSUMPTION 9.13 *The step size α^k satisfies the following conditions:*

$$\sum_{k=1}^{\infty} \alpha^k = \infty; \tag{9.23}$$

$$\sum_{k=1}^{\infty} \left(\alpha^k\right)^2 < \infty. \tag{9.24}$$

The above conditions are the famous **tapering size** conditions imposed on all stochastic approximation schemes. When noise is not present, the condition in (9.24) is not needed.

ASSUMPTION 9.14 $\left\{\vec{X}^k\right\}_{k=1}^{\infty}$ *remains bounded with probability 1.*

The above condition (i.e., boundedness of the iterates) is almost always needed in convergence of stochastic approximation. We now present some background for the last condition.

A major contribution of stochastic approximation theory has been to establish that underlying each component of the iterate, there exists a continuous-valued variable. Remember that in reinforcement learning and in simultaneous perturbation, the iterate changes values in discrete steps; the actual change depends on the step size amongst other factors. For instance, a Q-factor may change from a value of 8.1 to 8.3 in one iteration. The imaginary continuous-valued variable that lies under the iterate allows us to model a continuous change in values provided the step sizes are small enough. Further, the theory of stochastic approximation shows that for the purpose of analysis, the algorithm can be replaced by an ordinary differential equation (ODE) involving the continuous-valued variable and that the solution of the ODE can be used to study the algorithm's progress.

In mathematically sophisticated language, the sequence of values $\{\vec{X}^k\}_{k=1}^{\infty}$ can be replaced by the solution, $\vec{\phi}(t)$, to an ODE of the form in Eq. (9.19), in particular:

$$\frac{d\vec{x}}{dt} = F(\vec{x}), \text{ where note:} \tag{9.25}$$

- The lowercase letter x (or $\vec{x}$ in case of vectors) will denote the continuous-valued variable underlying the iterate X (or $\vec{X}$ in case of vectors).

- The scalar variable t in the ODE will play the role of the iteration number k.

Then, the solution of this ODE, $\vec{\phi}(t)$, will essentially replace the trajectory of our algorithm, and

$$\lim_{t\to\infty} \vec{\phi}(t)$$

will represent the behavior of the algorithm as k tends to infinity (i.e., in the long run). The reason for exploiting the ODE should perhaps be obvious now: the solution of the ODE better be the solution to which our algorithm *should* converge! We now present a critical condition related to the ODE.

ASSUMPTION 9.15 *The ODE in Eq. (9.25) has a unique asymptotically stable equilibrium point, which we denote by $\vec{x}_*$.*

If the condition above is unclear (especially if you wonder how an ODE has appeared out of nowhere!), please study the following simple example.

Example: Consider the following simple algorithm:

$$X^{k+1} \leftarrow X^k + \alpha^k \left[5X^k\right], \text{ where } X^k \in \Re \text{ for all } k.$$

The ODE for this algorithm will be (compare the above to Eq. (9.21)):

$$\frac{dx}{dt} = 5x.$$

Thus, all you need to identify the ODE is the algorithm's transformation, $F(.)$.

We are now at a position to present the important result from [184], which forms the cornerstone of convergence theory of stochastic approximation schemes via the ODE method.

THEOREM 9.16 *Consider the synchronous stochastic approximation scheme defined in Eq. (9.21). If Assumptions 9.11—9.15 hold, then with probability 1, the sequence* $\left\{\vec{X}^{\,k}\right\}_{k=1}^{\infty}$ *converges to* $\vec{x}_*$.

The proof of the above is rather deep, and involves some additional results that are beyond the scope of this text. The result is very powerful; it implies that if we can show these assumptions to hold, the algorithm is guaranteed to converge. The implications of the result are somewhat intuitive, and we will discuss those below. Also, we will show in subsequent chapters that this result, or some of its variants, can be used to show the convergence of simultaneous perturbation and many reinforcement learning algorithms.

We note that the result above also holds in a noise-free setting, i.e., when the noise term $w(l) = 0$ for every l in every iteration of the algorithm. As stated above, when we have a noise-free algorithm, the condition in (9.24) in Assumption 9.13 is not needed.

The intuition underlying the above result is very appealing. It implies that the effect of noise in the noisy algorithm will *vanish in the limit* if Assumption 9.12 is shown to be true. In other words, it is as if noise never existed in the algorithm and that we were using the following update:

$$X^{k+1}(l) = X^k(l) + \alpha^k \left[F(\vec{X}^k)(l)\right]$$

in which there is no noise. Note that if the above scheme converges to some point, say $\vec{Y}$, we would have that $\lim_{k\to\infty} X^{k+1}(l) = \lim_{k\to\infty} X^k(l) \equiv Y(l)$ for every l, which would lead us to

$$Y(l) = Y(l) + \alpha[F(\vec{Y})(l)], \text{ for every } l, \text{ which implies that for every } l$$

$$F(\vec{Y})(l) = 0. \tag{9.26}$$

In general, in stochastic approximation, we seek the above solution. In the context of reinforcement learning, the function $F(.)$ will be designed in a manner such that (9.26) is the solution of the Bellman equation, while in the context of simultaneous perturbation, (9.26) will define a stationary point of the function that we are seeking to optimize.

Bibliographic Remarks. All the material in this chapter until Sect. 11 is classical, and some of it is more than a hundred years old. Consequently, most of this material can be found in any standard text on mathematical analysis. The results that we presented will be needed in subsequent chapters. Gaughan [95] and Douglass [81] cover most of the topics dealt with here until Sect. 11. The fixed point theorem can be found in Rudin [254].

Material in Sect. 11 on ODEs can be found in [54], and Theorem 9.16 is from Kushner and Clark [184] (see also [185, 48]). Work on ODEs and stochastic approximation has originated from the work of Ljung [192].

Chapter 10

CONVERGENCE ANALYSIS OF PARAMETRIC OPTIMIZATION METHODS

1. Chapter Overview

This chapter deals with some simple convergence results related to the parametric optimization methods discussed in Chap. 5. The main idea underlying convergence analysis of an algorithm is to identify (mathematically) the solution to which the algorithm converges. Hence to prove that an algorithm works, one must show that the algorithm converges to the optimal solution. In this chapter, this is precisely what we will attempt to do with some algorithms of Chap. 5.

The convergence of simulated annealing requires some understanding of Markov chains and transition probabilities. To this end, it is sufficient to read all sections of Chap. 6 up to and including Sect. 3.1.3. It is also necessary to read about convergent sequences. For this purpose, it is sufficient to read all the material in Chap. 9 up to and including Theorem 9.2. Otherwise, all that is needed to read this chapter is a basic familiarity with the material of Chap. 5.

Our discussion on the analysis of the steepest-descent rule begins in Sect. 3. Before discussing the mathematical details, we review definitions of some elementary ideas from calculus and a simple theorem.

2. Preliminaries

In this section, we define some basic concepts needed for understanding convergence of continuous parametric optimization. The material in this section should serve as a refresher. All of these concepts will be required in this chapter. Readers familiar with them can skip this section without loss of continuity.

A. Gosavi, *Simulation-Based Optimization*, Operations Research/Computer Science Interfaces Series 55,
DOI 10.1007/978-1-4899-7491-4_10

2.1. Continuous Functions

DEFINITION 10.1 *A function $f(\vec{x})$ defined as $f : \mathcal{D} \to \Re$ is said to be continuous on the set $\mathcal{D}$ if and only if*

$$\lim_{\vec{x}\to\vec{c}} f(\vec{x}) = f(\vec{c}) \textit{ for every } \vec{c} \in \mathcal{D}.$$

We now illustrate this idea with the following example function from $\Re$ to $\Re$:

$$f(x) = 63x^2 + 5x.$$

Note that the function must be continuous since for any $c \in \Re$,

$$\lim_{x\to c} f(x) = \lim_{x\to c}(63x^2 + 5x) = 63c^2 + 5c = f(c).$$

Now consider the following example. A function $f : \Re \to \Re$ is defined as:

$$f(x) = \frac{x^2 - 5x + 6}{x^2 - 6x + 8} \text{ when } x \neq 2; f(x) = 90 \text{ when } x = 2.$$

Now, at $x \neq 2$, we have:

$$\lim_{x\to 2} f(x) = \lim_{x\to 2} \frac{x^2 - 5x + 6}{x^2 - 6x + 8} = \lim_{x\to 2} \frac{(x-2)(x-3)}{(x-2)(x-4)} = \lim_{x\to 2} \frac{x-3}{x-4} = \frac{1}{2} \neq 90.$$

This implies, from the definition above, that the function is not continuous at $x = 2$, and hence the function is not continuous on $\Re$.

2.2. Partial Derivatives

DEFINITION 10.2 *The partial derivative of a function of multiple variables $(x(1), x(2), \ldots, x(k))$ with respect to the ith variable, $x(i)$, is defined as*

$$\frac{\partial f(x(1), x(2), \ldots, x(k))}{\partial x(i)} \equiv$$

$$\lim_{h\to 0} \frac{f(x(1), x(2), \ldots, x(i) + h, \ldots + x(k)) - f(x(1), x(2), \ldots, x(k))}{h}.$$

The partial derivative defined here is actually the *first* partial derivative.

2.3. A Continuously Differentiable Function

DEFINITION 10.3 *A function $f : \mathcal{D} \to \Re$ is said to be continuously differentiable if each of its partial derivatives is a continuous function on the domain $(\mathcal{D})$ of the function f.*

Example: Consider the function

$$f(x, y) = 5x^2 + 4xy + y^3.$$

$$\text{Then, } \frac{\partial f(x,y)}{\partial x} = 10x + 4y, \text{ and} \frac{\partial f(x,y)}{\partial y} = 4x + 3y^2.$$

It is not hard to show that both partial derivatives are continuous functions. Hence, the function must be continuously differentiable.

2.4. Stationary Points and Local and Global Optima

DEFINITION 10.4 *A stationary point of a function is the point (vector) at which the first partial derivative has a value of 0.*

Hence if the function is denoted by $f(x(1), x(2), \ldots, x(k))$, then the stationary point can be determined by solving the following system of linear equations (composed of k equations):

$$\frac{\partial f(x(1), x(2), \ldots, x(k))}{\partial x(j)} = 0, \text{ for } j = 1, 2, \ldots, k.$$

DEFINITION 10.5 *A local minimum of a function $f : \Re^k \to \Re$ is a point, $\vec{x}_*$, which is no worse than its neighbors; i.e., there exists an $\epsilon > 0$ such that:*

$$f(\vec{x}_*) \leq f(\vec{x}), \tag{10.1}$$

for all $\vec{x} \in \Re^k$ satisfying the following property:

$$||\vec{x} - \vec{x}_*|| < \epsilon, \text{ where } ||.|| \text{ denotes any norm.}$$

Note that the definition does not hold for any ϵ, but says that there exists an ϵ that satisfies the condition above.

DEFINITION 10.6 *If the condition in (10.1) of the previous definition is satisfied with a strict inequality, i.e., if*

$$f(\vec{x}_*) < f(\vec{x}),$$

then $\vec{x}_*$ is called a strict local minimum of the function f.

When the definition is satisfied with a $\leq$ instead of $<$, $\vec{x}_*$ is still a local minimum, but it is not a strict local minimum.

If for every neighborhood of $\vec{x}$ that does not include $\vec{x}$, $f(\vec{x}_*) - f(\vec{x})$ assumes both negative and positive values, $\vec{x}_*$ is called a saddle point of the function f.

Definition 10.7 *A global minimum for a function $f : \Re^k \rightarrow \Re$ is a point (vector) which is no worse than all other points in the domain of the function; that is,*

$$f(\vec{x}_*) \leq f(\vec{x}), \textit{ for all } \vec{x} \in \Re^k.$$

This means that the global minimum is the minimum of all the local minima.

The definitions for local and global *maxima* can be similarly formulated. See Fig. 10.1 to get geometric intuition for strict local optima and saddle points. See Fig. 10.2 for an illustration of the difference between local and global optima.

2.5. Taylor's Theorem

The Taylor's theorem is a well-known result (see any elementary mathematics text, e.g., [181]). It shows that a function can be expressed as an infinite series involving derivatives if derivatives of

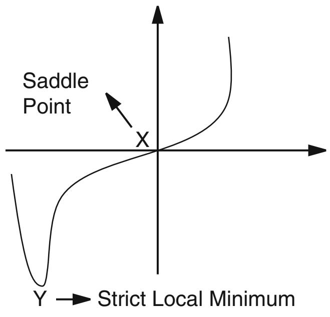

Figure 10.1. Strict local optima and saddle points in function minimization

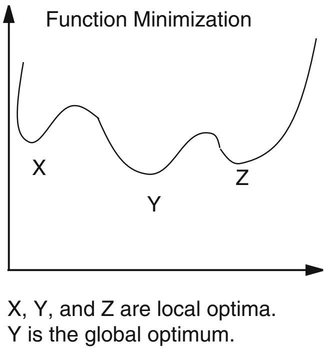

Figure 10.2. Local and global optima

all orders exist and if some other conditions hold. A function for which derivatives of all orders exist is also called a smooth function. We present this result without proof.

THEOREM 10.1 (**Taylor Series** with one variable) *A smooth function $f(x+h)$ can be expressed as follows:*

$$f(x+h) = f(x)+h\frac{df(x)}{dx}+\frac{(h)^2}{2!}\frac{d^2f(x)}{dx^2}+\ldots+\frac{(h)^n}{n!}\frac{d^nf(x)}{dx^n}+\ldots \quad (10.2)$$

We will now prove that a local minimum is a stationary point in the single variable case. The result can be easily extended to multiple variables.

THEOREM 10.2 *A local minimum x_* of a function $f(x)$ is a stationary point,*

$$\textit{that is } \left.\frac{df(x)}{dx}\right|_{x=x_*} = 0.$$

Proof If $|h|$ in the Taylor's series is a small quantity, one can ignore terms with h raised to 2 and higher values. Then setting $x = x_*$ in the Taylor's series, and selecting a sufficiently small value for $|h|$, one has that:

$$f(x_*+h) = f(x_*) + h\left.\frac{df(x)}{dx}\right|_{x=x_*}. \quad (10.3)$$

We will use contradiction logic. Let us assume that x_* is *not* a stationary point. Then,

$$\left.\frac{df(x)}{dx}\right|_{x=x_*} \neq 0, \text{ i.e.,}$$

$$\text{either } \left.\frac{df(x)}{dx}\right|_{x=x_*} > 0 \text{ or } \left.\frac{df(x)}{dx}\right|_{x=x_*} < 0.$$

In either case, by selecting a suitable sign for h, one can always have that:

$$h\left.\frac{df(x)}{dx}\right|_{x=x_*} < 0.$$

Using the above in Eq. (10.3) one has that

$$f(x_*+h) < f(x_*). \quad (10.4)$$

From the definition of a local minimum, we have that there exists an ϵ such that $f(x_* \pm \epsilon) \geq f(x_*)$. If the value of h that was selected above satisfies $|h| < \epsilon$, we have that: $f(x_* \pm |h|) \geq f(x_*)$. This implies that: $f(x_* + h) \geq f(x_*)$, which contradicts inequality (10.4). As a result, the local minimum must be stationary point. ▮

To prove that a stationary point is a local optimum, one has to establish some additional conditions, which we do not discuss, related to the derivatives of the function. It needs to be understood that a *stationary* point may be local or a global optimum. In other words, because a point is a local optimum, we have no guarantee that it is a global optimum. To ascertain whether a local optimum is also a global optimum, one needs to establish the so-called convexity properties for the function. Most of these conditions, including the convexity and derivative tests, are hard to verify in simulation-based optimization because the closed form of the function is unknown. As such, we will remain content with analysis related to the first partial derivative. We will be also assuming, somewhat arbitrarily, that the first partial derivative exists.

Our analysis of gradient (derivative) methods will be restricted to showing that the algorithms can reach *stationary* points of the function. We will have to hope that by using a multi-start approach, the algorithms are able to identify the *global* optima. In summary, it is very important to realize that if the function does *not* possess the first partial derivatives, the algorithm may not converge. Further, even if the first partial derivatives exist, we cannot be sure that the local optimum obtained is a global optimum unless we employ convexity arguments.

3. Steepest Descent

In this book, the principle of steepest descent was discussed in the context of neural networks and also simultaneous perturbation/finite differences. Hence, we now present some elementary analysis of this rule. We will prove that the steepest-descent rule converges to a stationary point of the function it seeks to optimize under certain conditions.

The main transformation in steepest descent is:

$$x^{m+1}(i) = x^m(i) - \mu \left.\frac{\partial f(\vec{x})}{\partial x(i)}\right|_{\vec{x}=\vec{x}^m} \quad \text{for } i = 1, 2, \ldots, k, \tag{10.5}$$

where $f : \Re^k \to \Re$, k denotes the number of decision variables (parameters), and μ is the value of the step size. We will need the following **column** vector in our analysis below.

$$\nabla f(\vec{x}) = \begin{bmatrix} \frac{\partial f(\vec{x})}{\partial x(1)} \\ \frac{\partial f(\vec{x})}{\partial x(2)} \\ \cdot \\ \cdot \\ \frac{\partial f(\vec{x})}{\partial x(k)} \end{bmatrix}. \tag{10.6}$$

We will frequently use the following notation instead of the above to save space:

$$\nabla f(\vec{x}) = \left[\frac{\partial f(\vec{x})}{\partial x(1)} \quad \frac{\partial f(\vec{x})}{\partial x(2)} \quad \cdots \quad \frac{\partial f(\vec{x})}{\partial x(k)} \right]^T.$$

We now present a result based on basic principles to show convergence of steepest descent when the step sizes are constant. It essentially shows that under certain conditions the algorithm converges to a point at which the gradient is zero, i.e., a stationary point.

THEOREM 10.3 *Let $\vec{x}^m$ denote the vector of values (of the parameters) in the mth iteration of the steepest-descent approach defined in Eq. (10.5). If the function f is continuously differentiable, is Lipschitz continuous, i.e.,*

$$||\nabla f(\vec{a_1}) - \nabla f(\vec{a_2})|| \leq L||\vec{a_1} - \vec{a_2}||, \qquad \forall\ \vec{a_1}, \vec{a_2}, \in \Re^k, \tag{10.7}$$

for some finite $L > 0$, and is bounded below, then for $\mu < 2/L$,

$$\lim_{m \to \infty} \nabla f(\vec{x}^m) = \vec{0}.$$

Proof In the proof, we will use the Euclidean norm . Hence, $||\cdot||$ will denote $||\cdot||_2$. Consider two vectors, $\vec{x}$ and $\vec{z}$, in $\Re^k$ and $\zeta \in \Re$. Let

$$g(\zeta) = f(\vec{x} + \zeta\vec{z}).$$

Then, from the chain rule, one has that:

$$\frac{dg(\zeta)}{d\zeta} = [\vec{z}]^T \nabla f(\vec{x} + \zeta\vec{z}). \tag{10.8}$$

We will use this below. We have

$$f(\vec{x} + \vec{z}) - f(\vec{x}) \quad = \quad g(1) - g(0) \text{ follows from the definition of } g(\zeta)$$

$$\begin{aligned}
&= \int_0^1 dg(\zeta) = \int_0^1 \frac{dg(\zeta)}{d\zeta} d\zeta \\
&= \int_0^1 [\vec{z}]^T \nabla f(\vec{x} + \zeta \vec{z}) d\zeta \text{ from (10.8)} \\
&\leq \int_0^1 [\vec{z}]^T \nabla f(\vec{x}) d\zeta + \left| \int_0^1 \vec{z}^T (\nabla f(\vec{x} + \zeta \vec{z}) - \nabla f(\vec{x})) d\zeta \right| \\
&\leq \int_0^1 [\vec{z}]^T \nabla f(\vec{x}) d\zeta + \int_0^1 ||\vec{z}|| \cdot ||\nabla f(\vec{x} + \zeta \vec{z}) - \nabla f(\vec{x})|| d\zeta \\
&\leq [\vec{z}]^T \nabla f(\vec{x}) \int_0^1 d\zeta + ||\vec{z}|| \int_0^1 L\zeta ||\vec{z}|| d\zeta \text{ from (10.7)} \\
&= [\vec{z}]^T \nabla f(\vec{x}) \int_0^1 d\zeta + L||\vec{z}||^2 \int_0^1 \zeta d\zeta \\
&= [\vec{z}]^T \nabla f(\vec{x}) \cdot 1 + L||\vec{z}||^2 \cdot \frac{1}{2}.
\end{aligned}$$

In the above, setting $\vec{x} = \vec{x}^m$ and $\vec{z} = -\mu \nabla f(\vec{x}^m)$, we obtain:

$$f(\vec{x}^m - \mu \nabla \vec{x}^m) - f(\vec{x}^m) \leq -\mu [\nabla f(\vec{x}^m)]^T \nabla f(\vec{x}^m) + \frac{1}{2} \mu^2 L ||\nabla f(\vec{x}^m)||^2.$$

The above can be written as:

$$\begin{aligned}
f(\vec{x}^m) - f(\vec{x}^m - \mu \nabla \vec{x}^m) &\geq \mu [\nabla f(\vec{x}^m)]^T \nabla f(\vec{x}^m) - \frac{1}{2} \mu^2 L ||\nabla f(\vec{x}^m)||^2 \\
&= \mu ||\nabla f(\vec{x}^m)||^2 - \frac{1}{2} \mu^2 L ||\nabla f(\vec{x}^m)||^2 \qquad (10.9) \\
&= \frac{\mu L}{2} \left(\frac{2}{L} - \mu \right) ||\nabla f(\vec{x}^m)||^2 \qquad (10.10) \\
&\geq 0 \text{ if } \mu < 2/L. \qquad (10.11)
\end{aligned}$$

Note: (10.9) follows from the fact that for the Euclidean norm

$$[\nabla f(\vec{x}^m)]^T \nabla f(\vec{x}^m) = ||\nabla f(\vec{x}^m)||^2.$$

From inequality (10.11), it follows that if $\mu < 2/L$,

$$f(\vec{x}^m) - f(\vec{x}^{m+1}) \geq 0$$

for any m. In other words, the values of the objective function $f(\vec{x}^m)$ for $m = 1, 2, 3, \ldots$ form a decreasing sequence. A decreasing sequence that is bounded below converges (see Theorem 9.2 from Chap. 9) to a finite number. Hence:

$$\lim_{m \to \infty} [f(\vec{x}^m) - f(\vec{x}^{m+1})] = 0.$$

From the above and (10.10), we have that

$$\lim_{m\to\infty} \frac{\mu L}{2}(\frac{2}{L} - \mu)||\nabla f(\vec{x}^m)||^2 \leq 0.$$

In the above, all the quantities in the left hand side are $\geq$ 0. Consequently,

$$\lim_{m\to\infty} \nabla f(\vec{x}^m) = \vec{0}. \quad \blacksquare$$

In order to show that every limit point of the sequence $\{\vec{x}^m\}_{m=1}^{\infty}$ is a stationary point of the function, one needs some additional conditions; also, under some other additional conditions, one can extend the above result to decreasing step sizes [30, Chap.3].

4. Finite Differences Perturbation Estimates

In this section, we present an important result which shows why the central difference formula (Eq. (5.2)) yields a more accurate estimate of the derivative than the forward difference formula (Eq. (5.3)). This result is not needed for showing convergence of simultaneous perturbation and can be skipped without loss of continuity.

THEOREM 10.4 *The forward difference formula (5.3) ignores terms of the order of h^2 and of higher orders, while the central difference formula (5.2) ignores terms of the order of h^3 and of higher orders, but not the terms of the order of h^2.*

Proof The forward difference formula assumes that all terms of the order of h^2 and higher orders of h are negligible. If h is small, this is a reasonable assumption, but it produces an error nevertheless. The central difference formula on the other hand does **not** neglect terms of the order of h^2. It neglects the terms of the order of h^3 and higher orders of h. From the Taylor Series, ignoring terms with h^2 and higher orders of h, we have that:

$$f(x+h) = f(x) + h\frac{df(x)}{dx}.$$

This, after re-arrangement of terms, yields:

$$\frac{df(x)}{dh} = \frac{f(x+h) - f(x)}{h},$$

which of course is the forward difference formula given in Eq. (5.3).

Now, from the Taylor series,

$$f(x-h) = f(x) - h\frac{df(x)}{dx} + \frac{(h)^2}{2!}\frac{d^2f(x)}{dx^2} - \cdots + (-1)^n\frac{(h)^n}{n!}\frac{d^nf(x)}{dx^n} + \ldots \tag{10.12}$$

Now if we ignore terms of the order of h^3 and higher in both Eqs. (10.2) and (10.12), then subtracting Eq. (10.2) from Eq. (10.12), we have that:

$$f(x+h) - f(x-h) = 2h\frac{df(x)}{dx},$$

which after re-arrangement yields

$$\frac{df(x)}{dx} = \frac{f(x+h) - f(x-h)}{2h}.$$

The above is the central differences formula of Eq. (5.2), which was obtained without ignoring terms of the order of h^2. ∎

5. Simultaneous Perturbation

Material in this section is devoted to a convergence analysis of simultaneous perturbation. We will discuss convergence of the algorithm under three progressively weaker sets of conditions. We first need to define some notation.

1. The shorthand notation $f(\vec{x}^m + \vec{h}^m)$ will be used to denote

$$f(x^m(1) + h^m(1), x^m(2) + h^m(2), \ldots, x^m(k) + h^m(k)).$$

2. $D^m(i)$ will denote the *true* value of the partial derivative of the function under consideration with respect to the ith variable at the mth iteration of the algorithm; the derivative will be calculated at $\vec{x} = \vec{x}^m$. Thus mathematically:

$$D^m(i) \equiv \left.\frac{\partial f(\vec{x})}{\partial x(i)}\right|_{\vec{x}=\vec{x}^m}. \tag{10.13}$$

3. $S_h^m(i)$ will denote the *simultaneous perturbation* estimate of the derivative in the mth iteration of the algorithm that uses $\vec{h}$ for perturbation. Mathematically, this estimate is defined as:

$$S_h^m(i) \equiv \frac{f(\vec{x}^m + \vec{h}^m) - f(\vec{x}^m - \vec{h}^m)}{2h^m(i)}. \tag{10.14}$$

The above assumes that **exact** values of the function are available in the computation above. In what follows, we will drop h from the subscript of S^m, but it will be understood that every simultaneous perturbation estimate will depend on the vector $\vec{h}$. Also, the vector $\vec{h}$ is computed using Eq. (5.4).

4. The update used in the simultaneous perturbation algorithm can be given by

$$x^{m+1}(i) = x^m(i) - \mu^m S^m(i) \text{ for } i = 1, 2, \ldots, k, \tag{10.15}$$

where $S^m(.)$ is defined in Eq. (10.14).

5. We define a set $\mathcal{K} = \{1, 2, \ldots, k\}$.

We now discuss how this section is organized. In Sect. 5.1, we will exploit a powerful result related to stochastic gradients. In Sect. 5.2, we will use an approach based on ODEs (ordinary differential equations) to show convergence. Finally, in Sect. 5.3, we will state the conditions used in Spall [280, 281] to establish convergence.

5.1. Stochastic Gradient

We will first consider a *stochastic* gradient algorithm of which simultaneous perturbation is a special case. Consider the stochastic gradient algorithm usually defined as:

$$x^{m+1}(i) = x^m(i) - \mu^m \left[\left.\frac{\partial f(\vec{x})}{\partial x(i)}\right|_{\vec{x}=\vec{x}^m} + w^m(i) \right] \text{ for } i = 1, 2, \ldots, k \tag{10.16}$$

where k is the number of decision variables, μ^m is the value of the step size in the mth iteration, and $w^m(i)$ is a noise term. Note that this algorithm is called the *stochastic* gradient algorithm because of the noise present in what is otherwise the update of the steepest-descent algorithm. We will now impose some conditions on this algorithm.

Assumption 10.5 *The function $f : \Re^k \to \Re$ satisfies the following conditions:*

- *$f(\vec{x}) \geq 0$ everywhere.*
- *$f(\vec{x})$ is continuously differentiable, and the function $\nabla f(\vec{x})$ is Lipschitz continuous , i.e.,*

$$||\nabla f(\vec{x_1}) - \nabla f(\vec{x_2})|| \leq L||\vec{x_1} - \vec{x_2}||, \qquad \forall\ \vec{x_1}, \vec{x_2}, \in \Re^k.$$

ASSUMPTION 10.6 *Let the step size satisfy the following conditions:*

$$\lim_{l\to\infty}\sum_{m=1}^{l}\mu^m=\infty;\ \lim_{l\to\infty}\sum_{m=1}^{l}(\mu^m)^2<\infty. \tag{10.17}$$

We now define the "history" of the algorithm up to and including the mth iteration by the set:

$$\mathcal{F}^m=\left\{\vec{x}^{\,0},\vec{x}^{\,1},\ldots,\vec{x}^{\,m},\vec{D}_s^0,\vec{D}_s^1,\ldots,\vec{D}_s^m,\mu^0,\mu^1,\ldots,\mu^m\right\}.$$

The history allows us to impose some further conditions on the noise in the algorithm.

ASSUMPTION 10.7 *For some scalars A and B*

$$E\left[w^m(i)|\mathcal{F}^m\right]=0 \text{ for every } i \tag{10.18}$$

$$\text{and } E\left[||\vec{w}^{\,m}||^2|\mathcal{F}^m\right]\le A+B||\nabla f(\vec{x}^{\,m})||^2 \tag{10.19}$$

We now present a key result without proof.

THEOREM 10.8 *Consider the algorithm defined in Eq. (10.16). If Assumptions 10.5—10.7 hold, then, with probability 1,*

R1. *The sequence $\{f(\vec{x}^{\,m})\}_{m=1}^{\infty}$ converges.*

R2. $\lim_{m\to\infty}\nabla f(\vec{x}^{\,m})=0$.

The condition **R2** essentially implies that the algorithm will converge with probability 1 to a stationary point, i.e., to a point at which the gradient will be 0. A more general version of this result can be found in [33, Prop 4.1; pg 141].

In order to show convergence via Theorem 10.8, we need to prove that the conditions imposed in Theorem 10.8 hold for simultaneous perturbation. One of these assumptions is Assumption 10.7 which will be shown to hold via Lemma 10.11. For the latter, we need the following elementary result.

THEOREM 10.9 (**Taylor Series** for a function of two variables: $x(1)$ and $x(2)$) *A smooth function $f(x(1)+h(1),x(2)+h(2))$ (infinitely differentiable in both variables) can be expressed as follows if $\vec{x}=(x(1),x(2))$.*

$$f(x(1)+h(1),x(2)+h(2))=f(x(1),x(2))+h(1)\frac{\partial f(\vec{x})}{\partial x(1)}+h(2)\frac{\partial f(\vec{x})}{\partial x(2)}$$

$$+\frac{1}{2!}\left[[h(1)]^2\frac{\partial^2 f(\vec{x})}{\partial x^2(1)}+2h(1)h(2)\frac{\partial^2 f(\vec{x})}{\partial x(1)\partial x(2)}+[h(2)]^2\frac{\partial^2 f(\vec{x})}{\partial x^2(2)}\right]+\ldots \tag{10.20}$$

The proof of the above can be found in any standard calculus text.

We are now at a position to prove Lemma 10.11 from [108].

Assumption 10.10 *The function f is smooth (i.e., infinitely differentiable).*

Lemma 10.11 *If Assumption 10.10 is true of the function defined in the update in Eq. (10.15), the noise terms in the update satisfy Assumption 10.7.*

Proof We will use the Euclidean norm below, i.e., $||.||$ will mean $||.||_2$. The proof's road map is as follows: First a relationship is developed between the simultaneous perturbation estimate and the true derivative, which helps define the noise—that will be shown to satisfy Assumption 10.7.

We will assume for the time being that $k = 2$. From the Taylor series result, i.e., Eq. (10.20), ignoring terms with h^3 and higher orders of h and suppressing the superscript m, we have that:

$$f(x(1)+h(1),x(2)+h(2)) = f(x(1),x(2))+h(1)\frac{\partial f(\vec{x})}{\partial x(1)}+h(2)\frac{\partial f(\vec{x})}{\partial x(2)}+$$
$$\frac{1}{2!}\left[[h(1)]^2\frac{\partial^2 f(\vec{x})}{\partial x^2(1)}+2h(1)h(2)\frac{\partial^2 f(\vec{x})}{\partial x(1)\partial x(2)}+[h(2)]^2\frac{\partial^2 f(\vec{x})}{\partial x^2(2)}\right] \tag{10.21}$$

From the same Taylor series, we also have that:

$$f(x(1)-h(1),x(2)-h(2)) = f(x(1),x(2))-h(1)\frac{\partial f(\vec{x})}{\partial x(1)}-h(2)\frac{\partial f(\vec{x})}{\partial x(2)}+$$
$$+\frac{1}{2!}\left[[h(1)]^2\frac{\partial^2 f(\vec{x})}{\partial x^2(1)}+2h(1)h(2)\frac{\partial^2 f(\vec{x})}{\partial x(1)\partial x(2)}+[h(2)]^2\frac{\partial^2 f(\vec{x})}{\partial x^2(2)}\right]. \tag{10.22}$$

Subtracting Eq. (10.22) from Eq. (10.21), we have

$$f(x(1)+h(1),x(2)+h(2)) - f(x(1)-h(1),x(2)-h(2)) =$$
$$2h(1)\frac{\partial f}{\partial x(1)}+2h(2)\frac{\partial f(\vec{x})}{\partial x(2)}.$$

From the above, by re-arranging terms, we have:

$$\frac{f(x(1)+h(1),x(2)+h(2)) - f(x(1)-h(1),x(2)-h(2))}{2h(1)} =$$

$$\frac{\partial f(\vec{x})}{\partial x(1)} + \frac{h(2)}{h(1)}\frac{\partial f(\vec{x})}{\partial x(2)}.$$

Noting the fact that we had suppressed m in the superscript and from the definitions of $S^m(i)$ and $D^m(i)$, the above can be written as

$$S^m(1) = D^m(1) + \frac{h^m(2)}{h^m(1)}\frac{\partial f(\vec{x})}{\partial x(2)}. \tag{10.23}$$

Equation (10.23), using the k-variable version of Eq. (10.20), can be generalized to:

$$S^m(i) = D^m(i) + \sum_{j\neq i; j=1}^{k} \frac{h^m(j)}{h^m(i)}\frac{\partial f(\vec{x})}{\partial x(j)} \text{ for every } i \in \mathcal{K}. \tag{10.24}$$

If we define the noise in simultaneous perturbation as

$$w^m(i) = S^m(i) - D^m(i) \text{ for every } i \in \mathcal{K}, \tag{10.25}$$

we have that the simultaneous perturbation update in Eq. (10.15) is of the stochastic descent update defined in Eq. (10.16).

The definition of noise in Eq. (10.25) implies that

$$w^m(i) = \sum_{j\neq i; j=1}^{k} \frac{h^m(j)}{h^m(i)}\frac{\partial f(\vec{x})}{\partial x(j)} \text{ for every } i \in \mathcal{K}. \tag{10.26}$$

Let us also define the set of the history of the algorithm up to and including the mth iteration by:

$$\mathcal{F}^m = \{\vec{x}^{\,0}, \vec{x}^{\,1}, \ldots, \vec{x}^{\,m}, \vec{S}^0, \vec{S}^1, \ldots, \vec{S}^m, \mu^0, \mu^1, \ldots, \mu^m\}.$$

From Eq. (5.4), we know that for any (i,j) pair where $i \in \mathcal{K}$ and $j \in \mathcal{K}$,

$$\frac{h^m(j)}{h^m(i)} = \frac{H^m(j)}{H^m(i)}.$$

Now, if the history of the algorithm is known, from the Bernoulli distribution used in computing $\vec{H}$ (see algorithm description in Chap. 5 and Eq. (5.4)), it follows that for any given (i,j) pair, where $i \in \mathcal{K}$ and $j \in \mathcal{K}$,

$$\begin{aligned}
\mathsf{E}\left[\frac{h^m(j)}{h^m(i)}\frac{\partial f(\vec{x})}{\partial x(j)}\middle|\, \mathcal{F}^m\right] &= \mathsf{E}\left[\frac{H^m(j)}{H^m(i)}\frac{\partial f(\vec{x})}{\partial x(j)}\middle|\, \mathcal{F}^m\right] \\
&= [(0.5)(-1)+(0.5)(1)]\mathsf{E}\left[\frac{1}{H^m(i)}\frac{\partial f(\vec{x})}{\partial x(j)}\middle|\, \mathcal{F}^m\right] \\
&= 0.
\end{aligned}$$

From the above equation and from Eq. (10.26), it follows that for every $i \in \mathcal{K}$,

$$\mathsf{E}[w^m(i)|\mathcal{F}^m] = \sum_{j\neq i;j=1}^{k} 0 = 0,$$

thereby proving Eq. (10.18) in Assumption 10.7. Condition (10.19) in the same assumption will now be shown.

Remember from the algorithm description in Chap. 5 that for any (i, j) pair, where $i \in \mathcal{K}$ and $j \in \mathcal{K}$,

$$\frac{h(j)}{h(i)} = 1 \text{ or } -1.$$

Hence for any $i, j \in \mathcal{K}$,

$$\left[\frac{h(j)}{h(i)}\right]^2 = 1 \text{ and } \left|\frac{h(j)}{h(i)}\right| = 1. \qquad (10.27)$$

Then, it can be shown that:

$$\begin{aligned} ||\vec{w}^m||^2 &= [w^m(1)]^2 + [w^m(2)]^2 + \cdots + [w^m(k)]^2 \\ &= \sum_{i=1}^{k}\left(\sum_{j\neq i;j=1}^{k} \frac{h^m(j)}{h^m(i)}\frac{\partial f(\vec{x})}{\partial x(j)}\right)^2 \\ &= (k-1)||\nabla f(\vec{x}^m)||^2 + A. \end{aligned}$$

The above follows from noting that A represents the sum of products within the square of each $w^m(.)$; the above also employs the Euclidean norm and exploits (10.27) and the fact that each of the partial derivatives is bounded. ∎

We note that in the proof above, the noise did *not* represent the noise induced by simulation; rather it is the noise in the derivative due to Spall's formula. Remember that Spall's formula does not compute the exact derivative. We now show the convergence of simultaneous perturbation.

THEOREM 10.12 *The simultaneous perturbation algorithm converges to a stationary point of the objective function, $f(\vec{x})$, with probability 1 if (i) Assumption 10.10 is true and (ii) Assumptions 10.5 and 10.6 are true.*

Proof This follows from a combination of Theorem 10.8 and Lemma 10.11. ∎

Note that smoothness, continuous differentiability, and Lipschitz continuity of the first derivative are conditions that can be only shown when the closed form of the function is available. This is rarely the case in simulation-based optimization. In fact, simulation-based optimization is used when the closed form is not available; when the closed form is available, simulation optimization is rarely the preferred method. Hence Spall refers to showing these conditions as an "abstract ideal" [281]. Nonetheless, it is important to know if there are certain conditions under which the algorithm can converge. The first condition in Assumption 10.5, which allows the function to take on only non-negative values, can certainly be verified. The condition on the step sizes (Assumption 10.6) can be shown to be true for a large number of step-size rules (see Spall [281] for a detailed discussion). A simple rule that satisfies these conditions is $A/(B+m)$ where A and B are non-negative scalars.

In the next subsection, we will show convergence under a slightly weaker set of conditions, which do not require that the function assume only non-negative values.

5.2. ODE Approach

We will now show convergence under conditions milder than the ones imposed in the previous subsection on the function. In particular, we note that we will not need to assume that the function f has to be non-negative everywhere, which is a rather strong condition that may not hold generally. We will use the result on ODEs (ordinary differential equations) presented as Theorem 9.16 in the previous chapter. We make the following assumptions on the function $f(.)$ used in the update defined in Eq. (10.15).

But, first remember that in the context of ODEs, we represent the iterates using upper-case letters (X) and the continuous-valued process underlying them by the lower-case letter (x). Thus, the update will be represented using the following notation here:

$$X^{m+1}(i) = X^m(i) - \mu^m S_h^m(i) \text{ for } i = 1, 2, \ldots, k, \text{ where } S_h^m(.)$$

(defined in Eq. (10.14)), using our uppercase notation, is:

$$S_h^m(i) \equiv \frac{f(\vec{X}^m + \vec{h}^m) - f(\vec{X}^m - \vec{h}^m)}{2h^m(i)}; \text{ see algorithm for definition of } \vec{h}^m.$$

Assumption 10.13 *The function $\nabla f(.)$ is Lipschitz continuous.*

Assumption 10.14 *The function $f(.)$ is Lipschitz continuous.*

We now consider the ODE defined as follows:

$$\frac{d\vec{x}}{dt} = F(\vec{x}).$$

If we define $F(\vec{x})$ for Theorem 9.16 as

$$F(\vec{x}) = \nabla f(\vec{x}),$$

then any local minimum of the function $f(.)$ is an equilibrium point for the ODE since the local minimum is defined as

$$\nabla f(\vec{x}) = \vec{0}, \text{ while an equilibrium point is } \frac{d\vec{x}}{dt} = 0.$$

We now make two additional assumptions.

Assumption 10.15 *A local minimum of the function $f(\vec{x})$ is a unique asymptotically stable equilibrium point for the ODE*

$$\frac{d\vec{x}}{dt} = F(\vec{x}) = \nabla f(\vec{x}).$$

Assumption 10.16 *The iterate $\vec{X}^m$ remains bounded with probability 1.*

We now state convergence under conditions weaker than those needed in the previous subsection.

Theorem 10.17 *The simultaneous perturbation algorithm converges to a local minimum of the objective function, $f(.)$, with probability 1, if Assumptions 10.6, 10.10, and 10.13–10.16 are true.*

For the proof, we will use Theorem 9.16 from Chap. 9. The reader should review this result at this point.

Proof In order to invoke Theorem 9.16, we must show that conditions required for convergence in Theorem 9.16 hold here. Assumption 10.6 $\equiv$ Assumption 9.13; Assumption 10.13 implies that Assumption 9.11 is true for $F(.) = \nabla f(.)$; Assumption 10.15 $\equiv$ Assumption 9.15; Assumption 10.16 $\equiv$ Assumption 9.14.

We now show that Assumption 9.12 also holds. From Lemma 10.11, Assumption 10.10 implies that condition (10.18) of Assumption 10.7 must hold, i.e., the first condition of Assumption 9.12 must hold. From Assumption 10.14, we have that the derivative of $f(.)$ is bounded, and hence the conditional mean of the square of the noise in (10.19) must be bounded. Since the iterates are also bounded (by Assumption 10.16), the second condition (see (9.22)) in Assumption 9.12 must hold. Then, the result follows from Theorem 9.16. ∎

5.3. Spall's Conditions

Spall [280] showed convergence of simultaneous perturbation under conditions weaker than those described above. While we do not present all the details of his ODE-based convergence analysis and refer the interested reader to the text [281], we note the following: Instead of the strong Assumption 10.10, he needs only the following weaker condition:

ASSUMPTION 10.18 *The function $f(.)$ is bounded and is three times continuously differentiable.*

Some of the other conditions needed in [280] are Assumptions 10.6, 10.15, and 10.16.

6. Stochastic Adaptive Search

The convergence theory for many Stochastic Adaptive Search (SAS) techniques relies on Markov chains. The reader not familiar with Markov chains is hence advised to read material in Chap. 6 until and including Sect. 3.1. In particular, we will need concepts related to transition probabilities and properties such as recurrence, ergodicity, and regularity. We now present some fundamental concepts that will be needed throughout this section.

Underlying Markov chain. We use i and j to denote indices of solutions encountered by the technique (algorithm). Let $P(i, j)$ denote the probability that the next solution be j when the current solution is i. If this probability depends only on i, then clearly we have a Markov chain underlying the algorithm's transitions, i.e., if we think of a solution as a state in the Markov chain, the trajectory of the solutions generated by the algorithm can be viewed as a Markov chain. $P(i, j)$, the element in the ith row and jth column of $\mathbf{P}$, will then denote the (one-step) transition probability of the underlying Markov chain.

In many SAS algorithms described in Chap. 5, we have used the notion of the generator matrix, $\mathbf{G}$, where $G(i, j)$ denotes the probability that a solution indexed by j will be generated as a candidate for the solution in the next iteration given that the solution in the current iteration is indexed by i. Of course, a candidate may or may not be accepted, depending on how the algorithm works. (If the current solution is generated as a candidate, it is accepted by default.) This idea can be mathematically captured by the notion of *acceptance probability*, $A(i, j)$, which denotes the probability that a candidate solution indexed by j is accepted by the algorithm provided the current solution is the solution indexed by i. This leads to the following model for the transition probability of the underlying Markov chain:

$$P(i,j) = \begin{cases} G(i,j)A(i,j) \text{ when } i \neq j; \\ G(i,i)A(i,i) + \sum_{i \neq j} G(i,j)(1 - A(i,j)), \end{cases} \tag{10.28}$$

where $A(i,i) = 1$. Note that in the above, the expression for $P(i,j)$ with $i \neq j$ follows from elementary product rule of probabilities, while the same for $P(i,i)$ follows from the fact that the algorithm remains in the same solution either if it is generated again and accepted, or if some other solution is generated but rejected. Note that $A(i,i)$ must equal 1 for the probabilities to be well-defined.

In some SAS algorithms, we will have a unique Markov chain that will define its behavior. In some algorithms, however, the Markov chain will change with every iteration or after a few iterations. When the Markov chain does not change with iterations, it will be called a *stationary* or *homogenous* Markov chain, while if it changes with iterations, it will be called a *non-stationary* or *non-homogenous* Markov chain. These ideas will be clarified further in the context of every algorithm that we will analyze.

Convergence metrics. To analyze the convergence of any SAS technique, we will be interested in answering the following three questions:

1 Does the algorithm reach the global optimum in the limit (*asymptotically*)?

2 How many iterations are needed to *first* strike or hit the optimal solution?

3 Does the algorithm settle down into the optimal solution in the limit?

The first question is related to whether the algorithm will ever reach the global optimum and is the most fundamental of questions we ask of any optimization algorithm. The second question revolves around how long the *first hitting time* will be. The last question is regarding whether the algorithm will eventually be absorbed into the global optimum.

For some algorithms, as we will see below, it is possible to answer *all three* questions. But, for any SAS technique, we would like to have at the very least an answer to the first question. Answers to the second question may provide information about the *rate of convergence* of the algorithm. The reader should note that an affirmative response to the first question does not imply that the algorithm will take fewer iterations than exhaustive search. All it ensures is that *eventually*, we will visit the global optimum. But there is nothing specified about how long that might take! Hence, one must also answer the second question and hopefully establish that the rate of convergence is faster than that of exhaustive search. Finally, an affirmative answer to the third question assures us that in the limit the algorithm in some sense *converges* to the optimal solution.

It should be clear now that if the algorithm converges to the optimal solution in the limit (answer to the third question), then the answer to the first question is yes. However, producing an answer to the second question is more important from the practical standpoint since it gives us an idea of how long an algorithm may take before identifying an optimal solution. Oftentimes answering this question is the hardest task.

We also note that the number of iterations to first strike/hit the optimal is likely to be a random variable in an SAS technique. Therefore, one is interested in the mean and also in the variance (and possibly higher order moments) of this number. Finally, we note that in algorithms where the "best solution thus far" is maintained, the number of iterations to first strike the optimal is sufficient to measure the rate of convergence. But in algorithms like the nested partitions and stochastic ruler, where one does not keep this in memory, one may have to use other ways to measure the rate of convergence. We will explore answers to these questions for some stochastic search algorithms now. We begin with pure random search where our discussion follows [333].

6.1. Pure Random Search

Let m denote the iteration number in pure random search and let $\mathcal{X}$ denote the set of solutions in the problem. We now state the following result.

THEOREM 10.19 *Pure random search strikes the global optimum almost surely as $m \to \infty$.*

Proof Let $q(j)$, where $q(j) > 0$ for every j, denote the probability of selecting the solution indexed by j such that $\sum_{j=1}^{|\mathcal{X}|} q(j) = 1$. Then the transitions from one solution to another in the algorithm can be modeled by a stationary Markov chain whose transition probabilities are defined as: $P(i, j) = q(j)$ for every i. Clearly then, the Markov chain is regular, and hence ergodic. Hence, every state, including the global optimal solution, will be visited infinitely often in the limit, and we are done. ∎

We now present a result (see e.g., [333]) that computes the mean and variance of the number of iterations needed for the first strike at the global optimum for any pure random search.

THEOREM 10.20 *Let q^* denote the probability of selecting the global optimum, and let M denote the number of iterations needed to first strike the global optimum. For pure random search,*

$$\mathsf{E}[M] = \frac{1}{q^*} \text{ and } \mathsf{Var}[M] = \frac{1-q^*}{(q^*)^2}.$$

Proof If we associate "success" with striking the global optimum, M will have the geometric distribution (see Appendix), whose probability of success is q^*. The result follows from the expressions for the mean and variance of the geometric distribution. ∎

The implication is that for pure random search, we can expect the algorithm to first strike the global optimal in an average of $1/q^*$ iterations. Note that typically since the structure of the problem is unknown, one uses the algorithm in which every solution is selected with the same probability. (If the structure is known, one may design the algorithm such that the probability of selecting solutions in the area where the global optimum is likely to be is higher than $1/|\mathcal{X}|$.) Thus, if we have 1,000 solutions, $q^* = 1/1{,}000 = 0.001$, i.e., we will strike the global optimal in an average of 1,000 iterations. If we perform an exhaustive search instead, we are guaranteed to reach the optimal in 1,000 iterations. In fact, with pure random search, we reach the global optimum in *an average of* 1,000 iterations and the actual value may be much larger since the variance is $(1-0.001)/(0.001)^2 = 999{,}000$. In other words, there is a large amount of variability in the performance of this algorithm. Clearly, the expectation and variance will be smaller for smaller problems, but for small problems, we are better served by exhaustive enumeration.

Finally, we note that the algorithm will not be absorbed into any state (solution), since we have a regular Markov chain. Its limiting probabilities ($q(.)$) will satisfy $q(j) > 0$ for every state j. Hence, the answer to the third question will be in the negative.

One reason for presenting the results above was to demonstrate the limitations of pure random search. The results also provide us with some insights on how stochastic search techniques work if they fail to adapt. As should be clear by now, the performance of a stochastic search technique can be rather discouraging if it fails to adapt. Also very importantly, Theorem 10.20 provides us with a benchmark with which to compare (theoretically or empirically) the performance of any other (existing or newly developed) SAS algorithm. Obviously, any algorithm that does not require fewer (average) iterations than pure random search will be of questionable value, since pure random search is usually easier to program and execute than most algorithms. And as we saw above, pure random search is itself of questionable value in discrete optimization.

Before moving on to the next technique, we relate the transition probabilities to the generator matrix of pure random search. For pure random search, the two matrices are identical, i.e., $P(i,j) = G(i,j)$ for every (i,j) pair.

6.2. Learning Automata Search Technique

We now present a result in regards to the convergence of LAST (Learning Automata Search Technique). Unlike pure random search, LAST is an adaptive technique where one hopes that the algorithm will eventually settle down on the global optimum. Before presenting a result along those lines, we discuss its underlying Markov chain.

For any given iteration, we can define the transition probabilities as follows. Let $q^m(l)$ be the probability of selecting a solution indexed by l in the mth iteration. (In describing the algorithm, we used $p^m(i,a)$ to denote the probability of selecting the value a for the ith decision variable. Clearly then, if we set $l \equiv (i,a)$, $p^m(i,a) = q^m(l)$.) For the mth iteration, we then have a Markov chain whose transition probabilities are defined by $P_m(r,l) = q^m(l)$ for every r. But the probabilities $q^m(l)$ change with every iteration, and thus we have a new transition probability matrix $P_{m+1}(r,l) = q^{m+1}(l)$ during iteration $m+1$. A Markov chain of this kind is called a non-stationary Markov chain, which is usually harder to analyze than its stationary counterpart. We can relate the transition probability matric to the generator matrix: $G_m(r,l) \equiv P_m(r,l)$, where $G_m(r,l)$ denotes the generator matrix for the mth iteration.

We now present a convergence result for LAST from [298] without proof. Recall that $p^m(i,a)$ was defined as the probability of selecting the ath value for the ith decision variable in the mth iteration. Let $a^*(i)$ denote the optimal value for the ith decision variable. Further recall that μ was defined as the step size (learning rate) in LAST for all the decision variables. For the convergence result, we consider a more general case where the step size may in fact be different for each of the k decision variables. The step size for the ith decision variable will hence be defined as $\mu(i)$ for $i = 1, 2, \ldots k$.

THEOREM 10.21 [298] *Assume that the discrete optimization problem has a unique global optimum, which is denoted by $x(i, a^*(i))$ for every $i = 1, 2, \ldots k$. Then for any $\epsilon > 0$ and any $\delta > 0$, there exists a scalar $\mu^* > 0$ and an integer $M_0 < \infty$ such that for every $i = 1, 2, \ldots k$ and for every $m > M_0$,*

$$\mathsf{Pr}\left(|p^m(i, a^*(i)) - 1| < \epsilon\right) > (1 - \delta) \text{ if } \mu(i) \in (0, \mu^*).$$

In other words, there exists a finite value, M_0, for the number of iterations such that if the algorithm is run for more than M_0 iterations, for any given $\delta > 0$, the probability defined by $\mathsf{Pr}(|p^m(i, a^*(i)) - 1| < \epsilon)$ will be greater than $(1 - \delta)$. Clearly by choosing a sufficiently small value for δ, we can ensure that this probability will be sufficiently close to 1. Since the probability is that of the algorithm selecting the optimal value for a decision variable, the result essentially implies that by choosing sufficiently small values for each $\mu(i)$, LAST can determine the global optimum with an arbitrarily high probability. The above result also implies that eventually the algorithm will converge to the global optimum (third question). Note however that the result does not say anything about the number of iterations needed for the first strike at the global optimal.

6.3. Backtracking Adaptive Search

We now present a convergence analysis of BAS (Backtracking Adaptive Search) under a structural assumption on the generator matrix. The reader interested in a more comprehensive analysis under a more general assumption is referred to the original work in [333]. The motivation for presenting our analysis is to highlight the basic structure of the Markov chain underlying it. Towards the end of this subsection, we will comment on how this analysis in many ways unifies the theory of some SAS techniques.

The transition probabilities, the acceptance probabilities, and the elements of the generator matrix will follow the model defined in Eq. (10.28). The acceptance probabilities are defined as shown in Eq. (5.10). We make the following structural assumption about the generator matrix.

ASSUMPTION 10.22 *Every element of the generator matrix, $\mathbf{G}$, is non-zero.*

THEOREM 10.23 *Under Assumption 10.22, BAS converges to the global optimum almost surely as $m \to \infty$.*

Proof From Eq. (5.10), we have that $A(i,j) > 0$ for every (i,j)-pair. Hence, $P(i,j) > 0$ for every (i,j)-pair, which implies that $\mathbf{P}$ is regular, and hence ergodic. Then, every state can be visited from every other state, which implies that regardless of where the algorithm starts, it is almost surely guaranteed to visit the global optimum eventually. ∎

The above line of analysis is similar to that used for pure random search. However, one should note that on the average, BAS, unlike pure random search, seeks improving points. We now present a result that computes the mean number of iterations needed for the first strike at the global optimum. We first need an elementary result from Markov chain theory.

LEMMA 10.24 *Consider an ergodic Markov chain whose transition probability matrix is defined by $\mathbf{P}$. Let $\mathsf{E}[M(i)]$ denote the expected number of transitions needed to first reach a distinguished state i_1 in the chain given that the initial state is i. Then, we have the following linear system of equations:*

$$\mathsf{E}[M(i)] = 1 + \sum_{j \neq i_1} P(i,j)\mathsf{E}[M(j)] \text{ for every } i \text{ in the chain.}$$

Proof The result can be found in any text on Markov chains, e.g., [251]. ∎

Let i^* denote the index of the global optimum from the finite set of solutions denoted by $\mathcal{X}$.

THEOREM 10.25 *Let $M(i)$ denote the number of iterations needed for the first strike at the global optimum provided the algorithm starts at solution i. Under Assumption 10.22, we have that the mean number of iterations for the first strike can be computed by solving the following system of linear equations. For $i = 1, 2, \ldots, |\mathcal{X}|$,*

$$\mathsf{E}[M(i)] = 1 + \sum_{\substack{j \in \mathcal{X} \\ j \neq i^*}} P(i,j)\mathsf{E}[M(j)].$$

Proof Since $\mathbf{P}$ is a regular Markov chain, it is ergodic, and hence the result is immediate from Lemma 10.24. ∎

The above provides a mechanism to compute the mean number of iterations needed for the first strike at the global optimum provided we start at any state. It does need the elements of the underlying transition probability matrix, however, and those values may be rather difficult to obtain for large problems. Nonetheless, we are assured of convergence almost surely in a finite number of iterations. In practice, one may wish to compare the average number of iterations needed with the corresponding number for pure random search.

Under conditions weaker than Assumption 10.22, we may not have an underlying ergodic Markov chain. Under weaker conditions, one can still show convergence to a global optimum, via the remarkable analysis in [182, 333]. This analysis is beyond our scope here, but we explain their main ideas. Consider the trajectory of solutions pursued by the algorithm. Every time the algorithm accepts a non-improving point, one should assume that a new sub-trajectory begins. Thus, the actual trajectory is divided into sub-trajectories such that within in each sub-trajectory only improving points are accepted. Then, associated with each sub-trajectory of the algorithm, one can construct a Markov chain with an absorbing state. The algorithm can then be viewed as one that progresses through a sequence of Markov chains, which can be analyzed to (i) show that the system is eventually absorbed into the global optimum and (ii) determine the mean and variance of the number of iterations needed for the first strike at the global optimum.

The reader also needs to note that with Assumption 10.22, the algorithm is a form of pure random search in which improving points are accepted with a high probability. Further, simulated annealing in which the temperature, T, depends only on the function values at the current and the next iteration can also be described with BAS. In this sense, BAS is a remarkable SAS algorithm because its convergence theory provides insights on that of random search in general and a specific form of simulated annealing.

6.4. Simulated Annealing

Recall that we studied two categories of algorithms in simulated annealing: one used a fixed temperature within a phase which consisted of multiple iterations and the other changed the temperature after every iteration (i.e., the phase consisted of only one iteration). It turns out that for a given temperature, one can construct a stationary Markov chain. However, as soon as we change the temperature, we have a new Markov chain. The second category in which the Markov chain changes in every iteration can be modeled as a non-stationary

Markov chain. We will analyze the first category, where the Markov chain is stationary, in detail and provide references for the analysis of the second category to the interested reader.

We begin with a simple result, often called the *time reversibility* result of a Markov chain, that we will need later.

LEMMA 10.26 *Consider a regular Markov chain. Let the transition probability of transitioning from state i to state j in the chain be denoted by $P(i,j)$. If a vector $\vec{y}$ satisfies the following relations:*

$$\sum_i y(i) = 1 \quad \textit{and} \tag{10.29}$$

$$y(i)P(i,j) = y(j)P(j,i) \tag{10.30}$$

for all (i,j)-pairs, then $\vec{y}$ is the limiting (steady-state) probability vector of the Markov chain.

Proof Equation (10.30), when summed over i, yields for any j:

$$\sum_i y(i)P(i,j) = y(j)\sum_i P(j,i) = y(j)1 = y(j),$$

which together with Eq. (10.29) implies from elementary Markov chain theory that $\vec{y}$ *is* the limiting probability vector of the Markov chain. ∎

We now introduce the notion of symmetric neighborhood generation. Recall that neighbors are generated using generator matrices. We assume that a neighborhood generation scheme is *symmetric* when the following is true. If a point $\vec{y}$ is a neighbor of a point $\vec{x}$, then $\vec{x}$ must be an *equally likely* neighbor of $\vec{y}$. This implies that if the algorithm generates $\vec{y}$ as a neighbor of $\vec{x}$ with a probability of q, then $\vec{x}$ is generated as a neighbor of $\vec{y}$ with an **equal** probability (q). We will state this assumption formally now.

ASSUMPTION 10.27 *The generator matrix,* **G**, *is symmetric.*

We will need two additional assumptions for our convergence proof: Assumption 10.22 and

ASSUMPTION 10.28 *The global optimum is unique.*

We now present the main result related to the convergence of simulated annealing from [193] in which the temperature is fixed for multiple iterations within a phase. (We note that the proof in [193] assumes that the matrix, **G**, is irreducible, which is a weaker condition than that imposed by our Assumption 10.22.) We will assume that we are minimizing the objective function.

THEOREM 10.29 *Let $\vec{x}^{(m,T_m)}$ denote the vector of decision variables in the mth iteration of the phase in which the temperature is T_m. Let $\vec{x}_*$ denote the optimal solution. Then, under Assumptions 10.22, 10.27, and 10.28, almost surely:*

$$\lim_{T \downarrow 0} \left[\lim_{\substack{m\to\infty \\ T_m \equiv T}} \vec{x}^{(m,T_m)} = \vec{x}_* \right].$$

Proof We will first show that for a fixed temperature, a regular Markov chain can be associated the algorithm and then develop an expression for the limiting probability (steady-state) vector that involves the invariance equations. Thereafter, we will show that as T tends to 0, an absorbing Markov chain, whose absorbing state is the global optimum, results in the limit.

Assume the temperature to be fixed at T. The associated transition probability of going from i to j will be denoted by $P_T(i,j)$. Let $\mathcal{X}$ denote the set of states (solutions). We assume that the states are numbered as follows: $1, 2, 3, \ldots, |\mathcal{X}|$ such that $i < j$ if $f(\vec{x}_i) < f(\vec{x}_j)$. Using this ordering,

$$f(\vec{x}_1) = f(\vec{x}_*).$$

When in solution $\vec{x}_i$, the algorithm generates a neighbor $\vec{x}_j$ with probability $G(i,j)$. From Assumption 10.27 (symmetric neighborhoods), we have that

$$G(i,j) = G(j,i) \text{ for all } i, j.$$

The probability of accepting a neighbor $\vec{x}_j$ when in solution $\vec{x}_i$, which depends on the temperature, will be denoted by $A_T(i,j)$. Clearly, because of our ordering and because a better solution is always accepted:

$$A_T(i,j) = 1 \text{ for all } j \leq i, \text{ while if } j > i, 0 < A_T(i,j) < 1.$$

From the above definitions and the model in (10.28), it follows that:

$$P_T(i,j) = G(i,j)A_T(i,j) \text{ when } j \neq i. \tag{10.31}$$

We note that for simulated annealing:

$$A_T(i,j) = \begin{cases} \exp\left(\frac{f(\vec{x}_j) - f(\vec{x}_i)}{T}\right) & \text{if } f(\vec{x}_j) > f(\vec{x}_i), \\ 1 & \text{if } f(\vec{x}_j) \leq f(\vec{x}_i). \end{cases} \tag{10.32}$$

Now, for any $s \in (0,1]$, we define a vector,

$$\vec{\pi}_T \equiv (s, s\ A_T(1,2), s\ a_T(1,3), \ldots, s\ a_T(1,|\mathcal{X}|)), \tag{10.33}$$

and then claim that $\vec{\pi}_T$ is the limiting probability (steady-state) vector of the Markov chain whose transition probability from state i to state j equals $P_T(i,j)$. To prove this claim, some work is needed, which we now present.

Case 1: $1 < j < i$.

From (10.32), it is not hard to show that if $l < j < i$ for any l, including 1,

$$\begin{aligned} A_T(l,j)A_T(j,i) &= \exp\left(\frac{f(\vec{x}_j) - f(\vec{x}_l)}{T}\right)\exp\left(\frac{f(\vec{x}_i) - f(\vec{x}_j)}{T}\right) \\ &= \exp\left(\frac{f(\vec{x}_i) - f(\vec{x}_l)}{T}\right) = A_T(l,i). \end{aligned} \tag{10.34}$$

When $1 < j < i$,

$$\begin{aligned} \pi_T(i)P_T(i,j) &= (s\ A_T(1,i))(A_T(i,j)G(i,j)) \\ &\quad \text{(from Eqs. (10.33) and (10.31))} \\ &= s\ A_T(1,i)G(i,j) \text{ (since } A(i,j) = 1 \text{ when } j < i) \\ &= s\ A_T(1,j)A_T(j,i)G(i,j) \\ &\quad \text{(by setting } l = 1 \text{ in Equation (10.34))} \\ &= (s\ A_T(1,j))(A_T(j,i)G(j,i)) \text{ (from Assumption 10.27)} \\ &= \pi_T(j)P_T(j,i) \text{ (from Eqs. (10.33) and (10.31)).} \end{aligned}$$

Thus, from the above, one can conclude that when $1 < j < i$,

$$\pi_T(i)P_T(i,j) = \pi_T(j)P_T(j,i).$$

Case 2: $i = j$. This case is trivial since the left- and right-hand sides of the above are identical.

Case 3: $1 < i < j$: This can be shown in a manner analogous to Case 1.

Then since $\pi_T(i)P_T(i,j) = \pi_T(j)P_T(j,i)$ is true for all (i,j)-pairs, from Lemma 10.26, we conclude that the vector $\vec{\pi}_T$ is the limiting probability vector of the Markov chain. Now, we consider what happens as T tends to 0.

From Eq. (10.32), it follows that

$$\lim_{T \to 0} A_T(1, j) = 0 \text{ for } j \neq 1. \tag{10.35}$$

From Eq. (10.33), we can write:

$$\begin{aligned}\lim_{T\to 0} \vec{\pi}_T &= (s, s \lim_{T\to 0} A_T(1,2), s \lim_{T\to 0} A_T(1,3), \ldots, s \lim_{T\to 0} A_T(1, |\mathcal{X}|)) \\ &= (s, 0, 0, \ldots, 0) \text{ (using Eq. (10.35))} \\ &= (1, 0, 0, \ldots, 0) \text{ (since the limiting probabilities sum to 1).}\end{aligned}$$

We thus have a regular Markov chain for any temperature T, but in the limit as T tends to 0, we have an absorbing Markov chain whose absorbing state is the solution numbered 1, i.e., the optimal solution, and we are done. ∎

It is important to note that at each temperature the algorithm needs to allow a sufficient number of iterations so that the proportion of time spent in the different states "resembles the equilibrium (limiting probability) distribution" [85] of the Markov chain for that temperature. In other words, this line of argument works only when the algorithm performs a sufficiently large number of iterations at all values of T.

We further note that the assumption of symmetric neighborhoods can be relaxed to obtain a slightly different result (see [193]), and that Assumptions 10.22 and 10.28 can also be relaxed [85].

The analysis of the simulated annealing algorithm in which the temperature changes in every iteration, as stated above, relies on constructing a non-stationary Markov chain [206, 67]. Unfortunately, the analysis of the non-stationary Markov chain relies on a number of other results and is beyond our scope here. See [85] for a clear account on this topic.

Simulation-induced noise. The simulation-induced noise may produce an effect on the behavior of simulated annealing, and, hence we present a simple analysis of this effect [108].

PROPOSITION 10.30 *With probability 1, the version of simulated annealing algorithm that uses simulation-based function estimates can be made to mimic the version that uses exact function values (noise-free version) as long as one selects a sufficiently large number of replications in the simulation-based function estimation.*

Proof There are two issues with introducing simulation-induced noise in the algorithm. Recall that in Step 3 of the noise-free version, we defined: $\Delta \equiv f(\vec{x}_{new}) - f(\vec{x}_{current})$. Two possible scenarios can be associated with Step 3: Case 1 for which $\Delta \leq 0$ and Case 2 for which $\Delta > 0$. Now if define $\tilde{\Delta} \equiv \tilde{f}(\vec{x}_{new}) - \tilde{f}(\vec{x}_{current})$, then clearly unless the following holds the noisy algorithm will stray from the noise-free version:

$$\tilde{\Delta} \leq 0 \text{ when } \Delta \leq 0, \tag{10.36}$$

and $\tilde{\Delta} > 0$ when $\Delta > 0$. This is the first step needed to show the result. The second step involves analyzing what happens to the exploration probability in the limit with the noisy estimates. We begin with the first step. Let us consider Case 1 first.

Case 1: We denote the simulation estimate of the function at $\vec{x}$ by $\tilde{f}(\vec{x})$ and the exact value by $f(\vec{x})$. Then, $\tilde{f}(\vec{x}) = f(\vec{x}) + \eta$ where $\eta \in \Re$ denotes the simulation-induced noise that can be positive or negative. Then we can write $\tilde{f}(\vec{x}_{current}) = f(\vec{x}_{current}) + \eta_{current}$ and $\tilde{f}(\vec{x}_{new}) = f(\vec{x}_{new}) + \eta_{new}$. Now if $\eta_1 = |\eta_{new}|$ and $\eta_2 = |\eta_{current}|$, then we have four scenarios:

Scenario 1: $\tilde{f}(\vec{x}_{new}) = f(\vec{x}_{new}) + \eta_1$ and $\tilde{f}(\vec{x}_{current}) = f(\vec{x}_{current}) + \eta_2$

Scenario 2: $\tilde{f}(\vec{x}_{new}) = f(\vec{x}_{new}) + \eta_1$ and $\tilde{f}(\vec{x}_{current}) = f(\vec{x}_{current}) - \eta_2$

Scenario 3: $\tilde{f}(\vec{x}_{new}) = f(\vec{x}_{new}) - \eta_1$ and $\tilde{f}(\vec{x}_{current}) = f(\vec{x}_{current}) + \eta_2$

Scenario 4: $\tilde{f}(\vec{x}_{new}) = f(\vec{x}_{new}) - \eta_1$ and $\tilde{f}(\vec{x}_{current}) = f(\vec{x}_{current}) - \eta_2$

From the strong law of large numbers (see Theorem 2.1), η_1 and η_2 can be made arbitrarily small, i.e., for a given value of $\epsilon > 0$, a sufficiently large number of replications (samples) can be selected such that with probability 1, $\eta_1 \leq \epsilon$ and $\eta_2 \leq \epsilon$. By choosing $\epsilon = -\frac{\Delta}{2}$, we have that the following will be true with probability 1:

$$\eta_1 \leq -\frac{\Delta}{2} \text{ and } \eta_2 \leq -\frac{\Delta}{2}. \tag{10.37}$$

To prove that our result holds for Case 1, we need to show that the relationship in (10.36) is satisfied. Let us first consider Scenario 1. What we show next can be shown for each of the other scenarios in an analogous manner.

$$\begin{aligned}
\tilde{f}(\vec{x}_{new}) - \tilde{f}(\vec{x}_{current}) &= f(\vec{x}_{new}) - f(\vec{x}_{current}) + \eta_1 - \eta_2 \text{ (Scenario 1)} \\
&= \Delta + \eta_1 - \eta_2 \\
&\leq \Delta - \frac{\Delta}{2} - \eta_2 \text{ (from (10.37))} \\
&= \frac{\Delta}{2} - \eta_2 \leq 0 \text{ (from (10.37) since } \eta_2 \geq 0)
\end{aligned}$$

Case 2: Using arguments very similar to those used above, we can show that by selecting a suitable number of replications, we can ensure that:

$$\tilde{\Delta} \equiv \tilde{f}(\vec{x}_{new}) - \tilde{f}(\vec{x}_{current}) > 0 \tag{10.38}$$

when $\Delta > 0$.

We now consider the second step in the proof. We now study the probability of selecting a worse neighbor (exploration) and what happens to it in the limit. Remember, this probability converges to 0 as $m \to \infty$ in the noise-free version. In the noisy version, we need to show that the algorithm exhibits the same behavior. The probability corrupted by noise is:

$$\exp\left(\frac{\tilde{f}(\vec{x}_{new}) - \tilde{f}(\vec{x}_{current})}{T}\right). \tag{10.39}$$

From inequality (10.38), the numerator in the power of the exponential term will always be strictly positive. As a result,

$$\lim_{T \to 0} \exp\left(\frac{\tilde{f}(\vec{x}_{new}) - \tilde{f}(\vec{x}_{current})}{T}\right) = 0. \quad \blacksquare$$

6.5. Modified Stochastic Ruler

Like BAS, the modified stochastic ruler has a stationary Markov chain underlying it. The reader is urged to review the steps and terminology associated with the stochastic ruler from Chap. 5.

We now define the following probability:

$$P(\vec{x}_{new}, a, b) = \mathsf{Pr}(f(\vec{x}_{new}, \omega) \leq U(\omega)),$$

where $f(\vec{x}, \omega)$, a random variable, is the random value obtained of the function $f(\vec{x})$ from one replication (sample) and $U(\omega)$ is a random value obtained from the uniform distribution $Unif(a, b)$. Note that $\vec{x}_{new}$ is itself a random variable since the candidate is generated using the generator matrix $\mathbf{G}$. Then, from the steps in the algorithm, it is clear that the transition probabilities of the Markov chain underlying the modified stochastic ruler can be described as:

$$P(i,j) = \begin{cases} [P(\vec{x}_{new}, a, b)]^I & \text{if } i \neq j \\ 1 - [P(\vec{x}_{new}, a, b)]^I & \text{if } i = j \end{cases}$$

It is not hard to see that when the current solution is the global optimum, $P(\vec{x}_{new}, a, b)$ will equal zero for any $\vec{x}_{new}$. Hence from the above, at the global optimum, $\vec{x}_*$, we will have that $P(\vec{x}_*, \vec{x}_*) = 1$, i.e., the Markov chain will be an absorbing one in which the global optimum is the absorbing state, assuming we have a unique global optimum. This implies that eventually, the system will be absorbed into the global optimum. The reader is referred to [6] for additional analysis and insights.

7. Concluding Remarks

Our goal in this chapter was to present a subset of the results in the convergence theory of model-free parametric optimization techniques. Our goal in this chapter was not very ambitious in that we restricted our attention to results that can be proved without using very complicated mathematical arguments. The initial result on the steepest-descent rule was presented because it is repeatedly used throughout this book in various contexts. Overall, we presented some preliminary analysis related to the convergence of steepest descent and some SAS techniques. The bibliographic remarks mention several references to additional works that cover this topic in greater depth.

Bibliographic Remarks. Classical steepest-descent theory can be found in [29]. In Theorem 10.8, result **R1** was shown by [233] and result **R2** can be found in [33]. The convergence of simultaneous perturbation in the presence of noise was first established in Spall [280]. Our account here using stochastic gradients (Lemma 10.11 and Theorem 10.12) follows from Gosavi [108]. The ODE analysis in Sect. 5.2 is new, while the material in Sect. 5.3 is from Spall [281].

Simulated annealing was first analyzed in Lundy and Mees [193]. Some other papers that also study its convergence properties are Fox and Heine [88], Gelfand and Mitter [96], and Alrefaei and Andradóttir [5]. For the use of non-stationary Markov chains in convergence of simulated annealing, see Fielding [85], Cohn and Fielding [67], and Mitra et al. [206]. Our analysis studying the effect of simulation noise is based on Gosavi [108]. Tabu search [102], the genetic algorithm [256], LAST [298], BAS [182], stochastic ruler [329, 6], and nested partitions [273] have also been treated for convergence analysis in the literature. The text of Zabinsky [333] provides a sophisticated treatment of the topic of SAS techniques for global optimization.

Chapter 11

CONVERGENCE ANALYSIS OF CONTROL OPTIMIZATION METHODS

1. Chapter Overview

This chapter will discuss the proofs of optimality of *a subset* of algorithms discussed in the context of control optimization. The chapter is organized as follows. We begin in Sect. 2 with some definitions and notation related to discounted and average reward Markov decision problems (MDPs). Subsequently, we present convergence theory related to dynamic programming (DP) for MDPs in Sects. 3 and 4. In Sect. 5, we discuss some selected topics related to semi-MDPs (SMDPs). Thereafter, from Sect. 6, we present a selected collection of topics related to convergence of reinforcement learning (RL) algorithms.

For DP, we begin by establishing that the Bellman equation can indeed be used to generate an optimal solution. Then we prove that the classical versions of value and policy iteration can be used to generate optimal solutions. It has already been discussed that the classical value-function-based algorithms have Q-factor equivalents. For RL, we first present some fundamental results from stochastic approximation. Thereafter, we use these results to prove convergence of the algorithm. The reader will need material from Chap. 9 and should go back to review material from there, as and when it is required.

2. Dynamic Programming: Background

We have seen the Bellman equation in various forms thus far. For the sake of convergence analysis, we need to express the Bellman equation in the form of a "transformation." The value function can

A. Gosavi, *Simulation-Based Optimization*, Operations Research/Computer Science Interfaces Series 55,
DOI 10.1007/978-1-4899-7491-4_11

then be viewed as a vector that gets transformed every time the Bellman transformation is applied on it.

We will first define a couple of transformations related to the discounted reward problem, and then define the corresponding transformations for the average reward problem.

The transformation T, you will recognize, is the one that we use in the value iteration algorithm for discounted reward (of course, it is derived from the Bellman optimality equation). T is defined as:

$$TJ(i) = \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i,a,j)J(j) \right] \text{ for all } i \in \mathcal{S}, \quad (11.1)$$

where $TJ(i)$ denotes the ith component of the vector $T(\vec{J})$. Although all the terms used here have been defined in previous chapters, we repeat the definitions for the sake of your convenience.

- The symbols i and j stand for the states of the Markov chain and are members of $\mathcal{S}$—the set of states. The notation $|\mathcal{S}|$ denotes the number of elements in this set.
- a denotes an action and $\mathcal{A}(i)$ denotes the set of actions allowed in state i.
- λ stands for the discounting factor.
- $p(i,a,j)$ denotes the probability of transition (of the Markov chain) in one step from state i to state j when action a is selected in state i.
- $\bar{r}(i,a)$ denotes the **expected** immediate reward earned in a one-step transition (of the Markov chain) when action a is selected in state i. The term $\bar{r}(i,a)$ is defined as shown below:

  $$\bar{r}(i,a) = \sum_{j=1}^{|\mathcal{S}|} p(i,a,j)r(i,a,j), \quad (11.2)$$

  where $r(i,a,j)$ is the immediate reward earned in a one-step transition (of the Markov chain) when action a is selected in state i and the next state happens to be j.
- $J(i)$ denotes the ith component of the vector $\vec{J}$, which is the vector that is transformed by T.

Important note: The summation notation of (11.2) will also be at times denoted by $\sum_{j \in \mathcal{S}}$. Further note that the transformation T can be also be viewed as a function $T(.)$ where $T : \Re^{|\mathcal{S}|} \rightarrow \Re^{|\mathcal{S}|}$.

The next important transformation that we define is the one associated with the Bellman equation for a given policy. It is denoted by $T_{\hat{\mu}}$ and is defined as:

$$T_{\hat{\mu}}J(i) = \bar{r}(i,\mu(i)) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i,\mu(i),j)J(j) \text{ for all } i \in \mathcal{S}. \tag{11.3}$$

Here $\mu(i)$ denotes the action to be taken in state i when policy $\hat{\mu}$ is followed.

By setting $\lambda = 1$ in T, one obtains L. Similarly, by setting $\lambda = 1$ in $T_{\hat{\mu}}$, one obtains $L_{\hat{\mu}}$. Thus, $L_{\hat{\mu}}$ and L are the average reward operators corresponding to $T_{\hat{\mu}}$ and T respectively. We note, however, that in average reward algorithms, we often use modifications of L and $L_{\hat{\mu}}$. For the sake of completeness, we next define the transformations L and $L_{\hat{\mu}}$, where $L(\vec{x})$ and $L_{\hat{\mu}}(\vec{x})$ will denote vectors, while $Lx(i)$ and $L_{\hat{\mu}}x(i)$ will denote their ith components respectively.

$$\text{For all } i \in \mathcal{S},\ LJ(i) = \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \sum_{j=1}^{|\mathcal{S}|} p(i,a,j)J(j) \right]; \tag{11.4}$$

$$L_{\hat{\mu}}J(i) = \bar{r}(i,\mu(i)) + \sum_{j=1}^{|\mathcal{S}|} p(i,\mu(i),j)J(j). \tag{11.5}$$

All the relevant terms have been defined in the context of T and $T_{\hat{\mu}}$. Some more notation and definitions are needed:

2.1. Special Notation

Let F denote a dynamic programming operator (such as T or $L_{\hat{\mu}}$). Then the notation F^k has a special meaning, which has been explained in the previous chapter. We quickly explain it here again.

$T_{\hat{\mu}}^2\vec{J}$ will denote the mapping $T_{\hat{\mu}}$ (see Eq. (11.3)) applied to the vector $T_{\hat{\mu}}(\vec{J})$. In other words, the definition is:

$$T_{\hat{\mu}}^2 J(i) = \bar{r}(i,\mu(i)) + \lambda \sum_{j \in \mathcal{S}} p(i,\mu(i),j)T_{\hat{\mu}}J(j) \text{ for all } i \in \mathcal{S}.$$

In general, for any $k = 2, 3, 4, \ldots$,

$$T_{\hat{\mu}}^k(\vec{J}) = T_{\hat{\mu}}\left(T_{\hat{\mu}}^{k-1}(\vec{J})\right).$$

Similarly, T^2 will denote the following:

$$T^2 J(i) = \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \lambda \sum_{j \in \mathcal{S}} p(i,a,j) T J(j) \right] \text{ for all } i \in \mathcal{S},$$

where T has been defined in (11.1). The meaning of the notations L^k and $L^k_{\hat{\mu}}$ follows in an analogous manner. We now discuss an important property of monotonicity that will be needed later.

2.2. Monotonicity of $T, T_{\hat{\mu}}, L$, and $L_{\hat{\mu}}$

The monotonicity of a transformation F implies that given two vectors $\vec{J}$ and $\vec{J}'$, which satisfy the relationship:

$$J(i) \leq J'(i)$$

for all values of i, the following is true for every positive value of k:

$$F^k J(i) \leq F^k J'(i) \text{ for all values of } i.$$

We will establish this monotonicity result for T and $T_{\hat{\mu}}$. The monotonicity result can be established for L and $L_{\hat{\mu}}$ by setting $\lambda = 1$ in the respective results for T and $T_{\hat{\mu}}$.

Let us consider the result for T. We will use an induction argument. $\mathcal{A}(i)$ will denote the set of actions allowable in state i. Now for all $i \in \mathcal{S}$,

$$\begin{aligned} TJ(i) &= \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \lambda \sum_{j \in \mathcal{S}} p(i,a,j) J(j) \right] \\ &\leq \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \lambda \sum_{j \in \mathcal{S}} p(i,a,j) J'(j) \right] \\ &= TJ'(i). \end{aligned}$$

Thus, the relation is true when $k = 1$. Next, we assume that the result holds when $k = m$. Thus if $J(i) \leq J'(i)$ for all values of i, then $T^m(J(i)) \leq T^m(J'(i))$.

$$\begin{aligned} \text{Now, for all } i \in \mathcal{S},\ T^{m+1} J(i) &= \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \lambda \sum_{j \in \mathcal{S}} p(i,a,j) T^m J(j) \right] \\ &\leq \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \lambda \sum_{j \in \mathcal{S}} p(i,a,j) T^m J'(j) \right] \\ &= T^{m+1} J'(i), \text{ completing the induction.} \end{aligned}$$

Next, we will show that the result holds for $T_{\hat{\mu}}$, i.e., for policy $\hat{\mu}$. The proof is similar to the one above.

$$\begin{aligned}
\text{Since for all } i \in \mathcal{S},\ T_{\hat{\mu}}J(i) &= \left[\bar{r}(i,\mu(i)) + \lambda \sum_{j\in\mathcal{S}} p(i,\mu(i),j)J(j)\right] \\
&\leq \left[\bar{r}(i,\mu(i)) + \lambda \sum_{j\in\mathcal{S}} p(i,\mu(i),j)J'(j)\right] \\
&= T_{\hat{\mu}}J'(i), \text{ the result must hold for } k{=}1.
\end{aligned}$$

Now assuming that the result is true when $k = m$,

$$J(i) \leq J'(i) \text{ for all } i \in \mathcal{S} \text{ will imply that } T^m_{\hat{\mu}}(J(i)) \leq T^m_{\hat{\mu}}(J'(i)).$$

$$\begin{aligned}
\text{Then for all } i \in \mathcal{S},\ T^{m+1}_{\hat{\mu}}J(i) &= \left[\bar{r}(i,\mu(i)) + \lambda \sum_{j\in\mathcal{S}} p(i,\mu(i),j)T^m_{\hat{\mu}}J(j)\right] \\
&\leq \left[\bar{r}(i,\mu(i)) + \lambda \sum_{j\in\mathcal{S}} p(i,\mu(i),j)T^m_{\hat{\mu}}J'(j)\right] \\
&= T^{m+1}_{\hat{\mu}}J'(i), \text{ completing the induction.}
\end{aligned}$$

2.3. Key Results for Average and Discounted MDPs

We will now present some useful lemmas for discounted and average reward. These results will be used for proving the optimality of the Bellman equation and can be found (without the proofs) in Vol II of [30]. Lemma 11.1 is related to discounted reward and Lemma 11.2 is related to average reward.

LEMMA 11.1 *Given a bounded function $h : \mathcal{S} \to \Re$, if $r(x_s, a, x_{s+1})$ denotes the immediate reward earned in the sth jump of the Markov chain under the influence of action a and $\hat{\mu}$ denotes the policy used to control the Markov chain, then, for all values of $i \in \mathcal{S}$:*

$$T^k_{\hat{\mu}}h(i) = \mathsf{E}_{\hat{\mu}}\left[\lambda^k h(x_{k+1}) + \sum_{s=1}^{k} \lambda^{s-1} r(x_s, \mu(x_s), x_{s+1}) \middle| x_1 = i\right], \text{ where}$$

x_s denotes the state from which the sth jump of the Markov chain of the policy occurs, λ^k equals λ raised to the kth power, and $\mathsf{E}_{\hat{\mu}}$, the expectation over the trajectory of states produced by policy $\hat{\mu}$, is defined as follows:

$$\mathsf{E}_{\hat{\mu}}\left[\lambda^k h(x_{k+1}) + \sum_{s=1}^{k} \lambda^{s-1} r(x_s, \mu(x_s), x_{s+1}) \middle| x_1 = i\right]$$

$$\equiv \sum_{x_2 \in \mathcal{S}} p(x_1, \mu(x_1), x_2)[r(x_1, \mu(x_1), x_2)] +$$

$$\sum_{x_2 \in \mathcal{S}} p(x_1, \mu(x_1), x_2) \times \sum_{x_3 \in \mathcal{S}} p(x_2, \mu(x_2), x_3)[\lambda r(x_2, \mu(x_2), x_3)] + \cdots +$$

$$\sum_{x_2 \in \mathcal{S}} p(x_1, \mu(x_1), x_2) \times \sum_{x_3 \in \mathcal{S}} p(x_2, \mu(x_2), x_3) \times \cdots \times \sum_{x_{k+1} \in \mathcal{S}} p(x_k, \mu(x_k), x_{k+1}) \times$$

$$\left[\lambda^{k-1} r(x_k, \mu(x_k), x_{k+1}) + \lambda^k h(x_{k+1})\right]. \tag{11.6}$$

We now present a simple proof for this, but the reader can skip it without loss of continuity.

Proof We will use induction on k. From the definition of $T_{\hat{\mu}}$ in Eq. (11.3), for all $i \in \mathcal{S}$,

$$\begin{aligned} T_{\hat{\mu}} h(i) &= \sum_{j \in \mathcal{S}} p(i, \mu(i), j)\left[r(i, \mu(i), j) + \lambda h(j)\right] \\ &= \mathsf{E}_{\hat{\mu}}\left[\lambda h(x_2) + \sum_{s=1}^{1} [r(x_s, \mu(x_s), x_{s+1})] \middle| x_1 = i\right], \text{ when } j = x_2, \end{aligned}$$

and thus the result holds for $k = 1$. We must now show that the result holds for $k = m + 1$. Assuming the result to hold for $k = m$, we have:

$$\text{For all } i \in \mathcal{S},\quad T_{\hat{\mu}}^m h(i) = \mathsf{E}_{\hat{\mu}}\left[\lambda^m h(x_{m+1}) + \sum_{s=1}^{m} \lambda^{s-1} r(x_s, \mu(x_s), x_{s+1}) \middle| x_1 = i\right]$$

which implies that if $x_2 = i$ (instead of x_1 equaling i):

$$T_{\hat{\mu}}^m h(i) = \mathsf{E}_{\hat{\mu}}\left[\lambda^m h(x_{m+2}) + \sum_{s=2}^{m+1} \lambda^{s-2} r(x_s, \mu(x_s), x_{s+1}) \middle| x_2 = i\right],$$

$$\text{i.e., } T_{\hat{\mu}}^m h(x_2) = \mathsf{E}_{\hat{\mu}}\left[\lambda^m h(x_{m+2}) + \sum_{s=2}^{m+1} \lambda^{s-2} r(x_s, \mu(x_s), x_{s+1})\right]. \tag{11.7}$$

Now, $T_{\hat{\mu}}^{m+1}h(x_1) = T_{\hat{\mu}}(T_{\hat{\mu}}^{m}h(x_1))$

$$= \sum_{x_2} p(x_1,\mu(x_1),x_2)\left[r(x_1,\mu(x_1),x_2) + \lambda T_{\hat{\mu}}^{m}h(x_2)\right] \tag{11.8}$$

$$= \sum_{x_2} p(x_1,\mu(x_1),x_2)\Bigg[r(x_1,\mu(x_1),x_2) + \lambda\times$$

$$\mathsf{E}_{\hat{\mu}}\left[\lambda^m h(x_{m+2}) + \sum_{s=2}^{m+1}\lambda^{s-2} r(x_s,\mu(x_s),x_{s+1})\right]\Bigg] \tag{11.9}$$

$$= \mathsf{E}_{\hat{\mu}}\left[\lambda^{m+1} h(x_{m+2}) + \sum_{s=1}^{m+1}\lambda^{s-1} r(x_s,\mu(x_s),x_{s+1})\right], \tag{11.10}$$

where (11.8) follows from Eq. (11.3), (11.9) follows from Eq. (11.7), and (11.10) follows from the definition of $\mathsf{E}_{\hat{\mu}}$ in (11.6). Then, for $x_1 = i$, we have that for all $i \in \mathcal{S}$:

$$T_{\hat{\mu}}^{m+1}h(i) = \mathsf{E}_{\mu}\left[\lambda^{m+1} h(x_{m+2}) + \sum_{s=1}^{m+1} r(x_s,\mu(x_s),x_{s+1})\middle| x_1 = i\right]. \quad \blacksquare$$

The following lemma can be obtained by setting $\lambda = 1$ in the previous lemma.

LEMMA 11.2 *Given a bounded function $h : \mathcal{S} \to \Re$, if $r(x_s,a,x_{s+1})$ denotes the immediate reward earned in the sth jump of the Markov chain under the influence of action a and $\hat{\mu}$ denotes the policy used to control the Markov chain, then, for all values of $i \in \mathcal{S}$:*

$$L_{\hat{\mu}}^{k}h(i) = \mathsf{E}_{\hat{\mu}}\left[h(x_{k+1}) + \sum_{s=1}^{k} r(x_s,\mu(x_s),x_{s+1})\middle| x_1 = i\right].$$

The implication of Lemma 11.1 (Lemma 11.2) is that if one selects any $|\mathcal{S}|$-dimensional bounded vector and applies the mapping $T_{\hat{\mu}}$ ($L_{\hat{\mu}}$) upon it k times, one obtains the expected *total discounted reward* (*total reward*) earned in a finite trajectory of k jumps (of the Markov chain) starting at state i and using the policy $\hat{\mu}$.

Before starting a discussion on analysis of DP algorithms, we would like to emphasize that throughout the book, we have made the following assumptions:

1. All immediate rewards are finite, i.e.,

   $|r(x_s,\mu(x_s),x_{s+1})| \le M_1$ for all s for some positive value of M_1.

2. All transition times are finite, i.e.,

$$|t(x_s, \mu(x_s), x_{s+1})| \leq M_2 \quad \text{for all } s \text{ for some positive value of } M_2.$$

3. The state space and the action space in every problem (MDP or SMDP) are finite. Further, the Markov chain of any policy in the MDP/SMDP is regular (regularity is discussed in Chap. 6).

3. Discounted Reward DP: MDPs

This section will deal with the analysis of the convergence of algorithms used for the (total) discounted reward criterion. This section will only discuss some classical dynamic programming methods.

3.1. Bellman Equation for Discounted Reward

The Bellman equation for discounted reward was motivated in a heuristic sense in previous chapters. We will now discuss its optimality, i.e., show that a solution of the Bellman equation identifies the optimal solution for a discounted reward MDP. We first provide some key definitions.

Definition 11.1 *The value function vector associated with a given policy $\hat{\mu}$ for the discounted reward case is defined as:*

$$J_{\hat{\mu}}(i) \equiv \lim_{k\to\infty} \mathsf{E}_{\hat{\mu}}\left[\sum_{s=1}^{k} \lambda^{s-1} r(x_s, \mu(x_s), x_{s+1}) \middle| x_1 = i\right] \qquad \text{for all } i \in \mathcal{S}. \tag{11.11}$$

In this definition, x_s is the state from where the sth transition occurs in the Markov chain. The definition implies that $J_{\hat{\mu}}(i)$ is the total discounted reward earned along an infinitely long trajectory (sequence) of states by starting at state i and following policy $\hat{\mu}$ at every state encountered. So in other words, we control the Markov chain with a policy $\hat{\mu}$, and observe the rewards over an infinitely large number of jumps (transitions). Of course, future rewards are discounted with a factor of λ. The expectation operator is used in the definition above because x_{s+1} is a stochastic element. (Remember there is no unique x_{s+1} associated with a given x_s because of the stochastic nature of the transitions). The expectation is over all the trajectories.

Admissible policies. Thus far, by policy, we have meant a *stationary, deterministic* policy. A deterministic policy is one in which one always selects a single action in a given state; this is opposed to a so-called stochastic policy in which every action allowed in that state is selected with some probability less than or equal to 1. Thus, the notion of a stochastic policy is more general than that of a deterministic policy. Every deterministic policy can be viewed as a special case of a stochastic policy in which in every state the probability of selecting one action equals 1 while the probabilities of selecting the other actions equal 0.

In a stationary policy, the probability of selecting a given action (which could equal 1 or 0) in a state is the same regardless of which jump the state is encountered in. In a non-stationary policy, on the other hand, this probability could be different in every jump for any and every state. In general, a so-called *admissible* policy is non-stationary, possibly composed of different stationary, stochastic policies, $\hat{\mu}_1, \hat{\mu}_2, \ldots$, associated to each jump. Thus, in each jump, we could have action selection being performed according to a different stationary, stochastic policy.

We now present a very important definition, that of the **optimal** value function.

DEFINITION 11.2 *The optimal value function vector for the discounted MDP is defined as:*

$$J^*(i) \equiv \max_{all\ \sigma} J_\sigma(i) \qquad \textit{for each } i \in \mathcal{S}, \tag{11.12}$$

where $\sigma = \{\hat{\mu}_1, \hat{\mu}_2 \ldots\}$ *is an admissible policy in which policy* $\hat{\mu}_s$ *is selected in the sth jump of the Markov chain, and for each* $i \in \mathcal{S}$*,*

$$J_\sigma(i) \equiv \lim_{k\to\infty} \mathsf{E}_\sigma \left[\sum_{s=1}^{k} \lambda^{s-1} r(x_s, \mu_s(x_s), x_{s+1}) \middle| x_1 = i \right].$$

What is important to note is that via Eq. (11.12), we define an optimal solution over all admissible policies. In other words, the value function of the optimal policy is one that maximizes the value function over all admissible policies. Note also that we drop the hat $\hat{}$ symbol over the policy's name to indicate that it is an admissible policy. Thus, J_σ denotes the value function of an admissible policy σ, while $J_{\hat{d}}$ denotes the value function of a stationary deterministic policy $\hat{d}$.

We now prove an important property of the transformation underlying the Bellman equation, namely, the contraction property. This property will help in showing that the Bellman equation has a unique solution. The property will also be helpful later in showing the ϵ-convergence of value iteration. To proceed any further, one should become familiar with the notion of contraction mappings and the Fixed Point Theorem (Theorem 9.10 on page 307)—both topics are discussed in detail in Chap. 9.

PROPOSITION 11.3 *The mapping T, i.e., the Bellman operator for discounted reward, is contractive with respect to the max norm. In other words, given two vectors, $\vec{J}$ and $\vec{J}'$ in $\mathcal{S}$, we have that:*

$$||T\vec{J} - T\vec{J}'|| \leq \varsigma ||\vec{J} - \vec{J}'||,$$

where $||.||$ is the max-norm, $\varsigma \in (0,1)$, and ς equals λ, the discounting factor.

Proof Let state space $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$ where $\mathcal{S}_1$ and $\mathcal{S}_2$ will be defined below.

Case 1: Assume that for all $i \in \mathcal{S}_1$, $TJ(i) \geq TJ'(i)$.

Case 2: Assume that $TJ'(i) \geq TJ(i)$ for all $i \in \mathcal{S}_2$.

We first consider **Case 1**.

$$\text{Now, define for every } i \in \mathcal{S}_1, \quad a(i) \in \arg\max_{u \in \mathcal{A}(i)} \left[\bar{r}(i,u) + \lambda \sum_{j \in \mathcal{S}} p(i,u,j)J(j) \right].$$

In other words, $a(i)$ denotes an action in the ith state that will maximize the quantity in the square brackets above. This implies that:

$$TJ(i) = \left[\bar{r}(i,a(i)) + \lambda \sum_{j \in \mathcal{S}} p(i,a(i),j)J(j) \right] \text{ for every } i \in \mathcal{S}_1.$$

$$\text{Similarly, let } b(i) \in \arg\max_{u \in \mathcal{A}(i)} \left[\bar{r}(i,u) + \lambda \sum_{j \in \mathcal{S}} p(i,u,j)J'(j) \right]$$

for all $i \in \mathcal{S}_1$, which will imply that:

$$TJ'(i) = \left[\bar{r}(i,b(i)) + \lambda \sum_{j \in \mathcal{S}} p(i,b(i),j)J'(j) \right] \text{ for every } i \in \mathcal{S}_1.$$

Since action $b(i)$ maximizes the quantity in the square brackets above,

$$\text{for every } i \in \mathcal{S}_1, \; TJ'(i) \geq \left[\bar{r}(i,a(i)) + \lambda \sum_{j \in \mathcal{S}} p(i,a(i),j)J'(j) \right].$$

$$\text{Then, for every } i \in \mathcal{S}_1, \quad -TJ'(i) \le -\left[\bar{r}(i,a(i)) + \lambda \sum_{j\in\mathcal{S}} p(i,a(i),j)J'(j)\right].$$

Combining this with the definition of $TJ(i)$ and the fact that $TJ(i) \ge TJ'(i)$ for all $i \in \mathcal{S}_1$, we have that for every $i \in \mathcal{S}_1$:

$$\begin{aligned}
0 &\le TJ(i) - TJ'(i) \\
&\le \left[\bar{r}(i,a(i)) + \lambda \sum_{j\in\mathcal{S}} p(i,a(i),j)J(i)\right] - \\
&\left[\bar{r}(i,a(i)) + \lambda \sum_{j\in\mathcal{S}} p(i,a(i),j)J'(i)\right] \\
&= \lambda \sum_{j\in\mathcal{S}} p(i,a(i),j)[J(i) - J'(j)] \\
&\le \lambda \sum_{j\in\mathcal{S}} p(i,a(i),j) \max_j |J(j) - J'(j)| \\
&= \lambda \max_j |J(j) - J'(j)| \left(\sum_{j\in\mathcal{S}} p(i,a(i),j)\right) \\
&= \lambda \max_j |J(j) - J'(j)|(1) = \lambda ||\vec{J} - \vec{J'}||.
\end{aligned}$$

$$\text{Thus, for all } i \in \mathcal{S}_1\text{: } TJ(i) - TJ'(i) \le \lambda ||\vec{J} - \vec{J'}||. \tag{11.13}$$

Case 2: Using logic similar to that used above, one can show that:

$$TJ'(i) - TJ(i) \le \lambda ||\vec{J'} - \vec{J}|| = \lambda ||\vec{J} - \vec{J'}|| \text{ for all } i \in \mathcal{S}_2. \tag{11.14}$$

Together (11.13) and (11.14) imply that:

$$|TJ(i) - TJ'(i)| \le \lambda ||\vec{J} - \vec{J'}|| \text{ for all } i \in \mathcal{S}. \tag{11.15}$$

This follows from the fact that the LHS of each of the inequalities (11.13) and (11.14) has to be positive. Hence, when the absolute value of the LHS is selected, both (11.13) and (11.14) will imply (11.15).

Since inequality (11.15) holds for **any** $i \in \mathcal{S}$, it also holds for the value of $i \in \mathcal{S}$ that maximizes $|TJ(i) - TJ'(i)|$. Therefore:

$$\max_i |TJ(i) - TJ'(i)| \le \lambda ||\vec{J} - \vec{J'}||; \text{ i.e., } ||T\vec{J} - T\vec{J'}|| \le \lambda ||\vec{J} - \vec{J'}||. \quad \blacksquare$$

The next result shows that the property also holds for the mapping associated with a given policy.

PROPOSITION 11.4 *The mapping $T_{\hat{\mu}}$, i.e., the dynamic programming operator associated with a policy $\hat{\mu}$ for discounted reward, is contractive. In particular we will prove that given two vectors $\vec{J}$ and $\vec{J}'$,*

$$||T_{\hat{\mu}}\vec{J} - T_{\hat{\mu}}\vec{J}'|| \leq \lambda||\vec{J} - \vec{J}'||,$$

where $||.||$ is the max-norm and $\lambda \in (0,1)$ is the discounting factor.

Proof The proof is very similar to that above. Let state space $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$ where $\mathcal{S}_1$ and $\mathcal{S}_2$ will be defined below.

Case 1: Assume that for all $i \in \mathcal{S}_1$, $T_{\hat{\mu}}J(i) \geq T_{\hat{\mu}}J'(i)$.

Case 2: Assume that for all $i \in \mathcal{S}_2$, $T_{\hat{\mu}}J'(i) \geq T_{\hat{\mu}}J(i)$.

We consider Case 1 first. Using arguments similar to that in Case 1 for the proof above, we have that for every $i \in \mathcal{S}_1$:

$$\begin{aligned}
0 &\leq T_{\hat{\mu}}J(i) - T_{\hat{\mu}}J'(i) \\
&= \left[\bar{r}(i,\mu(i)) + \lambda\sum_{j\in\mathcal{S}} p(i,\mu(i),j)J(i)\right] - \\
&\left[\bar{r}(i,\mu(i)) + \lambda\sum_{j\in\mathcal{S}} p(i,\mu(i),j)J'(i)\right] \\
&= \lambda\sum_{j\in\mathcal{S}} p(i,\mu(i),j)[J(i) - J'(j)] \\
&\leq \lambda\sum_{j\in\mathcal{S}} p(i,\mu(i),j)\max_j |J(j) - J'(j)| \\
&= \lambda\max_j |J(j) - J'(j)| \left(\sum_{j\in\mathcal{S}} p(i,a(i),j)\right) \\
&= \lambda\max_j |J(j) - J'(j)|(1) = \lambda||\vec{J} - \vec{J}'||.
\end{aligned}$$

$$\text{Thus, for all } i \in \mathcal{S}_1\text{: } T_{\hat{\mu}}J(i) - T_{\hat{\mu}}J'(i) \leq \lambda||\vec{J} - \vec{J}'||.$$

The Case 2 can be argued as in the proof above, and the result follows using very similar arguments. ∎

We will now present two key results, Propositions 11.5 and 11.6, that will help in analyzing the Bellman optimality and the Bellman policy equation, respectively.

PROPOSITION 11.5 *(i) For any bounded function $h : \mathcal{S} \to \Re$, the following limit exists:*

$$\lim_{k\to\infty} T^k h(i) \qquad \textit{for all } i \in \mathcal{S}.$$

(ii) Furthermore, this limit equals the optimal value function.

The proposition implies that if **any** vector (say $\vec{h}$) with finite values for all its components is selected and the transformation T is applied infinitely many times on it, one obtains the **optimal** discounted reward value function vector (i.e., $\vec{J}^*$).

Proof Part (i): Proposition 11.3 implies that T is contractive, and hence Theorem 9.10 implies that the limit exists. Part (ii): We skip this proof; the reader is referred to Prop. 1.2.1 from [30]. ∎

We now present the key result related to $T_{\hat{\mu}}$.

PROPOSITION 11.6 *Consider the value function vector, $\vec{J}_{\hat{\mu}}$, associated to a policy, $\hat{\mu}$, which was defined in Definition 11.1. Then, for any bounded function $h : \mathcal{S} \to \Re$, the following equation applies:*

$$J_{\hat{\mu}}(i) = \lim_{k\to\infty} T^k_{\hat{\mu}} h(i) \textit{ for all } i \in \mathcal{S}.$$

Proof The proof is somewhat long and is organized as follows. First, we will express the value function as a sum of two parts: one part constitutes of the reward earned in the first P (where P is any finite integer) steps of the trajectory and the second constitutes of the same earned over the remainder of the trajectory. Then, we will use bounds on the function $h(.)$ to develop an inequality containing the value function. Finally, this inequality will be used to show that the value function vector is sandwiched between the limiting value of $\left\{T^k_{\hat{\mu}}(\vec{h})\right\}_{k=1}^{\infty}$ and will hence equal the limit.

Via Eq. (11.11):

$$\begin{aligned} J_{\hat{\mu}}(i) &= \lim_{k\to\infty} \mathsf{E}_{\hat{\mu}} \left[\sum_{s=1}^{k} \lambda^{s-1} r(x_s, \mu(x_s), x_{s+1}) \middle| x_1 = i \right] \\ &= \lim_{k\to\infty} \mathsf{E}_{\hat{\mu}} \left[\sum_{s=1}^{P} \lambda^{s-1} r(x_s, \mu(x_s), x_{s+1}) \middle| x_1 = i \right] \end{aligned}$$

$$+ \lim_{k\to\infty} \mathsf{E}_{\hat{\mu}} \left[\sum_{s=P+1}^{k} \lambda^{s-1} r(x_s, \mu(x_s), x_{s+1}) \middle| x_1 = i \right]$$

$$= \mathsf{E}_{\hat{\mu}} \left[\sum_{s=1}^{P} \lambda^{s-1} r(x_s, \mu(x_s), x_{s+1}) \middle| x_1 = i \right]$$

$$+ \lim_{k\to\infty} \mathsf{E}_{\hat{\mu}} \left[\sum_{s=P+1}^{k} \lambda^{s-1} r(x_s, \mu(x_s), x_{s+1}) \middle| x_1 = i \right] \quad (11.16)$$

Throughout this book, we assume that all immediate rewards are finite, i.e.,

$$|r(x_s, \mu(x_s), x_{s+1})| \le M \quad \text{for all } s \text{ for some positive value of } M.$$

Hence $$\left| \lim_{k\to\infty} \mathsf{E}_{\hat{\mu}} \left[\sum_{s=P+1}^{k} \lambda^{s-1} r(x_s, \mu(x_s), x_{s+1}) \middle| x_1 = i \right] \right|$$

$$\le \lim_{k\to\infty} \left[\sum_{s=P+1}^{k} \lambda^{s-1} M \right] = M \sum_{s=P+1}^{\infty} \lambda^{s-1}$$

$$= M \frac{\lambda^P}{1-\lambda} \quad \text{(sum of infinite GP series (see Eq. (9.4)))}.$$

Denoting $\lim_{k\to\infty} \mathsf{E}_{\hat{\mu}} \left[\sum_{s=P+1}^{k} \lambda^{s-1} r(x_s, \mu(x_s), x_{s+1}) \middle| x_1 = i \right]$ by A,

we have from the above that, $|A| \le \dfrac{M\lambda^P}{1-\lambda}$

which implies that: $-\dfrac{M\lambda^P}{1-\lambda} \le A \le \dfrac{M\lambda^P}{1-\lambda}$.

Multiplying the above inequations by -1, we have that

$$\frac{M\lambda^P}{1-\lambda} \ge -A \ge -\frac{M\lambda^P}{1-\lambda}.$$

Adding $J_{\hat{\mu}}(i)$ to each side, we have:

$$J_{\hat{\mu}}(i) + \frac{M\lambda^P}{1-\lambda} \ge J_{\hat{\mu}}(i) - A \ge J_{\hat{\mu}}(i) - \frac{M\lambda^P}{1-\lambda}.$$

Using (11.16), the above can be written as:

$$J_{\hat{\mu}}(i) + \frac{M\lambda^P}{1-\lambda} \ge \mathsf{E}_{\hat{\mu}} \left[\sum_{s=1}^{P} \lambda^{s-1} r(x_s, \mu(x_s), x_{s+1}) \middle| x_1 = i \right]$$

$$\geq J_{\hat{\mu}}(i) - \frac{M\lambda^P}{1-\lambda}. \tag{11.17}$$

Now, for any bounded function $h(.)$, we have that $\max_{j\in\mathcal{S}} |h(j)| \geq h(j)$ for any $j \in \mathcal{S}$. Hence, it follows that

$$-\max_{j\in\mathcal{S}} |h(j)| \leq \mathsf{E}_{\hat{\mu}}[h(x_{P+1})|x_1 = i] \leq \max_{j\in\mathcal{S}} |h(j)|$$

where $\mathsf{E}_{\hat{\mu}}\,[.\ |x_1 = i]$ is used as defined in Eq. (11.6). Then, since $\lambda > 0$, one has that:

$$\max_{j\in\mathcal{S}} |h(j)|\lambda^P \geq \mathsf{E}_{\hat{\mu}}[\lambda^P h(x_{P+1})\big|\, x_1 = i] \geq -\max_{j\in\mathcal{S}} |h(j)|\lambda^P. \tag{11.18}$$

Adding (11.17) and (11.18), it follows that

$$J_{\hat{\mu}}(i) + \frac{M\lambda^P}{1-\lambda} + \max_{j\in\mathcal{S}} |h(j)|\lambda^P \geq$$

$$\mathsf{E}_{\hat{\mu}}\left[\sum_{s=1}^{P} \lambda^{k-1} r(x_s, \mu(x_s), x_{s+1})\middle|\, x_1 = i\right] + \mathsf{E}_{\hat{\mu}}[\lambda^P h(x_{P+1})|x_1 = i] \geq$$

$$J_{\hat{\mu}}(i) - \frac{M\lambda^P}{1-\lambda} - \max_{j\in\mathcal{S}} |h(j)|\lambda^P.$$

Using Lemma 11.1, the above becomes:

$$J_{\hat{\mu}}(i) + \frac{M\lambda^P}{1-\lambda} + \max_{j\in\mathcal{S}} |h(j)|\lambda^P \geq T^P_{\hat{\mu}} h(i) \geq J_{\hat{\mu}}(i) - \frac{M\lambda^P}{1-\lambda} - \max_{j\in\mathcal{S}} |h(j)|\lambda^P. \tag{11.19}$$

We take the limit with $P \to \infty$; since $\lim_{P\to\infty} \lambda^P = 0$, we then have from Theorem 9.7 that

$$J_{\hat{\mu}}(i) \geq \lim_{P\to\infty} T^P_{\hat{\mu}} h(i) \geq J_{\hat{\mu}}(i). \tag{11.20}$$

Clearly (11.20) implies from Theorem 9.8 that:

$$\lim_{P\to\infty} T^P_{\hat{\mu}} h(i) = J_{\hat{\mu}}(i) \text{ for all } i \in \mathcal{S}. \quad \blacksquare$$

The result implies that associated with any policy $\hat{\mu}$, there is a value function vector denoted by $\vec{J}_{\hat{\mu}}$ that can be obtained by applying the transformation $T_{\hat{\mu}}$ on any given vector infinitely many times. Also note the following:

- The ith element of this vector denotes the expected total discounted reward earned over an infinite time horizon, if one starts at state i and follows policy $\hat{\mu}$.

- In contrast, the ith element of the vector $\vec{J}^*$ denotes the expected total discounted reward earned over an infinite time horizon, if one starts at state i and follows the **optimal policy**.

We now formally present the optimality of the Bellman equation.

PROPOSITION 11.7 *Consider the system of equations defined by:*

$$h(i) = \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \lambda \sum_{j \in \mathcal{S}} p(i,a,j)h(j) \right] \quad \textit{for all } i \in \mathcal{S}, \tag{11.21}$$

where $\vec{h} \in \Re^{|\mathcal{S}|}$. Using the shorthand notation introduced previously, the equation can be expressed as:

$$h(i) = Th(i) \quad \textit{for all } i \in \mathcal{S} \ .$$

The optimal value function vector, $\vec{J}^*$, which is defined in Definition 11.2, is a solution of this equation. Furthermore, $\vec{J}^*$ is the **unique** *solution of this equation.*

Proof The proof is immediate from Proposition 11.5 ▮

Note that in the system of equations presented above, the h terms are the unknowns. Hence, finding the solution to this equation holds the key to finding the optimal solution to the MDP. The above result implies that a policy, $\hat{\mu}$, which satisfies

$$T_{\hat{\mu}}\vec{h} = T\vec{h},$$

for any given vector, $\vec{h}$, is an optimal policy.

We now formalize the result related to the Bellman policy equation.

PROPOSITION 11.8 *The Bellman equation for discounted reward* **for a given policy**, *$\hat{\mu}$, is a system of linear equations defined by:*

$$h(i) = \bar{r}(i,\mu(i)) + \lambda \sum_{j \in \mathcal{S}} p(i,\mu(i),j)h(j) \quad \textit{for all } i \in \mathcal{S}, \tag{11.22}$$

where $h : \mathcal{S} \to \Re$ is a bounded function. Using our shorthand notation, introduced previously,

$$h(i) = T_{\hat{\mu}}h(i) \quad \textit{for all } i \in \mathcal{S} \ .$$

The value function vector associated to policy $\hat{\mu}$, $\vec{J}_{\hat{\mu}}$, which is defined in Definition 11.1, is in fact a solution of the Bellman equation given above. Furthermore, $\vec{J}_{\hat{\mu}}$ is the **unique** *solution.*

Proof Proposition 11.4 implies that $T_{\hat{\mu}}$ is contractive, which means from Theorem 9.10 that the solution is unique, and that the solution must equal $\lim_{k\to\infty} T^k_{\hat{\mu}}$, which from Proposition 11.6 must be $\vec{J}_{\hat{\mu}}$. ∎

3.2. Policy Iteration

In this subsection, we will prove that policy iteration for discounted reward generates the optimal solution to the MDP. For details of the policy iteration algorithm, the reader should review Sect. 5.4 of Chap. 6. We present the core of the algorithm below since we use slightly different notation here.

Step 2. Policy Evaluation: Solve the following linear system of equations.

$$v^k(i) = \bar{r}(i, \mu_k(i)) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i, \mu_k(i), j) v^k(j).$$

Here one linear equation is associated with each value of i. In this system, the unknowns are the v^k terms.

Step 3. Policy Improvement: Choose a new policy $\hat{\mu}_{k+1}$ such that

$$\mu_{k+1}(i) \in \arg\max_{a \in \mathcal{A}(i)} \left[\bar{r}(i, a) + \lambda \sum_{j=1}^{|\mathcal{S}|} p(i, a, j) v^k(j) \right].$$

If possible, one should set $\hat{\mu}_{k+1} = \hat{\mu}_k$.

Proposition 11.9 *The policy iteration algorithm described above generates an optimal solution in a finite number of iterations.*

Proof (Convergence of Policy Iteration) The equation in Step 2 of the algorithm can be written as:

$$v^k(i) = T_{\hat{\mu}_k} v^k(i) \tag{11.23}$$

for all i. A careful examination of Step 3 will reveal that

$$T_{\hat{\mu}_{k+1}} v^k(i) = T v^k(i) \tag{11.24}$$

for all i. Thus for every i,

$$\begin{aligned} v^k(i) &= T_{\hat{\mu}_k} v^k(i) \quad \text{(from Eq. (11.23))} \\ &\leq T v^k(i) \quad \text{(follows from the fact that } T \text{ is a max operator)} \end{aligned}$$

$$= \quad T_{\hat{\mu}_{k+1}} v^k(i) \quad \text{(from Eq. (11.24))}.$$

Thus, for every i, $v^k(i) \leq T_{\hat{\mu}_{k+1}} v^k(i)$. Then for every i, using the monotonicity result from Sect. 2.2, it follows that

$$T_{\hat{\mu}_{k+1}} v^k(i) \leq T^2_{\hat{\mu}_{k+1}} v^k(i)$$

which implies that:

$$v^k(i) \leq T_{\hat{\mu}_{k+1}} v^k(i) \leq T^2_{\hat{\mu}_{k+1}} v^k(i).$$

Repeatedly applying $T_{\hat{\mu}_{k+1}}$, one has that for all i,

$$v^k(i) \leq T_{\hat{\mu}_{k+1}} v^k(i) \leq T^2_{\hat{\mu}_{k+1}} v^k(i) \leq \ldots \leq T^P_{\hat{\mu}_{k+1}} v^k(i).$$

Since the above is also true when $P \to \infty$, one has that for all i,

$$v^k(i) \leq \lim_{P \to \infty} T^P_{\hat{\mu}_{k+1}} v^k(i).$$

From Proposition 11.6, we know that $\lim_{P\to\infty} T^P_{\hat{\mu}_{k+1}} v^k(i)$ exists and equals $J_{\hat{\mu}_{k+1}}(i)$, where $J_{\hat{\mu}_{k+1}}(.)$ is the value function of policy $\hat{\mu}_{k+1}$. Hence

$$v^k(i) \leq J_{\hat{\mu}_{k+1}}(i) \text{ for all } i. \tag{11.25}$$

Now, by Proposition 11.8, $J_{\hat{\mu}_{k+1}}(i)$ satisfies

$$J_{\hat{\mu}_{k+1}}(i) = T_{\hat{\mu}_{k+1}} J_{\hat{\mu}_{k+1}}(i) \text{ for all } i. \tag{11.26}$$

But from Step 2 of the algorithm it follows that:

$$v^{k+1}(i) = T_{\hat{\mu}_{k+1}} v^{k+1}(i) \text{ for all } i. \tag{11.27}$$

From Eqs. (11.26) and (11.27), it is clear that both vectors $\vec{v}^{\,k+1}$ and $\vec{J}_{\hat{\mu}_{k+1}}$ satisfy the equation

$$\vec{h} = T_{\hat{\mu}} \vec{h}.$$

But the solution of this equation is unique by Proposition 11.8. Hence:

$$\vec{v}^{\,k+1} = \vec{J}_{\hat{\mu}_{k+1}}.$$

Therefore, from (11.25), one can now write that:

$$v^k(i) \leq v^{k+1}(i) \text{ for all } i. \tag{11.28}$$

This means that in each iteration (k) the value of the vector ($\vec{v}^k$) either increases or does not change. This iterative process cannot go on infinitely as the total number of policies is finite (since the number of states and the number of actions are finite). In other words, the process must terminate in a finite number of steps. When the policy repeats, i.e., when $\mu_k(i) = \mu_{k+1}(i)$ for all i, it is the case that one has obtained the optimal solution. Here is why:

$$v^k(i) = T_{\hat{\mu}_k} v^k(i) = T_{\hat{\mu}_{k+1}} v^k(i) = T v^k(i)$$

for all i. The first equality sign follows from (11.23) and the last equality sign follows from (11.24). Thus: $v^k(i) = Tv^k(i)$ for all i. In other words, the Bellman **optimality** equation has been solved. By Proposition 11.7, we have that $\hat{\mu}_k$ **is** the optimal policy. ∎

This proof can be found in Vol II of [30] and is as clever as the algorithm itself. Convergence of policy iteration has been established in many different ways in the literature. The proof presented above is relatively "easy" but perhaps rather long.

3.3. Value Iteration

Unlike policy iteration, value iteration, the way we have described it, takes an infinite number of iterations to converge in theory. Fortunately, there is a way to work around this. For all practical purposes, it can generate a policy that is very close to optimal. In this context, we will define a so-called **ϵ-optimal** policy, which is the best we can do with our version of value iteration.

A straightforward argument that can be used to prove the so-called ϵ-convergence of value iteration stems from Proposition 11.5. It implies that if you can take any finite-valued vector and use the mapping T on it an infinite number of times, you have the optimal value function vector. Once we have the "optimal" value function vector, since we know it satisfies the Bellman optimality equation, we can find the maximizing action in each state by using the Bellman optimality equation; the latter is the last step in value iteration. Thus, one way to show the convergence of value iteration is to use Proposition 11.5. However, the fact is that we can run our algorithm in practice for only a *finite number of times*, and as such we analyze this issue further.

We will prove that value iteration can produce a policy which can be as close to the optimal policy as one wishes. One way to determine how close is close enough is to use the *norm* of the *difference* of the value function vectors generated in successive iterations of the algorithm.

The norm of the difference vector is an estimate of how close the two vectors are. When we have performed infinitely many iterations, we have that the vectors generated in two successive iterations are *identical*. (Because $\vec{J}^* = T\vec{J}^*$.)

To prove the ϵ-convergence of value iteration, we need Proposition 11.3, which showed that the Bellman transformation is contractive. At this point, the reader should review steps in the value iteration algorithm for discounted reward MDPs in Sect. 5.5 of Chap. 6.

Proposition 11.10 **(Convergence of Value Iteration:)** *The value iteration algorithm generates an ϵ-optimal policy; i.e., for any $\epsilon > 0$, if $\vec{J}_{\hat{d}}$ denotes the value function vector associated with the policy (solution) $\hat{d}$ generated at the end of value iteration, and $\vec{J}^*$ denotes the optimal reward vector, then:*

$$||\vec{J}_{\hat{d}} - \vec{J}^*|| < \epsilon.$$

Proof $\vec{J}_{\hat{d}}$ denotes the vector associated with the policy $\hat{d}$. Note that this vector is never really calculated in the value iteration algorithm. But it is the reward vector associated with the solution policy returned by the algorithm. From Proposition 11.6, we know that $\lim_{P\to\infty} T_{\hat{d}}\vec{h}$ converges for any $\vec{h} \in \Re^{|\mathcal{S}|}$ and that the limit is $\vec{J}_{\hat{d}}$. From Proposition 11.8, we know that

$$T_{\hat{d}}\vec{J}_{\hat{d}} = \vec{J}_{\hat{d}}. \tag{11.29}$$

Now from Step 4, from the way $\hat{d}$ is selected, it follows that for any vector $\vec{h}$,

$$T_{\hat{d}}\vec{h} = T\vec{h}. \tag{11.30}$$

We will use this fact below. It follows from the properties of norms (see Sect. 3 of Chap. 9) that:

$$||\vec{J}_{\hat{d}} - \vec{J}^*|| \leq ||\vec{J}_{\hat{d}} - \vec{J}^{k+1}|| + ||\vec{J}^{k+1} - \vec{J}^*||. \tag{11.31}$$

$$\begin{aligned}
\text{Now, } ||\vec{J}_{\hat{d}} - \vec{J}^{k+1}|| &= ||T_{\hat{d}}\vec{J}_{\hat{d}} - \vec{J}^{k+1}|| \quad \text{using (11.29)} \\
&= ||T_{\hat{d}}\vec{J}_{\hat{d}} - T\vec{J}^{k+1} + T\vec{J}^{k+1} - \vec{J}^{k+1}|| \\
&\leq ||T_{\hat{d}}\vec{J}_{\hat{d}} - T\vec{J}^{k+1}|| + ||T\vec{J}^{k+1} - \vec{J}^{k+1}|| \\
&\qquad \text{using a property of a norm} \\
&\leq ||T_{\hat{d}}\vec{J}_{\hat{d}} - T\vec{J}^{k+1}|| + ||T\vec{J}^{k+1} - T\vec{J}^{k}|| \\
&= ||T\vec{J}_{\hat{d}} - T\vec{J}^{k+1}|| + ||T\vec{J}^{k+1} - T\vec{J}^{k}|| \quad \text{using (11.30)} \\
&\leq \lambda||\vec{J}_{\hat{d}} - \vec{J}^{k+1}|| + \lambda||\vec{J}^{k+1} - \vec{J}^{k}|| \\
&\quad \text{using the contraction property of } T.
\end{aligned}$$

$$\text{Rearranging terms, } ||\vec{J}_{\hat{d}} - \vec{J}^{k+1}|| \leq \frac{\lambda}{1-\lambda}||\vec{J}^{k+1} - \vec{J}^{k}||. \tag{11.32}$$

Similarly,

$$\begin{aligned}
||\vec{J}^{k+1} - \vec{J}^{*}|| &\leq ||\vec{J}^{k+1} - T\vec{J}^{k+1}|| + ||T\vec{J}^{k+1} - \vec{J}^{*}|| \\
&\quad \text{using a standard norm property} \\
&= ||T\vec{J}^{k} - T\vec{J}^{k+1}|| + ||T\vec{J}^{k+1} - \vec{J}^{*}|| \\
&\leq \lambda||\vec{J}^{k} - \vec{J}^{k+1}|| + ||T\vec{J}^{k+1} - \vec{J}^{*}|| \\
&\leq \lambda||\vec{J}^{k} - \vec{J}^{k+1}|| + ||T\vec{J}^{k+1} - T\vec{J}^{*}|| \quad \text{since } T\vec{J}^{*} = \vec{J}^{*} \\
&\leq \lambda||\vec{J}^{k} - \vec{J}^{k+1}|| + \lambda||\vec{J}^{k+1} - \vec{J}^{*}||.
\end{aligned}$$

$$\text{Rearranging terms, } ||\vec{J}^{k+1} - \vec{J}^{*}|| \leq \frac{\lambda}{1-\lambda}||\vec{J}^{k+1} - \vec{J}^{k}||. \tag{11.33}$$

Using (11.32) and (11.33) in (11.31), one has that $||\vec{J}_{\hat{d}} - \vec{J}^{*}|| \leq 2\frac{\lambda}{1-\lambda}||\vec{J}^{k+1} - \vec{J}^{k}|| < \epsilon$; the last inequality in the above stems from Step 3 of the algorithm. Thus: $||\vec{J}_{\hat{d}} - \vec{J}^{*}|| < \epsilon$. ∎

4. Average Reward DP: MDPs

In this section, we will present a number of results related to average reward in the context of classical dynamic programming. Our discussion will be more or less consistent with the approach used in the discounted reward sections. We will begin with the optimality of the Bellman equation and then go on to discuss the convergence of value and policy iteration methods.

4.1. Bellman Equation for Average Reward

We have defined the "average reward" of a policy heuristically in Chap. 6, and have also expressed it in terms of the steady-state (limiting) probabilities of the underlying Markov chain. We now present a more technical definition.

Definition 11.3 *The average reward of a policy $\hat{\mu}$ is defined as:*

$$\rho_{\hat{\mu}}(i) \equiv \liminf_{k\to\infty} \frac{\mathsf{E}_{\hat{\mu}}\left[\sum_{s=1}^{k} r(x_s, \mu(x_s), x_{s+1}) \middle| x_1 = i\right]}{k}$$

for any $i \in \mathcal{S}$, when i is the starting state in the trajectory.

Note that in the above definition, the average reward is defined in terms of the starting state i. Fortunately, when the Markov chain of the policy $\hat{\mu}$ is regular, the average reward is the same for every starting state, and then, we have that:

$$\rho_{\hat{\mu}} \equiv \rho_{\hat{\mu}}(i) \text{ for any } i \in \mathcal{S}.$$

It is thus important to recognize that when the Markov chain of the policy is regular, we have a unique value for its average reward that does not depend on the starting state. The maximum attainable value for the average reward over all admissible policies is the optimal average reward, denoted by ρ^*. We now turn our attention to the Bellman equation.

The Bellman equations for average reward were used without proof in previous chapters. Our first result below is a key result related to the Bellman equations: both the optimality and the policy versions. Our analysis will be under some conditions that we describe later. The first part of the result will prove that if a solution exists to the Bellman policy equation for average reward, then the scalar ρ in the equation will equal the average reward of the policy in question. The second part will show that if a solution exists to the Bellman optimality equation, then the unknown scalar in the equation will equal the average reward of the optimal policy and that the optimal policy will be identifiable from the solution of the equation.

Note that we will assume that a solution exists to the Bellman equation. That solutions exist to these equations can be proved, but it is beyond our scope here. The interested reader is referred to [30, 242] for proofs.

Proposition 11.11 *(i) If a scalar ρ and a $|S|$-dimensional vector $\vec{h}_{\hat{\mu}}$, where $|\mathcal{S}|$ denotes the number of elements in the set of states in the Markov chain, $\mathcal{S}$, satisfy*

$$\rho + h_{\hat{\mu}}(i) = \bar{r}(i, \mu(i)) + \sum_{j=1}^{|\mathcal{S}|} p(i, \mu(i), j) h_{\hat{\mu}}(j), \quad i = 1, 2, \ldots, |\mathcal{S}|, \tag{11.34}$$

then ρ is the average reward associated to the policy $\hat{\mu}$ defined in Definition 11.3.

(ii) Assume that one of the stationary, deterministic policies is optimal. If a scalar ρ^* and a $|\mathcal{S}|$-dimensional vector $\vec{J^*}$ satisfy

$$\rho^* + J^*(i) = \max_{u \in \mathcal{A}(i)} \left[\bar{r}(i, u) + \sum_{j=1}^{|\mathcal{S}|} p(i, u, j) J^*(j) \right], \quad i = 1, 2, \ldots, |\mathcal{S}|, \tag{11.35}$$

where $\mathcal{A}(i)$ is the set of allowable actions in state i, then ρ^* equals the average reward associated to the policy $\hat{\mu}^*$ that attains the max in the RHS of Eq. (11.35). Further, the policy $\hat{\mu}^*$ is an optimal policy.

Note that the assumption we made above in part (ii)—that one of the optimal policies is stationary and deterministic—is quite strong. But analysis without this condition is beyond our scope; again, for that, the interested reader is referred to [242, 30].

Proof Equation (11.34) can be written in vector form as:

$$\rho\vec{e} + \vec{h}_{\hat{\mu}} = L_{\hat{\mu}}(\vec{h}_{\hat{\mu}}), \tag{11.36}$$

where $L_{\hat{\mu}}$ is a mapping associated with policy $\hat{\mu}$ and $\vec{e}$ is an $|\mathcal{S}|$-dimensional vector of ones; that is each component of $\vec{e}$ is 1. In other words, $\vec{e}$ is

$$\begin{bmatrix} 1 \\ 1 \\ . \\ . \\ . \\ 1 \end{bmatrix}.$$

We will first prove that

$$L^k_{\hat{\mu}} h_{\hat{\mu}}(i) = k\rho + h_{\hat{\mu}}(i) \tag{11.37}$$

for $i = 1, 2, \ldots, |\mathcal{S}|$.

The above can be written in the vector form as:

$$L^k_{\hat{\mu}}\left(\vec{h}_{\hat{\mu}}\right) = k\rho\vec{e} + \vec{h}_{\hat{\mu}}. \tag{11.38}$$

We will use an induction argument for the proof. From Eq. (11.36), the above is true when $k = 1$. Let us assume that the above is true when $k = m$. Then, we have that

$$L^m_{\hat{\mu}}\left(\vec{h}_{\hat{\mu}}\right) = m\rho\vec{e} + \vec{h}_{\hat{\mu}}.$$

Using the transformation $L_{\hat{\mu}}$ on both sides of this equation, we have:

$$\begin{aligned} L_{\hat{\mu}}\left(L^m_{\hat{\mu}}\left(\vec{h}_{\hat{\mu}}\right)\right) &= L_{\hat{\mu}}\left(m\rho\vec{e} + \vec{h}_{\hat{\mu}}\right) \\ &= m\rho\vec{e} + L_{\hat{\mu}}\left(\vec{h}_{\hat{\mu}}\right) \end{aligned}$$

$$= m\rho\vec{e} + \rho\vec{e} + \vec{h}_{\hat{\mu}} \text{ (using Eq. (11.36))}$$
$$= (m+1)\rho\vec{e} + \vec{h}_{\hat{\mu}}.$$

Thus Eq. (11.38) is established using induction on k.

Using Lemma 11.2, we have for all i:

$$L^k_{\hat{\mu}} h_{\hat{\mu}}(i) = \mathsf{E}_{\hat{\mu}}\left[A + \sum_{s=1}^{k} r(x_s, \mu(x_s), x_{s+1})|x_1 = i\right],$$

where x_s is the state from where the sth jump of the Markov chain occurs and A is a finite scalar.

Using the above and Eq. (11.37), we have that:

$$\mathsf{E}_{\hat{\mu}}\left[A + \sum_{s=1}^{k} r(x_s, \mu(x_s), x_{s+1})|x_1 = i\right] = k\rho + h_{\hat{\mu}}(i).$$

Therefore,

$$\frac{\mathsf{E}_{\hat{\mu}}[A]}{k} + \frac{\mathsf{E}_{\hat{\mu}}[\sum_{s=1}^{k} r(x_s, \mu(x_s), x_{s+1})|x_1 = i]}{k} = \rho + \frac{h_{\hat{\mu}}(i)}{k}.$$

Taking limits as $k \to \infty$, we have:

$$\lim_{k\to\infty} \frac{\mathsf{E}_{\hat{\mu}}[\sum_{s=1}^{k} r(x_s, \mu(x_s), x_{s+1})|x_1 = i]}{k} = \rho.$$

(The above follows from the fact that $\lim_{k\to\infty} X/k = 0$, if $X \in \Re$ is finite.) In words, this means from the definition of average reward that the average reward of the policy $\hat{\mu}$ is indeed ρ, and the first part of the proposition is thus established.

Now for the second part. Using the first part of the proposition, one can show that a policy, let us call it $\hat{\mu}^*$, which attains the max in the RHS of Eq. (11.35), produces an average reward of ρ^*.

We will now show that any stationary and deterministic policy that deviates from $\hat{\mu}^*$ will produce an average reward lower than or equal to ρ^*. This will establish, under our assumption that one of the stationary deterministic policies has to be optimal, that the policy $\hat{\mu}^*$ will generate the maximum possible value for the average reward and *will therefore be an optimal policy.* Thus, all we need to show is that a policy, $\hat{\mu}$, which does not necessarily attain the max in Eq. (11.35), produces an average reward less than or equal to ρ^*.

Equation (11.35) can be written in vector form as:

$$\rho^*\vec{e} + \vec{J^*} = L(\vec{J^*}). \tag{11.39}$$

We will first prove that:

$$L^k_{\hat{\mu}}\left(\vec{J^*}\right) \leq k\rho^*\vec{e} + \vec{J^*}. \tag{11.40}$$

As before, we use an induction argument. Now, from Eq. (11.39)

$$L(\vec{J^*}) = \rho^*\vec{e} + \vec{J^*}.$$

But we know that

$$L_{\hat{\mu}}(\vec{J^*}) \leq L(\vec{J^*}),$$

which follows from the fact that any given policy may not attain the max in Eq. (11.35). Thus:

$$L_{\hat{\mu}}\left(\vec{J^*}\right) \leq \rho^*\vec{e} + \vec{J^*}. \tag{11.41}$$

This proves that Eq. (11.40) holds for $k = 1$. Assuming that it holds when $k = m$, we have that:

$$L^m_{\hat{\mu}}\left(\vec{J^*}\right) \leq m\rho^*\vec{e} + \vec{J^*}.$$

Using the fact that $L_{\hat{\mu}}$ is monotonic from the results presented in Sect. 2.2, it follows that

$$\begin{aligned} L_{\hat{\mu}}\left(L^m_{\hat{\mu}}\left(\vec{J^*}\right)\right) &\leq L_{\hat{\mu}}\left(m\rho^*\vec{e} + \vec{J^*}\right) \qquad (11.42)\\ &= m\rho^*\vec{e} + L_{\hat{\mu}}\left(\vec{J^*}\right)\\ &\leq m\rho^*\vec{e} + \rho^*\vec{e} + \vec{J^*} \text{ (using Eq. (11.41))}\\ &= (m+1)\rho^*\vec{e} + \vec{J^*}. \end{aligned}$$

This establishes Eq. (11.40). The following bears similarity to the proof of the first part of this proposition. Using Lemma 11.2, we have for all i:

$$L^k_{\hat{\mu}}J^*(i) = \mathsf{E}_{\hat{\mu}}\left[A + \sum_{s=1}^{k} r(x_s, \mu(x_s), x_{s+1})|x_1 = i\right],$$

where x_s is the state from which the sth jump of the Markov chain occurs and A is a finite scalar. Using this and Eq. (11.40), we have that:

$$\mathsf{E}_{\hat{\mu}}\left[A + \sum_{s=1}^{k} r(x_s, \mu(x_s)), x_{s+1}|x_1 = i\right] \leq k\rho^* + J^*(i).$$

Therefore,

$$\frac{\mathsf{E}_{\hat{\mu}}[A]}{k} + \frac{\mathsf{E}_{\hat{\mu}}[\sum_{s=1}^{k} r(x_s, \mu(x_s), x_{s+1})|x_1 = i]}{k} \leq \rho^* + \frac{J^*(i)}{k}.$$

Taking the limits with $k \to \infty$, we have:

$$\lim_{k\to\infty} \frac{\mathsf{E}_{\hat{\mu}}[\sum_{s=1}^{k} r(x_s, \mu(x_s), x_{s+1})|x_1 = i]}{k} \leq \rho^*.$$

(As before, the above follows from $\lim_{k\to\infty} X/k = 0$ for finite X.) In words, this means that the average reward of the policy $\hat{\mu}$ is less than or equal to ρ^*. This implies that the policy that attains the max in the RHS of Eq. (11.35) is indeed the optimal policy since no other policy can beat it. ▮

4.2. Policy Iteration

In this subsection, we will prove that policy iteration can generate an optimal solution to the average reward problem. We will assume that all states are recurrent under every allowable policy.

To prove the convergence of policy iteration under the condition of **recurrence**, we need Lemmas 11.12 and 11.13. Lemma 11.12 is a general result related to the steady-state (limiting) probabilities of a policy to its transition probabilities in the context of any function on the state space. This lemma will be needed to prove Lemma 11.13, which in turn will be needed in our main result.

LEMMA 11.12 *Let $\Pi_{\hat{\mu}}(i)$ denote the limiting probability of the ith state in a Markov chain run by the policy $\hat{\mu}$. If $p(i, \mu(i), j)$ denotes the element in the ith row and jth column of the transition probability matrix of the Markov chain underlying policy $\hat{\mu}$*

$$\sum_{i=1}^{|\mathcal{S}|} \Pi_{\hat{\mu}}(i) \left[\sum_{j=1}^{|\mathcal{S}|} p(i, \mu(i), j) h(j) - h(i) \right] = 0,$$

where $\vec{h}$ is any finite-valued, $|\mathcal{S}|$-dimensional vector.

Proof From Theorem 6.2 on page 133, $\vec{\Pi}\mathbf{P} = \vec{\Pi}$ one can write that for all $j \in \mathcal{S}$:

$$\sum_{i\in\mathcal{S}} \Pi_{\hat{\mu}}(i) p(i, \mu(i), j) = \Pi_{\hat{\mu}}(j),$$

which can be written as: $\displaystyle\sum_{i\in\mathcal{S}} \Pi_{\hat{\mu}}(i) p(i, \mu(i), j) - \Pi_{\hat{\mu}}(j) = 0.$

Hence: $\sum_{i \in \mathcal{S}} \Pi_{\hat{\mu}}(i)p(i,\mu(i),j)h(i) - \Pi_{\hat{\mu}}(j)h(i) = 0 \quad$ for all $i,j \in \mathcal{S}$.

Then summing the LHS of the above equation over all j, one obtains:

$$\sum_{j \in \mathcal{S}} \left[\sum_{i \in \mathcal{S}} \Pi_{\hat{\mu}}(i)p(i,\mu(i),j)h(i) - \Pi_{\hat{\mu}}(j)h(i) \right] = 0. \tag{11.43}$$

Now Eq. (11.43), by suitable rearrangement of terms, can be written as:

$$\sum_{i \in \mathcal{S}} \Pi_{\hat{\mu}}(i) \sum_{j \in \mathcal{S}} p(i,\mu(i),j)h(j) - \sum_{i \in \mathcal{S}} \Pi_{\hat{\mu}}(i)h(i) = 0.$$

This establishes Lemma 11.12. If the last step is not clear, see the Remark below. ∎

Remark. The last step in the lemma above can be *verified* for a two-state Markov chain. From $\vec{\Pi}\mathbf{P} = \vec{\Pi}$, if $\mathbf{P}$ denotes the transition probability matrix for policy $\hat{\mu}$, we have that:

$$\Pi(1)P(1,1) + \Pi(2)P(2,1) = \Pi(1);$$
$$\Pi(1)P(1,2) + \Pi(2)P(2,2) = \Pi(2).$$

(Note that in the above, $P(i,j) \equiv p(i,\mu(i),j)$.) Multiplying both sides of the first equation by $h(1)$ and those of the second by $h(2)$ and then adding the resulting equations one has that:

$$\Pi(1)P(1,1)h(1) + \Pi(2)P(2,1)h(1) + \Pi(1)P(1,2)h(2) + \Pi(2)P(2,2)h(2)$$
$$= \Pi(1)h(1) + \Pi(2)h(2).$$

This can be written, after rearranging the terms, as:

$$\Pi(1)P(1,1)h(1) + \Pi(1)P(1,2)h(2) + \Pi(2)P(2,1)h(1) + \Pi(2)P(2,2)h(2)$$
$$- \Pi(1)h(1) - \Pi(2)h(2) = 0,$$

which can be written as:

$$\sum_{i=1}^{2} \Pi(i) \sum_{j=1}^{2} p(i,\mu(i),j)h(j) - \sum_{i=1}^{2} \Pi(i)h(i) = 0.$$

This should explain the last step of the proof of Lemma 11.12.

The reader should now review the steps in policy iteration on page 152. We now present Lemma 11.13 which will be used directly in our main result that proves the convergence of policy iteration. It will show that the average reward in every iteration of policy iteration is non-diminishing.

LEMMA 11.13 *If ρ^k denotes the average reward (determined using the policy evaluation step) in the kth iteration of the policy iteration algorithm, then*

$$\rho^{k+1} \geq \rho^k.$$

Proof We know from our discussions in Chap. 6 that the **average reward of a policy** $\hat{\mu}$ can be written as:

$$\sum_{i \in \mathcal{S}} \Pi_{\hat{\mu}}(i) \bar{r}(i, \mu(i))$$

where $\vec{\Pi}_{\hat{\mu}}$ denotes the limiting probability vector of the Markov chain associated with the policy $\hat{\mu}$.

From Proposition 11.11, we know that the term ρ in the Bellman equation for a policy $\hat{\mu}$ denotes the average reward associated with the policy. Hence we can write an expression for the average reward in the $(k+1)$th iteration of policy iteration as:

$$\begin{aligned}
\rho^{k+1} &= \sum_{i \in \mathcal{S}} \Pi_{\hat{\mu}_{k+1}}(i) \bar{r}(i, \mu_{k+1}(i)) \qquad (11.44) \\
&= \rho^k - \rho^k + \sum_{i \in \mathcal{S}} \Pi_{\hat{\mu}_{k+1}}(i) \bar{r}(i, \mu_{k+1}(i)) \\
&= \rho^k + \sum_{i \in \mathcal{S}} \Pi_{\hat{\mu}_{k+1}}(i) [\bar{r}(i, \mu_{k+1}(i)) - \rho^k] \\
&= \rho^k + \sum_{i \in \mathcal{S}} \Pi_{\hat{\mu}_{k+1}}(i) [\bar{r}(i, \mu_{k+1}(i)) - \rho^k] \\
&\quad + \sum_{i \in \mathcal{S}} \Pi_{\hat{\mu}_{k+1}}(i) \left[\sum_{j \in \mathcal{S}} p(i, \mu_{k+1}(i), j) h^k(j) - h^k(i) \right] \\
&\text{(using Lemma 11.12)} \\
&= \rho^k + \sum_{i \in \mathcal{S}} \Pi_{\hat{\mu}_{k+1}}(i) \times \\
&\quad \left[\bar{r}(i, \mu_{k+1}(i)) - \rho^k + \sum_{j \in \mathcal{S}} p(i, \mu_{k+1}(i), j) h^k(j) - h^k(i) \right].
\end{aligned}$$

Now, from the policy improvement step, it is clear that $\hat{\mu}_{k+1}$ is chosen in a way such that:

$$\bar{r}(i,\mu_{k+1}(i))+\sum_{j\in\mathcal{S}}p(i,\mu_{k+1}(i),j)h^k(j)\geq\bar{r}(i,\mu_k(i))+\sum_{j\in\mathcal{S}}p(i,\mu_k(i),j)h^k(j) \tag{11.45}$$

which implies that for each i:

$$\bar{r}(i,\mu_{k+1}(i))+\sum_{j\in\mathcal{S}}p(i,\mu_{k+1}(i),j)h^k(j)-\rho^k-h^k(i)\geq$$

$$\bar{r}(i,\mu_k(i))+\sum_{j\in\mathcal{S}}p(i,\mu_k(i),j)h^k(j)-\rho^k-h^k(i).$$

But the policy evaluation stage of the policy iteration algorithm implies that the RHS of the above inequality equals 0. Thus for each i:

$$\bar{r}(i,\mu_{k+1}(i))+\sum_{j\in\mathcal{S}}p(i,\mu_{k+1}(i),j)h^k(j)-\rho^k-h^k(i)\geq 0. \tag{11.46}$$

Since, $\Pi(i)\geq 0$ for any policy and i, we can write from the above that:

$$\Pi_{\hat{\mu}_{k+1}}(i)\left[\bar{r}(i,\mu_{k+1}(i))+\sum_{j\in\mathcal{S}}p(i,\mu_{k+1}(i),j)h^k(j)-\rho^k-h^k(i)\right]\geq 0.$$

Summing over all values of $i\in\mathcal{S}$ and then adding ρ^k to both sides of the resulting inequation, we have:

$$\rho^k+\sum_{i\in\mathcal{S}}\Pi_{\hat{\mu}_{k+1}}(i)\left[\bar{r}(i,\mu_{k+1}(i))+\sum_{j\in\mathcal{S}}p(i,\mu_{k+1}(i),j)h^k(j)-\rho^k-h^k(i)\right]\geq\rho^k,$$

which using the last equation in (11.44) leads to: $\rho^{k+1}\geq\rho^k$. ∎

The next proposition will establish the convergence of policy iteration under the assumption of recurrence.

PROPOSITION 11.14 *The policy $\hat{\mu}^*$ generated by the policy iteration algorithm for average reward MDPs is an optimal policy if all states are recurrent.*

Proof From Lemma 11.13, the sequence of average rewards is a strictly increasing sequence until a policy repeats if all states are recurrent. This is because notice that if the policy chosen in in the policy improvement step is different from the previous policy, then the $\geq$ in (11.45) is replaced by $>$; since $\Pi(i) > 0$ for every i if all states are recurrent, this will lead to $\rho^{k+1} > \rho^k$. Since the number of states and actions is finite, there is a finite number of policies, and the convergence criterion $\hat{\mu}_{k+1} = \hat{\mu}_k$ must be satisfied at a finite value of k. When the policy repeats, we have that $\hat{\mu}_{k+1} = \hat{\mu}_k$ which means from the policy improvement and the policy evaluation steps that the policy $\hat{\mu}_k$ satisfies the Bellman equation. From Proposition 11.11, this implies that $\hat{\mu}_k$ is the optimal policy. ∎

4.3. Value Iteration

Convergence analysis of value iteration for average reward is more complicated than its counterpart for discounted reward because the Bellman optimality equation cannot be used directly; this is because ρ^*, the average reward of the optimal policy, is not known in advance (since obviously the optimal policy is not known in advance). A way to work around this difficulty is to set ρ^* to 0 or some constant. It is okay to do this in MDPs since the values that the resulting Bellman optimality equation (with a replaced ρ^*) generates differ from those generated by the actual Bellman optimality equation by a constant value, and this allows one to generate the policy that would have been obtained by using the Bellman optimality equation (with the true value of ρ^*).

A major difficulty with average reward value iteration, as discussed in previous chapters, is that due to the lack of a discounting factor, the associated mapping is not contractive, and hence, the max norm cannot be used for stopping the algorithm. In addition, some of the iterates (or values) become unbounded, i.e., they converge to negative or positive infinity. One way around this is to subtract a predetermined element of the value vector in Step 2 of value iteration. This ensures that the values themselves remain bounded. The resulting algorithm is called relative value iteration.

A function called the **span seminorm** (or simply the span) can become very useful in identifying the ϵ-optimal policy in this situation. Convergence of **regular** value iteration using a span semi-norm has been presented under a specific condition in [242, Theor. 8.5.2; pp. 368]. An elegant proof for convergence of *relative* value iteration *under a different condition* can be found in Vol II of [30], but it uses a norm (not a semi-norm). In this book, we will present a proof from [119] for the convergence of regular value iteration *and* relative value iteration using the span semi-norm, under yet another condition on the transition probability matrices involved.

In value iteration for *discounted* reward MDPs, we had convergence in the norm, i.e., with successive iterations of the algorithm, the norm of the so-called "difference vector" became smaller and smaller. In value iteration for average reward, we will not be able to show that. Instead, we will show convergence in the span seminorm, i.e., the span semi-norm of the "difference vector" will become smaller and smaller with every iteration. First, we quickly review the definition of the span seminorm and regular and relative value iteration algorithms.

Span semi-norm. The span semi-norm (or span) of the vector $\vec{x}$, denoted by $sp(\vec{x})$, is defined as:

$$sp(\vec{x}) = \max_i x(i) - \min_i x(i).$$

Example: $\vec{a} = (9, 2)$ and $\vec{b} = (-3, 8, 16)$. Then $sp(\vec{a}) = 7$ and $sp(\vec{b}) = 19$.

Steps in value iteration for average reward. The presentation is from Chap. 6.

Step 1: Set $k = 1$ and select an arbitrary vector $\vec{J}^1$. Specify $\epsilon > 0$.

Step 2: For each $i \in \mathcal{S}$, compute:

$$J^{k+1}(i) = \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \sum_{j \in \mathcal{S}} p(i,a,j) J^k(j) \right].$$

Step 3: If

$$sp(\vec{J}^{k+1} - \vec{J}^k) < \epsilon,$$

go to Step 4. Otherwise increase k by 1 and go back to Step 2.

Step 4: For each $i \in \mathcal{S}$ choose

$$d(i) \in \arg \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \sum_{j \in \mathcal{S}} p(i,a,j) J^k(j) \right] \tag{11.47}$$

and stop. The ϵ-optimal policy is $\hat{d}$.

Steps in Relative value iteration. We present the steps in a format that makes it easier to analyze the algorithm's convergence.

Step 1: Set $k = 1$, choose any state to be a distinguished state, i^*, and select an arbitrary vector $\vec{W}^1$. Specify $\epsilon > 0$ and set $\vec{v}^1 = \vec{W}^1 - W^1(i^*)\vec{e}$, where $\vec{e}$ is a column vector of ones.

Step 2: For each $i \in \mathcal{S}$, compute:

$$W^{k+1}(i) = \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \sum_{j \in \mathcal{S}} p(i,a,j) v^k(j) \right].$$

Then for each $i \in \mathcal{S}$, set $v^{k+1}(i) = W^{k+1}(i) - W^{k+1}(i^*)$.

Step 3: If $sp(\vec{v}^{k+1} - \vec{v}^k) < \epsilon$, go to Step 4; else increase k by 1 and return to Step 2.

Step 4: For each $i \in \mathcal{S}$ choose

$$d(i) \in \arg\max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \sum_{j \in \mathcal{S}} p(i,a,j) v^k(j) \right].$$

The convergence of regular and relative value iteration will hinge on the following fundamental result, which we state without proof.

Theorem 11.15 *Suppose F is an M-stage span contraction mapping; that is, for any two vectors $\vec{x}$ and $\vec{y}$ in a given vector space, for some positive, finite, and integral value of M,*

$$sp(F^M\vec{x} - F^M\vec{y}) \leq \eta \; sp(\vec{x} - \vec{y}) \text{ for } 0 \leq \eta < 1. \tag{11.48}$$

Consider the sequence $\{\vec{z}^k\}_{k=1}^{\infty}$ defined by: $\vec{z}^{k+1} = F\vec{z}^k = F^{k+1}\vec{z}^0$. Then, there exists a vector $\vec{z}^*$

$$\text{for which } sp(F\vec{z}^* - \vec{z}^*) = 0 \tag{11.49}$$

$$\text{and } \lim_{k \to \infty} sp(\vec{z}^k - \vec{z}^*) = 0. \tag{11.50}$$

Also, given an $\epsilon > 0$, there exists an N such that for all $k \geq N$:

$$sp(\vec{z}^{kM+1} - \vec{z}^{kM}) < \epsilon. \tag{11.51}$$

To prove the convergence of regular value iteration, we need to prove two critical results: Lemmas 11.16 and 11.17. But, first, we need to define the delta coefficients for which we first present some notation.

Let $\mathbf{R}$ be a matrix with non-negative elements such that the elements in each of its rows sum to 1. Let W denote the number of rows in the matrix and C the number of columns; additionally we define two sets: $\mathcal{W} = \{1, 2, \ldots, W\}$ and $\mathcal{C} = \{1, 2, \ldots, C\}$. Now, we define:

$$b_{\mathbf{R}}(i, j, l) = \min_{l \in \mathcal{C}} \{R(i, l), R(j, l)\} \text{ for every } (i, j) \in \mathcal{W} \times \mathcal{W},$$

where $R(i, j)$ denotes the element of the ith row and the jth column in $\mathbf{R}$. Further, we define

$$B_{\mathbf{R}}(i, j) = \sum_{l=1}^{C} b_{\mathbf{R}}(i, j, l) \text{ for every } (i, j) \in \mathcal{W} \times \mathcal{W}.$$

Definition 11.4 *The delta-coefficient of a matrix,* $\mathbf{R}$*, described above, is defined as:*

$$\alpha_{\mathbf{R}} \equiv 1 - \min_{(i,j) \in \mathcal{W} \times \mathcal{W}} B_{\mathbf{R}}(i, j). \tag{11.52}$$

Note that the matrix $\mathbf{R}$ above could be composed of two matrices stacked over each other. For example, consider the following example:

$$\mathbf{R} = \left[\frac{\mathbf{P}_1}{\mathbf{P}_2} \right],$$

where $\mathbf{P}_1$ and $\mathbf{P}_2$ are two matrices. For instance consider these cases:

$$\mathbf{P}_1 = \begin{bmatrix} 0.2 & 0.1 & 0.7 \\ 0.3 & 0.4 & 0.3 \\ 0.8 & 0.1 & 0.1 \end{bmatrix}; \mathbf{P}_2 = \begin{bmatrix} 0.5 & 0.4 & 0.1 \\ 0.2 & 0.3 & 0.5 \\ 0.4 & 0.4 & 0.2 \end{bmatrix}$$

$$\text{Then: } \mathbf{R} = \begin{bmatrix} 0.2 & 0.1 & 0.7 \\ 0.3 & 0.4 & 0.3 \\ 0.8 & 0.1 & 0.1 \\ 0.5 & 0.4 & 0.1 \\ 0.2 & 0.3 & 0.5 \\ 0.4 & 0.4 & 0.2 \end{bmatrix}.$$

Here $b_{\mathbf{R}}(1, 2, 1) = \min\{0.2, 0.3\} = 0.2$; $b_{\mathbf{R}}(1, 2, 2) = \min\{0.1, 0.4\} = 0.1$; and $b_{\mathbf{R}}(1, 2, 3) = \min\{0.7, 0.3\} = 0.3$, which implies that $B_{\mathbf{R}}(1, 2) = 0.2 + 0.1 + 0.3 = 0.6$. In this fashion, all the $B_{\mathbf{R}}(, .,)$ terms can be calculated to compute $\alpha_{\mathbf{R}}$ using Eq. (11.52).

We now present the first of the two critical lemmas that will be essential in showing the convergence of value iteration.

LEMMA 11.16 *Let $\vec{x}$ be any arbitrary column vector with C components and $\mathbf{R}$ be a matrix defined in Definition 11.4. Then, $sp(\mathbf{R}\vec{x}) \leq \alpha_{\mathbf{R}}\ sp(\vec{x})$.*

The proof presented below is from Seneta [269, Theor. 3.1; pp 81].

Proof Let $\vec{R}' = \mathbf{R}\vec{x}$. Since $\vec{x}$ is a column vector with C rows, it is clear that $\mathbf{R}\vec{x}$ is a column vector with as many rows as $\mathbf{R}$. Then for any $(i,j) \in \mathcal{C} \times \mathcal{C}$:

$$R'(i) - R'(j) = \sum_{l \in \mathcal{C}} R(i,l)x(l) - \sum_{l \in \mathcal{C}} R(j,l)x(l)$$

$$= \sum_{l \in \mathcal{C}} [R(i,l) - b_{\mathbf{R}}(i,j,l)]x(l) - \sum_{l \in \mathcal{C}} [R(j,l) - b_{\mathbf{R}}(i,j,l)]x(l)$$

$$\leq \sum_{l \in \mathcal{C}} [R(i,l) - b_{\mathbf{R}}(i,j,l)] \max_{l \in \mathcal{C}} x(l) - \sum_{l \in \mathcal{C}} [R(j,l) - b_{\mathbf{R}}(i,j,l)] \min_{l \in \mathcal{C}} x(l)$$

$$= \left[1 - \sum_{l \in \mathcal{C}} b_{\mathbf{R}}(i,j,l)\right] \left[\max_{l \in \mathcal{C}} x(l) - \min_{l \in \mathcal{C}} x(l)\right]$$

$$= \left[1 - \sum_{l \in \mathcal{C}} b_{\mathbf{R}}(i,j,l)\right] sp(\vec{x}) \leq \left[1 - \min_{i,j} \sum_{l \in \mathcal{C}} b_{\mathbf{R}}(i,j,l)\right] sp(\vec{x}) = \alpha_{\mathbf{R}} sp(\vec{x}).$$

From the above, one can summarize that for any $(i,j) \in \mathcal{C} \times \mathcal{C}$:

$$R'(i) - R'(j) \leq \alpha_{\mathbf{R}} sp(\vec{x}).$$

But $\max_{i \in \mathcal{C}} R'(i) - \min_{i \in \mathcal{C}} R'(i) \leq R'(i) - R'(j)$, which implies that

$$\max_{i \in \mathcal{C}} R'(i) - \min_{i \in \mathcal{C}} R'(i) \leq \alpha_{\mathbf{R}} sp(\vec{x}) \text{ i.e., } sp(\vec{R}') \leq \alpha_{\mathbf{R}} sp(\vec{x}). \quad \blacksquare$$

We will now present the second of the two critical lemmas. But before that, we provide some necessary notation and definitions. If transformation L is applied m times on vector $\vec{z}^k \in \Re^{|\mathcal{S}|}$, the resulting vector will be $\vec{z}^{k+m}$. Further, $L^{k+1}\vec{z} \equiv L\left(L^k \vec{z}\right)$. Also, $\hat{d}_{x^k}$ will denote a deterministic policy, associated with vector $\vec{x}^k \in \Re^{|\mathcal{S}|}$, which will prescribe actions that are determined as follows:

$$\text{For every } i \in \mathcal{S},\ d_{x^k}(i) \in \arg\max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \sum_{j \in \mathcal{S}} p(i,a,j)x^k(j)\right]. \tag{11.53}$$

Thus, if $\vec{y}^k$ is a vector in $\Re^{|\mathcal{S}|}$, using the definition of $L_{\hat{\mu}}$ in Eq. (11.5),

$$L_{d_{x^k}} y^k(i) = \left[\bar{r}(i, d_{x^k}(i)) + \sum_{j\in\mathcal{S}} p(i, d_{x^k}(i), j) y^k(j)\right] \text{ for every } i \in \mathcal{S}.$$

This implies that for every $i \in \mathcal{S}$ and any vector $\vec{x}^k \in \Re^{|\mathcal{S}|}$, using the definition of L in Eq. (11.4),

$$Lx^k(i) \equiv L_{d_{x^k}} x^k(i). \tag{11.54}$$

Lemma 11.17 *Let M be any positive, finite integer. Consider any two vectors $\vec{x}^1$ and $\vec{y}^1$ that have $|\mathcal{S}|$ components. Using $\mathbf{P}_\mu$ to denote the transition probability matrix associated with deterministic policy $\hat{\mu}$, we define two matrices:*

$$\mathbf{A}_x^M \equiv \mathbf{P}_{d_{x^M}} \mathbf{P}_{d_{x^{M-1}}} \ldots \mathbf{P}_{d_{x^1}} \textit{ and } \mathbf{A}_y^M \equiv \mathbf{P}_{d_{y^M}} \mathbf{P}_{d_{y^{M-1}}} \ldots \mathbf{P}_{d_{y^1}}.$$

Then, $sp(L^M\vec{y}^1 - L^M\vec{x}^1) \le \alpha_{\mathbf{A}}\ sp(\vec{y}^1 - \vec{x}^1)$, where $\mathbf{A} \equiv \begin{bmatrix}\mathbf{A}_y^M \\ \mathbf{A}_x^M\end{bmatrix}$.

Proof Let states s^* and s_* be defined as follows:

$$s^* = \arg\max_{s\in\mathcal{S}}\{L^M y^1(s) - L^M x^1(s)\};\ \ s_* = \arg\min_{s\in\mathcal{S}}\{L^M y^1(s) - L^M x^1(s)\}.$$

$$\text{For any } i \in \mathcal{S}, L^M x^1(i) = L_{d_{x^M}} L_{d_{x^{M-1}}} \ldots L_{d_{x^2}} L_{d_{x^1}} x^1(i). \tag{11.55}$$

$$\text{We can show that } L^M y^1(i) \ge L_{d_{x^M}} L_{d_{x^{M-1}}} \ldots L_{d_{x^2}} L_{d_{x^1}} y^1(i) \quad \forall i. \tag{11.56}$$

The above can be proved as follows. From definition, for all $i \in \mathcal{S}$, $Ly^1(i) \ge L_{d_{x^1}} y^1(i)$. Since L is monotonic, for all $i \in \mathcal{S}$, $L\left(Ly^1(i)\right) \ge L\left(L_{d_{x^1}} y^1(i)\right)$. From definition of L, for all $i \in \mathcal{S}$, $L\left(Ly^1(i)\right) \ge L_{d_{x^2}}\left(L_{d_{x^1}} y^1(i)\right)$. From the preceding inequalities, for all $i \in \mathcal{S}$, $L\left(Ly^1(i)\right) \ge L_{d_{x^2}}\left(L_{d_{x^1}} y^1(i)\right)$. In this way, by repeatedly using the monotonicity property, we can establish (11.56). From (11.55) and (11.56), it follows that

$$\begin{aligned}
&L^M y^1(s^*) - L^M x^1(s^*) \\
&\ge [L_{d_{x^M}} L_{d_{x^{M-1}}} \ldots L_{d_{x^1}} y^1(s^*)] - [L_{d_{x^M}} L_{d_{x^{M-1}}} \ldots L_{d_{x^1}} x^1(s^*)] \\
&= [\bar{r}(s^*, d_{x^1}(s^*)) + \bar{r}(s^*, d_{x^2}(s^*)) + \cdots + \bar{r}(s^*, d_{x^M}(s^*)) + \\
&\mathbf{P}_{d_{x^M}} \mathbf{P}_{d_{x^{M-1}}} \ldots \mathbf{P}_{d_{x^1}} y^1(s^*)] - \\
&[\bar{r}(s^*, d_{x^1}(s^*)) + \bar{r}(s^*, d_{x^2}(s^*)) + \cdots + \bar{r}(s^*, d_{x^M}(s^*)) + \\
&\mathbf{P}_{d_{x^M}} \mathbf{P}_{d_{x^{M-1}}} \ldots \mathbf{P}_{d_{x^1}} x^1(s^*)] = \mathbf{A}_x^M (y^1 - x^1)(s^*).
\end{aligned}$$

$$\text{Thus: } L^M y^1(s^*) - L^M x^1(s^*) \leq \mathbf{A}_x^M (y^1 - x^1)(s^*). \tag{11.57}$$

Using logic similar to that used above:

$$L^M y^1(s_*) - L^M x^1(s_*) \leq \mathbf{A}_y^M (y^1 - x^1)(s_*). \tag{11.58}$$

Then $sp(L^M \vec{y}^1 - L^M \vec{x}^1)$

$= \{L^M y^1(s^*) - L^M x^1(s^*)\} - \{L^M y^1(s_*) - L^M x^1(s_*)\}$

$\leq \mathbf{A}_y^M (y^1 - x^1)(s^*) - \mathbf{A}_x^M (y^1 - x^1)(s_*)$ (from (11.57) and (11.58))

$\leq \max_{s \in \mathcal{S}} \mathbf{A}_y^M (y^1 - x^1)(s) - \min_{s \in \mathcal{S}} \mathbf{A}_x^M (y^1 - x^1)(s)$

$\leq \max_{s \in \mathcal{S}} \left[\frac{\mathbf{A}_y^M}{\mathbf{A}_x^M} \right] (y^1 - x^1)(s) - \min_{s \in \mathcal{S}} \left[\frac{\mathbf{A}_y^M}{\mathbf{A}_x^M} \right] (y^1 - x^1)(s)$

$= sp \left(\left[\frac{\mathbf{A}_y^M}{\mathbf{A}_x^M} \right] (\vec{y}^1 - \vec{x}^1) \right)$

$\leq \alpha_{\mathbf{A}}\ sp(\vec{y}^1 - \vec{x}^1)$ (from Lemma 11.16). ∎

The following result from [119] presents the convergence of regular and relative value iteration under a specific condition on the transition probability matrices associated to the MDP.

Assumption 11.18 *Consider two sequences of M deterministic policies, $S_1 = \{\mu_1, \mu_2, \ldots \mu_M\}$ and $S_2 = \{\nu_1, \nu_2, \ldots \nu_M\}$ where M is a positive finite integer. Further, consider the stacked matrix:*

$$\mathbf{A}_{S_1,S_2} \equiv \left[\frac{\mathbf{P}_{\mu_1} \cdot \mathbf{P}_{\mu_2} \cdots \mathbf{P}_{\mu_M}}{\mathbf{P}_{\nu_1} \cdot \mathbf{P}_{\nu_2} \cdots \mathbf{P}_{\nu_M}} \right]. \tag{11.59}$$

Assume that there exists an integral value for $M \geq 1$ such that for every possible pair, (S_1, S_2), the delta coefficient of $\mathbf{A}_{S_1,S_2}$ for the MDP is strictly less than 1.

Theorem 11.19 *Assumption 11.18 holds. Then*

(a) The value iteration converges in the limit to an ϵ-optimal solution.

(b) Further, assume that $\vec{W}^1 = \vec{J}^1$. Then, value and relative value iteration algorithms choose the same sequence of maximizing actions and terminate at the same policy for any given value of ϵ, where ϵ is strictly positive and is used in Step 3 of the algorithms.

Proof(a) Consider the sequence of vectors $\vec{J}^k$ in value iteration. Then: $\vec{J}^{k+1} = L\vec{J}^k$, for all$k = 1, 2, \ldots$ where L is the transformation used in Step 2 of value iteration. The delta-coefficient condition in the assumption above implies that Lemma 11.17 can be used, from which one has that Theorem 11.15 holds for L. It follows from (11.49) then that there exists a $\vec{J}^*$ such that $L\vec{J}^* = \vec{J}^* + \psi_1\vec{e}$ for some scalar ψ_1. The above implies from Proposition 11.11 that $\vec{J}^*$ is an optimal solution of the MDP. However, from (11.50), we know that $\lim_{k\to\infty} \vec{J}^k = \vec{J}^* + \psi_2\vec{e}$ for some scalar ψ_2. From (11.47), then one has that $\vec{J}^*$ and $(\vec{J}^* + \psi_2\vec{e})$ will result in the same policy. Then, it follows from (11.51) that a finite termination rule can be developed with a user-specified value of ϵ.

(b) Let $\vec{v}^k$ denote the iterate vector in the kth iteration of relative value iteration. We will first show that:

$$\vec{v}^k = \vec{J}^k - \sum_{l=1}^{k} \zeta^l \vec{e}, \tag{11.60}$$

where $\vec{J}^k$ denotes the iterate vector of value iteration and ζ^l is some scalar constant, whose value will depend on iteration l, and $\vec{e}$ is a vector of ones. This will help us show that the span of the difference vector in each algorithm is the same. Further, it will also establish that both algorithms choose the same sequence of maximizing actions.

We will use induction on k to show (11.60). Since, $\vec{W}^1 = \vec{J}^1$, it is clear from Step 1 of relative value iteration that: $\vec{v}^1 = \vec{J}^1 - \zeta^1\vec{e}$, where $\zeta^1 = J^1(i^*)$, and hence (11.60) is true for $k = 1$. Assuming it is true for $k = m$, we have that:

$$\vec{v}^m = \vec{J}^m - \sum_{l=1}^{m} \zeta^l \vec{e}. \tag{11.61}$$

Now, from Step 2 of relative value iteration, by setting $\zeta^{m+1} = W^{m+1}(i^*)$, we have that for all $i \in \mathcal{S}$,:

$$\begin{aligned} v^{m+1}(i) &= \max_{j\in\mathcal{S}} \left(\sum_{j\in\mathcal{S}} p(i,a,j)\left[r(i,a,j) + v^m(j)\right] \right) - \zeta^{m+1} \\ &= \max_{j\in\mathcal{S}} \left(\sum_{j\in\mathcal{S}} p(i,a,j)\left[r(i,a,j) + J^m(j) - \sum_{l=1}^{m} \zeta^l \right] \right) \\ &\quad -\zeta^{m+1} \text{ (from (11.61))} \end{aligned}$$

$$= \max_{j \in \mathcal{S}} \left(\sum_{j \in \mathcal{S}} p(i,a,j) \left[r(i,a,j) + J^m(j) \right] \right) - \sum_{l=1}^{m+1} \zeta^l$$

$$= J^{m+1}(i) - \sum_{l=1}^{m+1} \zeta^l \text{ (from Step 2 of value iteration),}$$

from which the induction is complete.

Then, the span of the difference vector in any iteration of both algorithms will be equal, since: $sp\left(\vec{v}^{k+1} - \vec{v}^k\right)$

$$= sp\left(\vec{J}^{k+1} - \vec{J}^k - \zeta^{k+1}\vec{e}\right) \text{ (from (11.60))}$$
$$= sp\left(\vec{J}^{k+1} - \vec{J}^k\right).$$

The two algorithms will choose the same sequence of maximizing actions (see Step 4 in each) since:

$$\arg\max_{j \in \mathcal{S}} \left[\sum_{j \in \mathcal{S}} p(i,a,j) \left[r(i,a,j) + v^k(j) \right] \right]$$
$$= \arg\max_{j \in \mathcal{S}} \left[\sum_{j \in \mathcal{S}} p(i,a,j) \left[r(i,a,j) + J^k(j) - \sum_{l=1}^{k} \zeta^l \right] \right]$$
$$= \arg\max_{j \in \mathcal{S}} \left[\sum_{j \in \mathcal{S}} p(i,a,j) \left[r(i,a,j) + J^k(j) \right] \right].$$

Thus, since the difference vectors have the same span in both algorithms and both algorithms choose the same sequence of maximizing actions, we have that both algorithms terminate at the same policy for a given value of ϵ. ∎

Note that this convergence proof relies on the condition provided in Assumption 11.18. This condition can be easy to show for $M = 1$ in the MDP, but can be difficult to show in practice for larger values of M. However, it is the first result to show convergence of relative value iteration under the span semi-norm, which is often faster than convergence in the norm.

Further, note that we have not proved that the iterates (the elements of the value function vector) will themselves converge; all we have shown is that both the regular and relative versions of the algorithm will terminate with ϵ-optimal solutions when using the span criterion. But, it has been seen in practice that *relative* value iteration also keeps the iterates bounded, which indicates that relative value iteration will not only terminate with an ϵ-optimal solution but will also be numerically stable.

5. DP: SMDPs

Discounted reward SMDPs were only briefly covered in the context of dynamic programming under the general assumption. Hence, we will not cover discounted reward SMDPs in this section; we will discuss them in the context of RL algorithms later. In this section, we focus on average reward SMDPs. We first define average reward of a policy in an SMDP. Thereafter, we present the main result related to the Bellman equations.

DEFINITION 11.5 *The average reward of a policy $\hat{\mu}$ in an SMDP is defined as:*

$$\rho_{\hat{\mu}}(i) \equiv \liminf_{k\to\infty} \frac{\mathsf{E}_{\hat{\mu}}\left[\sum_{s=1}^{k} r(x_s,\mu(x_s),x_{s+1})\Big| x_1=i\right]}{\mathsf{E}_{\hat{\mu}}\left[\sum_{s=1}^{k} t(x_s,\mu(x_s),x_{s+1})\Big| x_1=i\right]}$$

for any $i \in \mathcal{S}$, when i is the starting state in the trajectory.

Like in the MDP case, when the Markov chain of the policy $\hat{\mu}$ is regular, the average reward is the same for every starting state, and then, we have that:

$$\rho_{\hat{\mu}} \equiv \rho_{\hat{\mu}}(i) \text{ for any } i \in \mathcal{S}.$$

We now present the main result for the Bellman equations for average reward SMDPs. This is the SMDP counterpart of Proposition 11.11, and it is presented without proof (the interested reader can find the proof in [30, 242]).

PROPOSITION 11.20 *(i) If a scalar ρ and a $|S|$-dimensional vector $\vec{h}_{\hat{\mu}}$, where $|\mathcal{S}|$ denotes the number of elements in the set of states in the Markov chain, $\mathcal{S}$, satisfy*

$$\rho\bar{t}(i,\mu(i))+h_{\hat{\mu}}(i)=\bar{r}(i,\mu(i))+\sum_{j=1}^{|\mathcal{S}|} p(i,\mu(i),j)h_{\hat{\mu}}(j), \quad i=1,2,\ldots,|\mathcal{S}|, \tag{11.62}$$

then ρ is the average reward associated to the policy $\hat{\mu}$ defined in Definition 11.5.

(ii) Assume that one of the stationary, deterministic policies is optimal. If a scalar ρ^* and a $|\mathcal{S}|$-dimensional vector $\vec{J}^*$ satisfy

$$J^*(i) = \max_{u \in \mathcal{A}(i)} \left[\bar{r}(i,u) - \rho^* \bar{t}(i,u) + \sum_{j=1}^{|\mathcal{S}|} p(i,u,j) J^*(j) \right], \quad i = 1, 2, \ldots, |\mathcal{S}|, \tag{11.63}$$

where $\mathcal{A}(i)$ is the set of allowable actions in state i, then ρ^* equals the average reward associated to the policy $\hat{\mu}^*$ that attains the max in the RHS of Eq. (11.63). Further, the policy $\hat{\mu}^*$ is an optimal policy.

6. Asynchronous Stochastic Approximation

RL (reinforcement learning) schemes are derived from dynamic programming schemes, and yet they need a separate convergence analysis because mathematically the two schemes are quite different. One difference is in the use of the step size in RL, which is absent in the DP schemes. The use of the step size makes any RL scheme belong to the class called "stochastic approximation." The other difference is that RL schemes tend to be "asynchronous," a concept that we will discuss later.

We will discuss the convergence concepts in this section via the ordinary (deterministic) differential equation (ODE) method. It can be shown that many stochastic approximation schemes, e.g., RL algorithms, are *tracked* by ODEs. In other words, it can be shown that we can derive ODEs associated with many stochastic approximation schemes. Properties of the associated ODE can be exploited to study convergence properties of the RL algorithm.

6.1. Asynchronous Convergence

We first discuss the difference between a synchronous and an asynchronous algorithm. The difference is best explained with an example.

Consider a problem with three states, which are numbered 1 through 3. In a synchronous algorithm, one would first go to state 1, update value(s) associated with it, then go to state 2, update its value(s), and then go to state 3 and update its value(s). Then one would return to state 1, and this would continue. In asynchronous updating, the order of updating is haphazard. It could be for instance:

$$1 \to 3 \to 1 \to 2 \to 1 \to \cdots$$

On the other hand, in synchronous updating, the order is always as follows:

$$1 \to 2 \to 3 \to 1 \to 2 \to 3 \cdots .$$

Dynamic programming is usually synchronous (note that the Gauss-Seidel version of value iteration is however an exception and is *partially* asynchronous). The asynchronous use of the RL algorithm makes it quite different from a regular dynamic programming algorithm. There are in fact two differences:

1. The updating in the RL algorithm follows a haphazard order dictated by the trajectory of the sample path, whereas the order used in dynamic programming is not random.

2. When a state is visited, a Q-factor version of a synchronous dynamic programming algorithm would update *all* the Q-factors associated with that state, while an asynchronous RL algorithm would update only one Q-factor, which is the Q-factor associated with the action selected in the previous state.

It is also important to understand that in one iteration of the RL algorithm, we visit one state-action pair and update only one Q-factor as a result (the Q-factor associated with the state-action pair). Each Q-factor can be viewed as an iterate in the asynchronous stochastic approximation algorithm. We will define Θ^k as index of the state-action pair visited in the kth iteration of the algorithm.

The reader is encouraged at this point to review Definition 9.14 from Sect. 11.3 in Chap. 9 before reading any further. Then, our asynchronous algorithm can be described formally as: For $l = 1, 2 \ldots, N$,

$$X^{k+1}(l) = X^k(l) + \alpha^k(l)\left[F(\vec{X}^k)(l) + w^k(l)\right] I(l = \Theta^k), \text{ where} \tag{11.64}$$

- N: the (finite) number of iterates (or state-action pairs)
- $\alpha^k(l)$: the step size for the lth iterate for $l = 1, 2, \ldots, N$
- Θ^k: the index of the iterate updated in the kth iteration

Note that in RL, $l \equiv (i, a)$ and $X^k(l) \equiv Q^k(i, a)$. Further note that the indicator function in Eq. (11.64) implies that in the kth iteration, only one Q-factor is updated, and that the Q-factor which does get updated is the one associated to the state-action pair visited in that iteration. The indicator function will return a 0 for a Q-factor *not* visited in the kth iteration, which will imply that the Q-factors *not* visited in that iteration will not be updated. Thus,

$$\{\Theta^1, \Theta^2, \ldots\}$$

will denote the trajectory of state-action pairs visited in the simulator.

6.1.1 Asynchronous Counterpart of Theorem 9.16

We will now present the asynchronous counterpart of Theorem 9.16 (Chap. 9). Consider the stochastic approximation scheme in Theorem 9.16 *but for asynchronous updating.* In Theorem 9.16, which applies only to synchronous updating, convergence was shown under a number of conditions. Under some additional conditions on the step sizes and the nature of updating, we will now present a result that establishes for asynchronous updating the same conclusion as Theorem 9.16, i.e., convergence of the stochastic approximation scheme. We will present *all* the conditions needed for convergence here (for the reader's convenience), repeating those needed for synchronous convergence.

THEOREM 11.21 *Consider the asynchronous stochastic approximation algorithm defined in Eq. (11.64). Assume that the following conditions hold:*

Condition 1. *Lipschitz continuity of $F(.)$: Assumption 9.11 from Chap. 9: The function $F(.)$ is Lipschitz continuous.*

Condition 2. *Conditions on noise: Assumption 9.12 from Chap. 9: For $l = 1, 2, \ldots, N$ and for every k, the following should be true about the noise terms:*

$$\mathsf{E}\left[w^k(l)|\mathcal{F}^k\right] = 0; \quad \mathsf{E}\left[\left(w^k(l)\right)^2 \middle| \mathcal{F}^k\right] \le z_1 + z_2||\vec{X}^k||^2;$$

where z_1 and z_2 are scalar constants and $||.||$ could be any norm.

Condition 3. *Standard step-size conditions of stochastic approximation: Assumption 9.13 of Chap. 9: The step size $\alpha^k(l)$ satisfies the following conditions for every $l = 1, 2, \ldots, N$:*

$$\sum_{k=1}^{\infty} \alpha^k(l) = \infty; \quad \sum_{k=1}^{\infty} \left(\alpha^k(l)\right)^2 < \infty.$$

Condition 4. *Asymptotically stable critical point for ODE: Assumption 9.15 from Chap. 9: Let $\vec{x}$ denote the continuous-valued variable underlying the iterate $\vec{X}^k$ (discussed in Chap. 9). Then, ODE*

$$\frac{d\vec{x}}{dt} = F(\vec{x})$$

has a unique globally asymptotically stable equilibrium point, denoted by $\vec{x}_*$.

Condition 5. *Boundedness of iterates: Assumption 9.14 from Chap. 9: The iterate $\vec{X}^k$ remains bounded with probability 1.*

Condition 6. *ITS (ideal tapering sizes) conditions on step sizes: For every $l = 1, 2, \ldots, N$,*

(i) $\alpha^{k+1}(l) \le \alpha^k(l)$ *for some k onward*

(ii) For any $z \in (0, 1)$, $\sup_k \alpha^{[zk]}(l)/\alpha^k(l) < \infty$ where $[.]$ denotes "the integer part of."

(iii) For any $z \in (0, 1)$,

$$\lim_{k\to\infty} \frac{\sum_{m=1}^{[zk]+1} \alpha^m(l)}{\sum_{m=1}^{k} \alpha^m(l)} = 1.$$

Condition 7. *EDU (evenly distributed updating) conditions on updating: Let $V^k(l) = \sum_{m=1}^{k} I(l = \Theta^k)$ for $l = 1, 2, \ldots, N$. Then with probability 1, there exists a deterministic scalar $\chi > 0$ such that for all l:*

$$\liminf_{k\to\infty} \frac{V^k(l)}{k} \ge \chi.$$

Further, for any scalar $z > 0$, we define for every l,

$$K^k(z, l) \equiv \min\left\{ m > k : \sum_{s=k+1}^{m} \alpha^s(l) > z \right\}.$$

Then, with probability 1, the following limit must exist for all (l, l') pairs, where each of l and l' assumes values from $\{1, 2, \ldots, N\}$, and any $z > 0$:

$$\lim_{k\to\infty} \frac{\sum_{m=V^k(l)}^{V^{K^k(z,l)}(l)} \alpha^k(l)}{\sum_{m=V^k(l')}^{V^{K^k(z,l')}(l')} \alpha^k(l')}.$$

Then, the sequence $\{\vec{X}^k\}_{k=1}^{\infty}$ converges to $\vec{x}_*$ with probability 1.

The above is a powerful result with a deep proof that we skip. The interested reader is referred for the proof to Borkar [46] (see also Chapter 7 in [48], [49, Theorems 2.2 and 2.5], and [136, Theorem 1]).

Fortunately, showing the additional conditions of asynchronous convergence, i.e., Condition 6 (ITS) and Condition 7 (EDU), does not require additional work in RL: Standard step sizes, e.g., $\frac{A}{B+k}$ and $\frac{log(k)}{k}$, satisfy the ITS condition, and when our updating ensures that each state-action pair is visited infinitely often in the limit, the EDU condition is satisfied. Usually, a judicious choice of exploration ensures that each state-action pair is tried infinitely often. Thus, although these conditions appear formidable to prove, unless one selects non-standard step sizes or non-standard exploration, they are automatically satisfied.

Conditions 1 and 2 are usually straightforward to show in RL. Condition 1 is usually easy to show for the Q-factor version of the Bellman equation. Condition 2 is usually satisfied for most standard RL algorithms based on the notion of one-step updates. Condition 3 is satisfied by our standard step sizes, but note that the condition is defined in terms of a separate step size for each iterate (l) unlike in the synchronous case. Now, Condition 3 can be easily met with a separate step size for each iterate, but this would significantly increase our storage burden. Fortunately, it can be shown (see e.g., Chap. 7 in [48], p. 80) that a single step size for all iterates which is updated after every iteration (i.e., whenever k is incremented) also satisfies this condition, although it makes the resulting step size random. That it becomes random poses no problems in our analysis. This leads to the following:

Important note: When we will use Theorem 11.21 in analyzing specific RL algorithms, we will drop l from our notation for the step size and denote the step size by α^k, since we will be assuming that a single step size is used for all values of l.

It is necessary to point out that showing Condition 4 (asymptotic stability of the ODE's equilibrium) and Condition 5 (boundedness of iterate) usually require additional work and they should never be assumed to hold automatically. In fact, much of the analysis of an RL algorithm's convergence may hinge on establishing these two conditions for the algorithm concerned.

6.1.2 Useful Results for Conditions 4 and 5

We now provide some results that can help us show that Conditions 4 and 5 hold.

A Result for Condition 4. The existence of a unique globally asymptotically stable equilibrium can be shown for the ODE in question if the underlying transformation that drives the stochastic approximation scheme is contractive with respect to the weighted max norm. Contractive transformations were defined in Chap. 9, and in the Appendix, you will find a definition of contractions with weighted max norms. Contraction with the max norm is a special case of contraction with the weighted max norm in which all weights equal 1. Formally, the result is as follows.

THEOREM 11.22 *Consider the transformation $F(.)$ defined above for the stochastic approximation scheme. Define $F'(.)$, a continuous function from $\Re^N$ to $\Re^N$, such that for any integer value of k,*

$$F\left(\vec{X}^k\right) = F'\left(\vec{X}^k\right) - \vec{X}^k \text{ where } \vec{X}^k \in \Re^N.$$

If $F'(.)$ is a contractive transformation with respect to a weighted max norm, the ODE $\frac{d\vec{x}}{dt} = F(\vec{x})$ has a unique globally asymptotically stable equilibrium point.

The proof of the above is beyond the scope of this text, and the interested reader is referred to [48, p. 127]. The above is a very useful condition that will be used if $F'(.)$ is contractive. When the contraction is not present, showing this condition can require significant additional mathematics.

A Result for Condition 5. We now present an important condition for boundedness of iterates that can be often shown when Condition 4 is true and some other conditions hold. As such, Condition 4 is a very critical condition in the analysis. We first need to define a so-called "scaled" function.

DEFINITION 11.6 *Consider a function $\Lambda : \Re^N \to \Re$. For any scalar c where $c > 0$, the scaled function $\Lambda_c : \Re^N \to \Re$ is defined as:*

$$\Lambda_c(\vec{x}) = \frac{\Lambda(c\vec{x})}{c} \text{ where } \vec{x} \in \Re^N.$$

We provide a simple example to illustrate this idea. Let $\Lambda(x) = 5x+9$, where $x \in \Re$. Then,

$$\Lambda_c(x) = \frac{\Lambda(cx)}{c} = \frac{5cx+9}{c} = 5x + \frac{9}{c}.$$

We now define a limit for this scaled function. Assuming that

$$\lim_{c\to\infty} \Lambda_c(\vec{x}) \text{ exists, we will name the limit as follows: } \Lambda_\infty(\vec{x}) \equiv \lim_{c\to\infty} \Lambda_c(\vec{x}).$$

Then, in the example above, $\Lambda_\infty(x) = \lim_{c\to\infty}\left(5x + \frac{9}{c}\right) = 5x$.

We now provide the condition needed for showing boundedness.

THEOREM 11.23 *Assume that Conditions 1–3 hold for the stochastic approximation scheme concerned. Consider the scaled function $F_c(.)$ derived from $F(.)$ for $c > 0$. Now assume that the following limit exists for the scaled function: $F_\infty(.)$. Further assume that the origin in $\Re^N$ is the globally asymptotically stable equilibrium for the ODE:*

$$\frac{d\vec{x}}{dt} = F_\infty(\vec{x}). \tag{11.65}$$

Then, with probability 1, the iterates $\vec{X}^k$ remain bounded.

The above result turns out to be very useful in showing the boundedness of iterates in many RL algorithms. Again, its proof is beyond our scope here; the interested reader is referred to [49, Theorem 2.1] or [48, Theorem 7; Chap. 3].

An interesting fact related to the above is that if (i) Condition 4 holds and (ii) the scaled limit $F_\infty(.)$ exists, then oftentimes in reinforcement learning, the ODE in Eq. (11.65) does indeed have the origin as the globally asymptotically stable equilibrium, i.e., Condition 5 also holds. However, all of this needs to be carefully verified separately for every algorithm.

6.2. Two-Time-Scale Convergence

The analysis in this subsection will be useful in showing convergence of R-SMART. *Readers not interested in R-SMART can skip this subsection without loss of continuity.*

We will now consider a more involved setting for stochastic approximation in which we have two classes of iterates, and each class uses a different step-size rule. If the two step-size rules satisfy some conditions, in addition to a host of other conditions that the updating schemes for the two classes of iterates must satisfy, we have convergence of both classes of iterates. It is important to note that the fate of the two classes is inter-twined. This is because the updating scheme for each class uses values of iterates from the other class. Of course, if they were independent of each other, their convergence could be studied separately, but that is not the case here.

The question that arises naturally is as follows: what is to be gained by separating the iterates into two classes and selecting different step size rules for each class? The answer is with two separate step sizes, one may potentially obtain convergence for *both* classes, which may be difficult to attain otherwise.

One class of iterates operates under a step size that converges to zero faster than the other. The one that converges to zero faster is said to belong to the **slower** time scale, while the other is said to belong to the **faster** time scale.

The framework described above is called the two-time-scale framework. It has been popular in electrical engineering for many years. However, it was a result from Borkar [45] that established conditions for convergence and provided it with a solid footing. This framework is useful for showing convergence of R-SMART. In what follows, we will present the framework more formally.

Let $\vec{X}^k$ denote the vector (class) of iterates on the faster time scale and $\vec{Y}^k$ that on the slower time scale. We will assume that we have N_1 iterates on the faster time scale and N_2 iterates on the slower time scale. Further, we assume that the underlying random process in the system generates within the simulator two trajectories,

$$\{\Theta_1^1, \Theta_1^2, \ldots\} \text{ and } \{\Theta_2^1, \Theta_2^2, \ldots\},$$

where Θ_1^k denotes the iterate from the faster time scale updated in the kth iteration while Θ_2^k denotes the iterate from the slower time scale updated in the kth iteration. Thus for $k = 1, 2, \ldots$:

$$\Theta_1^k \in \{1, 2, \ldots, N_1\} \text{ and } \Theta_2^k \in \{1, 2, \ldots, N_2\}.$$

The two-time-scale algorithm under asynchronous updating can now be described as: For $l = 1, 2 \ldots, N_1$ and $l_2 = 1, 2, \ldots, N_2$:

$$X^{k+1}(l_1) = X^k(l_1) + \alpha^k(l_1)\left[F\left(\vec{X}^k, \vec{Y}^k\right)(l_1) + w_1^k(l_1)\right] I(l_1 = \Theta_1^k);$$
$$Y^{k+1}(l_2) = Y^k(l_2) + \beta^k(l_2)\left[G\left(\vec{X}^k, \vec{Y}^k\right)(l_2) + w_2^k(l_2)\right] I(l_2 = \Theta_2^k); \tag{11.66}$$

where

- $\alpha^k(.)$ and $\beta^k(.)$ are the step sizes for the faster and slower time-scale iterates respectively
- $F(.,.)$ and $G(.,.)$ denote the transformations driving the faster and slower time-scale updates respectively
- $\vec{w}_1^k$ and $\vec{w}_2^k$ denote the noise vectors in the kth iteration on the faster and slower time scales respectively

Note that $F(.,.)$ is a function of X and Y, and the same is true of $G(.,.)$. Clearly, hence, the fates of the iterates are intertwined (or coupled) because their values are inter-dependent.

We now present the conditions of convergence and the convergence result more formally. The conditions will follow the format used in Theorem 11.21.

THEOREM 11.24 *Consider the two-time-scale asynchronous algorithm defined in Eq. (11.66). Assume that the following conditions hold:*

Condition 1. *Lipschitz continuity of the underlying transformations: The functions $F(.,.)$ and $G(.,.)$ are Lipschitz continuous.*

Condition 2. *Conditions on noise: For $l_1 = 1, 2, \ldots, N_1$, $l_2 = 1, 2, \ldots, N_2$, and for every k, the following should be true about the noise terms:*

$$\mathsf{E}\left[w_1^k(l_1)|\mathcal{F}^k\right]=0;\quad \mathsf{E}\left[\left(w_1^k(l_1)\right)^2\middle|\mathcal{F}^k\right] \leq z_1+z_2||\vec{X}^k||^2+z_3||\vec{Y}^k||^2;$$

$$\mathsf{E}\left[w_2^k(l_2)|\mathcal{F}^k\right]=0;\quad \mathsf{E}\left[\left(w_2^k(l_2)\right)^2\middle|\mathcal{F}^k\right] \leq z_1'+z_2'||\vec{X}^k||^2+z_3'||\vec{Y}^k||^2;$$

where z_1, z_2, z_3, z_1', z_2', and z_3' are scalar constants and $||.||$ could be any norm.

Condition 3. *Step-size conditions of stochastic approximation: The step sizes satisfy the usual tapering size conditions for every $l_1 = 1, 2, \ldots, N_1$ and $l_2 = 1, 2, \ldots, N_2$:*

$$\sum_{k=1}^{\infty}\alpha^k(l_1)=\infty;\quad \sum_{k=1}^{\infty}\left(\alpha^k(l_1)\right)^2<\infty;$$

$$\sum_{k=1}^{\infty}\beta^k(l_2)=\infty;\quad \sum_{k=1}^{\infty}\left(\beta^k(l_2)\right)^2<\infty;$$

In addition, the step sizes must satisfy the following condition for every (l_1, l_2) pair:

$$\limsup_{k\to\infty}\frac{\beta^k(l_2)}{\alpha^k(l_1)}=0; \tag{11.67}$$

Condition 4a. *ODE condition: For any fixed value of $\vec{y} \in \Re^{N_2}$, the ODE*

$$\frac{d\vec{x}}{dt}=F(\vec{x},\vec{y}) \tag{11.68}$$

has a globally asymptotically stable equilibrium point which is a function of $\vec{y}$ and will be denoted by $\Omega(\vec{y})$, where $\Omega : \Re^{N_2} \to \Re^{N_1}$. Further, the function $\Omega(\vec{y})$ has to be Lipschitz continuous in $\vec{y}$.

Condition 4b. *ODE condition: The following ODE*

$$\frac{d\vec{y}}{dt} = G\left(\Omega(\vec{y}), \vec{y}\right) \tag{11.69}$$

has a globally asymptotically stable equilibrium $\vec{y}_*$.

Condition 5. *Boundedness of iterates: The iterates $\vec{X}^k$ and $\vec{Y}^k$ remain bounded with probability 1.*

Condition 6. *ITS Condition: The step sizes satisfy Condition 6 (ITS) of Theorem 11.21.*

Condition 7. *EDU Condition: The updating of all components of the X- and the Y-iterates is evenly distributed as discussed in Condition 7 of Theorem 11.21.*

Then, with probability 1, the sequence of iterates $\left\{\vec{X}^k, \vec{Y}^k\right\}_{k=1}^{\infty}$ converges to $(\Omega(\vec{y}_*), \vec{y}_*)$.

While the result appears formidable, the underlying intuition can be *roughly* summarized as follows. Assume the following:

(i) The step sizes obey the following condition:

$$\lim_{k\to\infty} \frac{\beta^k}{\alpha^k} = 0;$$

(ii) If the values of the Y-iterates are frozen (i.e., fixed at any vector), the X-iterates converge (to a solution that is Lipschitz continuous in $\vec{Y}$);

(iii) The Y-iterates converge.

Then, if all the iterates remain bounded, the sequence $\{\vec{X}^k, \vec{Y}^k\}_{k=1}^{\infty}$ will converge.

Essentially, what the above implies is that under suitable conditions on the step sizes and the updating (read exploration), which can clearly be controlled by the compute program, if one can show that the faster iterate (X^k) converges when the slower iterate (Y^k) is fixed and the slower iterate can be shown to converge to some value, then we can hope for convergence of the slower *and* the faster iterates.

It is important to note that the conditions that distinguish two-time-scale behavior from the one-time-scale setting of the previous subsection are Conditions 4a and 4b, and the condition on the step sizes in Eq. (11.67). To gain additional insight into the relation between one time-scale and two-time-scale convergence, note that if you fix $\vec{y}$, the result above essentially becomes equivalent to Theorem 11.21, i.e., that for one-time-scale convergence. Finally, note that like in the one-time-scale setting, we will drop l_1 and l_2 from the step sizes when we use the result above, since we will use one step size (α^k) for all the faster iterates and one (β^k) for the slower iterates (this implies that α^k and β^k are updated whenever k is incremented). Like in the case of one-time-scale convergence, this makes the step sizes random but since the random step sizes satisfy all our conditions on step sizes, they pose no difficulties to us. (As stated above, using separate step sizes for each iterate is something to be avoided because that would immensely increase our computational burden.)

7. Reinforcement Learning: Convergence Background

In this section, we shall establish some fundamental results on which the convergence theory of many RL algorithms rests. These results are essentially the Q-factor versions of the basic results related to the Bellman optimality and the Bellman policy (Poisson) equations. These results will be needed in the sections that follow in showing convergence of RL algorithms based on the Bellman optimality and policy equations. We will first cover the Q-factor results for discounted reward MDPs, then those for average reward MDP, followed by those for discounted reward SMDPs, and then finally those for average reward SMDPs.

Important Note. In what follows in this chapter, although the Q-factors are essentially stored in matrices, we will assume that the matrix will be converted into a column vector after suitable mapping. This can always be done. We will refer to the resulting column vector as $\vec{Q}$, the Q-vector. For instance, consider the Q-matrix:

$$\mathbf{Q} = \begin{bmatrix} Q(1,1) & Q(1,2) \\ Q(2,1) & Q(2,2) \end{bmatrix}.$$

We can write the above as a vector, e.g.,

$$\vec{Q} = \begin{bmatrix} Q(1,1) \\ Q(1,2) \\ Q(2,1) \\ Q(2,2) \end{bmatrix}.$$

We will assume that each Q-factor is assigned a unique index in the vector $\vec{Q}$.

7.1. Discounted Reward MDPs

In this section, we will present the Q-factor versions of the Bellman optimality and policy equations for discounted reward MDPs. We first cover the Bellman *optimality* equation.

Bellman optimality equation. Consider Eq. (11.21) in Proposition 11.7. Now, we define the Q-factors as follows: For all (i,a) pairs,

$$Q(i,a) = \left[\bar{r}(i,a) + \lambda \sum_{j\in\mathcal{S}} p(i,a,j) J^*(j)\right], \tag{11.70}$$

where $\vec{J^*} \in \Re^{|\mathcal{S}|}$ and $\vec{J^*}$ is the unique solution of Equation (11.21). Then, from Eq. (11.21), we have that for any $i \in \mathcal{S}$:

$$J^*(i) = \max_{a\in\mathcal{A}(i)} Q(i,a), \text{ which from Eq. (11.70) implies that}$$

$$\text{for all } (i,a) \text{ pairs, } Q(i,a) = \sum_{j\in\mathcal{S}} p(i,a,j) \left[r(i,a,j) + \lambda \max_{b\in\mathcal{A}(j)} Q(j,b)\right].$$

The above is a Q-factor version of the Bellman optimality equation in (11.21). Thus the above equation can be replaced in Proposition 11.7 to obtain the following result.

PROPOSITION 11.25 *Consider the system of equations defined as follows. For all $i \in \mathcal{S}$ and $a \in \mathcal{A}(i)$:*

$$Q(i,a) = \sum_{j\in\mathcal{S}} p(i,a,j) \left[r(i,a,j) + \lambda \max_{b\in\mathcal{A}(j)} Q(j,b)\right].$$

The vector $\vec{Q^*}$ that solves this equation is the optimal Q-factor vector, i.e., if for all $i \in \mathcal{S}$, $\mu^*(i) \in \arg\max_{a\in\mathcal{A}(i)} Q^*(i,a)$, then $\hat{\mu}^*$ denotes an optimal policy.

Bellman policy equation. Now, we will consider the Bellman policy (Poisson) equation. Consider a policy $\hat{\mu}$ and the following definition for the Q-factor: We define the Q-factors as follows: For all (i,a) pairs,

$$Q(i,a) = \left[\bar{r}(i,a) + \lambda \sum_{j\in\mathcal{S}} p(i,a,j) J_{\hat{\mu}}(j)\right], \tag{11.71}$$

where $\vec{J_{\hat{\mu}}} \in \Re^{|\mathcal{S}|}$ and $\vec{J_{\hat{\mu}}}$ is the unique solution of the linear Equation (11.22). From Eq. (11.22), then we have that for any $i \in \mathcal{S}$:

$$J_{\hat{\mu}}(i) = Q(i,\mu(i)), \text{ which from Eq. (11.71) implies that}$$

$$\text{for all } (i,a) \text{ pairs, } Q(i,a) = \sum_{j\in\mathcal{S}} p(i,a,j)\left[r(i,a,j) + \lambda Q(j,\mu(j))\right].$$

The above is a Q-factor version of the Bellman policy equation in (11.22). Thus the above equation can be replaced in Proposition 11.8 to obtain the following result.

PROPOSITION 11.26 *Consider a policy $\hat{\mu}$ and the system of equations defined as follows. For all $i \in \mathcal{S}$ and $a \in \mathcal{A}(i)$:*

$$Q(i,a) = \sum_{j\in\mathcal{S}} p(i,a,j)\left[r(i,a,j) + \lambda Q(j,\mu(j))\right]. \tag{11.72}$$

Then, there exists a unique solution, $\vec{Q}_{\hat{\mu}}$, to the above equation. Further, if

$$J_{\hat{\mu}}(i) = Q_{\hat{\mu}}(i,\mu(i)) \textit{ for all } i \in \mathcal{S},$$

then the vector $\vec{J}_{\hat{\mu}}$ equals the vector that can be obtained by applying transformation $T_{\hat{\mu}}$ on any finite-valued vector infinitely many times, i.e., $\vec{J}_{\hat{\mu}}$ is the value-function vector associated to policy $\hat{\mu}$.

7.2. Average Reward MDPs

In this section, we will present the Q-factor versions of the Bellman optimality and policy equations for average reward MDPs. We first cover the Bellman *optimality* equation.

Bellman optimality equation. Consider Proposition 11.11 and the vector $\vec{J^*}$ and the scalar ρ^* associated to Eq. (11.35). We can write Eq. (11.35) as: For all $i \in \mathcal{S}$:

$$J^*(i) = \max_{a\in\mathcal{A}(i)}\left[\bar{r}(i,a) + \sum_{j\in\mathcal{S}} p(i,a,j)J^*(j) - \rho^*\right]. \tag{11.73}$$

Now, if we define $Q(i,a)$ as follows:

$$Q(i,a) = \left[\bar{r}(i,a) + \sum_{j\in\mathcal{S}} p(i,a,j)J^*(j) - \rho^*\right],$$

where $\vec{J^*}$ and ρ^* together solve Eq. (11.35), then, we have from the above and Eq. (11.73) that for any $i \in \mathcal{S}$:

$$J^*(i) = \max_{\mathcal{A}(i)} Q(i,a), \text{ which from Eq. (11.73) implies that}$$

$$\text{for all } (i,a) \text{ pairs, } Q(i,a) = \left[\bar{r}(i,a) + \sum_{j\in\mathcal{S}} p(i,a,j) \max_{b\in\mathcal{A}(j)} Q(j,b) - \rho^*\right].$$

The above is a Q-factor version of the Bellman optimality equation in (11.35). The above equation can be replaced in the second part of Proposition 11.11 to obtain the following result.

PROPOSITION 11.27 *Consider the following system of equations defined as follows. For all $i \in \mathcal{S}$ and $a \in \mathcal{A}(i)$, and for any scalar $\rho \in \Re$:*

$$Q(i,a) = \sum_{j\in\mathcal{S}} p(i,a,j) \left[r(i,a,j) + \max_{b\in\mathcal{A}(j)} Q(j,b) - \rho\right]. \tag{11.74}$$

If there exist a vector $\vec{Q}^*$ and a scalar $\rho^* \in \Re$ that together satisfy the above equation, then ρ^* is the average reward of the policy $\hat{\mu}^*$, where for all $i \in \mathcal{S}$, $\mu^*(i) \in \arg\max_{a\in\mathcal{A}(i)} Q^*(i,a)$. Further, $\hat{\mu}^*$ denotes an optimal policy.

Bellman policy equation. Consider Proposition 11.11, a policy $\hat{\mu}$, a vector $\vec{h}_{\hat{\mu}} \in \Re^{|\mathcal{S}|}$, and a scalar ρ. Now, if we define $Q(i,a)$ as follows:

$$Q(i,a) = \left[\bar{r}(i,a) + \sum_{j\in\mathcal{S}} p(i,a,j) h_{\hat{\mu}}(j) - \rho\right],$$

where $\vec{h}_{\hat{\mu}}$ and ρ are solutions of Equation (11.34), then, we have from the above and Eq. (11.34) that for any $i \in \mathcal{S}$:

$$h_{\hat{\mu}}(i) = Q(i,\mu(i)), \text{ which from the equation above implies that}$$

$$\text{for all } (i,a) \text{ pairs, } Q(i,a) = \left[\bar{r}(i,a) + \sum_{j\in\mathcal{S}} p(i,a,j) Q(j,\mu(j)) - \rho\right].$$

The above is a Q-factor version of the Bellman policy equation in (11.34). The above equation can be replaced in the first part of Proposition 11.11 to obtain the following result.

PROPOSITION 11.28 *Consider a policy $\hat{\mu}$. Further, consider the following system of equations defined as follows. For all $i \in \mathcal{S}$ and $a \in \mathcal{A}(i)$, and for any scalar $\rho \in \Re$:*

$$Q(i,a) = \sum_{j\in\mathcal{S}} p(i,a,j) \left[r(i,a,j) + Q(j,\mu(j)) - \rho\right].$$

If there exist a vector, $\vec{Q}_{\hat{\mu}}$, and a scalar $\rho \in \Re$ that satisfy the above equation, then ρ is the average reward of the policy $\hat{\mu}$.

8. Reinforcement Learning for MDPs: Convergence

In this section, we shall apply results from the previous section to analyze a subset of RL algorithms for solving MDPs that we have covered in Chap. 7. This will include algorithms belonging to the Q-Learning, Q-P-Learning, non-optimistic API and R-SMART classes. This section is rather long, and each subsection is devoted to the analysis of one algorithm.

8.1. Q-Learning: Discounted Reward MDPs

We will now prove that Q-Learning for discounted reward MDPs converges under asynchronous conditions. The core of the Q-Learning algorithm can be expressed by the following transformation:

$$Q^{k+1}(i,a) \leftarrow Q^k(i,a) + \alpha \left[r(i,a,\xi^k) + \lambda \max_{b \in \mathcal{A}(\xi^k)} Q^k(\xi^k, b) - Q^k(i,a) \right], \tag{11.75}$$

where ξ^k is a random variable that depends on (i,a) and k. Notice that in the past, we have used j in place of ξ^k. The reason for using more technically correct notation here is that ξ^k is a **random** variable. Let the policy generated in the kth iteration be defined by:

$$\mu^k(i) \in \arg\max_{a \in \mathcal{A}(i)} Q^k(i,a) \text{ for all } i \in \mathcal{S}.$$

Let the optimal policy be denoted by $\hat{\mu}^*$. The convergence result for Q-Learning is as follows.

PROPOSITION 11.29 *When the step sizes and action selection used in the algorithm satisfy Conditions 3, 6, and 7 of Theorem 11.21, with probability 1, the sequence of policies generated by the Q-Learning algorithm, $\{\hat{\mu}^k\}_{k=1}^{\infty}$, converges to $\hat{\mu}^*$.*

Proof In order to invoke Theorem 11.21, we first need to show that this algorithm is of the form described in that result. To this end, we first need to define some transformations.

We define the transformations $F(.)$ and $F'(.)$ on the vector $\vec{Q}^k$ as follows:

$$F\left(\vec{Q}^k\right)(i,a) = \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) \left[r(i,a,j) + \lambda \max_{b \in \mathcal{A}(j)} Q^k(j,b) \right] - Q^k(i,a);$$

$$F'\left(\vec{Q}^k\right)(i,a) = \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) \left[r(i,a,j) + \lambda \max_{b \in \mathcal{A}(j)} Q^k(j,b) \right];$$

which implies that for all (i,a) pairs,

$$F\left(\vec{Q}^k\right)(i,a) = F'\left(\vec{Q}^k\right)(i,a) - Q^k(i,a). \tag{11.76}$$

We further define a transformation $f'(.)$ as follows (roughly speaking, the transformation $f'(.)$ contains the sample of which $F'(.)$ computes the expectation):

$$f'\left(\vec{Q}^k\right)(i,a) = \left[r(i,a,\xi^k) + \lambda \max_{b\in\mathcal{A}(\xi^k)} Q^k(\xi^k,b)\right].$$

Now, if we define the noise term as:

$$w^k(i,a) = f'\left(\vec{Q}^k\right)(i,a) - F'\left(\vec{Q}^k\right)(i,a), \tag{11.77}$$

then, we can write the updating transformation in our algorithm (Eq. (11.75)) as:

$$Q^{k+1}(i,a) = Q^k(i,a) + \alpha^k\left[f'\left(\vec{Q}^k\right)(i,a) - Q^k(i,a)\right].$$

Now, using the relation defined in Eq. (11.76), we can re-write the above:

$$Q^{k+1}(i,a) = Q^k(i,a) + \alpha^k\left[F\left(\vec{Q}^k\right)(i,a) + w^k(i,a)\right],$$

which is of the same form as the updating scheme defined for Theorem 11.21 (replace X^k by Q^k and l by (i,a)). Then, we can invoke the following ODE as in Condition 4 of Theorem 11.21:

$$\frac{d\vec{q}}{dt} = F(\vec{q}), \tag{11.78}$$

where $\vec{q}$ denotes the continuous-valued variable underlying the iterate Q. We now need to evaluate the conditions of Theorem 11.21 in order to determine whether Q-Learning converges.

Condition 1. Lipschitz continuity of $F(.)$ can be shown from the fact that the partial derivatives $F(\vec{Q}^k)$ with $Q^k(i,a)$ are bounded. See Definition 9.13 and the related discussion on page 313.

Condition 2. From its definition it is intuitively clear that the noise term $w^k(.,.)$ is the difference between a random value and a conditional mean of the value. Using rigorous mathematical arguments that are beyond our scope here the noise term can be shown to be a martingale, whose conditional mean is zero. Its (conditional) second moment can be bounded by a function of the square of the iterate.

Condition 3. These conditions can be proved for a step size such as $A/(B+k)$. (See any standard undergraduate analysis text such as Rudin [254] for a proof.)

Condition 4. To show this condition, we will exploit Theorem 11.22. Consider two vectors $\vec{Q}_1^k$ and $\vec{Q}_2^k$ in $\Re^N$. From the definition of $F'(.)$ above, it follows that:

$$F'\left(\vec{Q}_1^k\right)(i,a) - F'\left(\vec{Q}_2^k\right)(i,a) = \lambda \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) \left[\max_{b\in\mathcal{A}(j)} Q_1^k(j,b) - \max_{b\in\mathcal{A}(j)} Q_2^k(j,b)\right].$$

From this, we can write that for any (i,a) pair:

$$\begin{aligned}\left|F'\left(\vec{Q}_1^k\right)(i,a) - F'\left(\vec{Q}_2^k\right)(i,a)\right| &\le \lambda \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) \left|\max_{b\in\mathcal{A}(j)} Q_1^k(j,b) - \max_{b\in\mathcal{A}(j)} Q_2^k(j,b)\right| \\ &\quad \text{(from triangle inequality; page 285)} \\ &\le \lambda \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) \max_{b\in\mathcal{A}(j)} |Q_1^k(j,b) - Q_2^k(j,b)| \\ &\le \lambda \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) \max_{j\in\mathcal{S}, b\in\mathcal{A}(j)} |Q_1^k(j,b) - Q_2^k(j,b)| \\ &= \lambda \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) ||\vec{Q}_1^k - \vec{Q}_2^k||_\infty \\ &= \lambda ||\vec{Q}_1^k - \vec{Q}_2^k||_\infty \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) = \lambda ||\vec{Q}_1^k - \vec{Q}_2^k||_\infty \cdot 1.\end{aligned} \tag{11.81}$$

Thus for any (i,a) : $|F'\left(\vec{Q}_1^k\right)(i,a) - F'\left(\vec{Q}_2^k\right)(i,a)| \le \lambda ||\vec{Q}_1^k - \vec{Q}_2^k||_\infty$.

Since the above holds for all values of (i,a), it also holds for the values that maximize the left hand side of the above. Therefore

$$||F'\vec{Q}_1^k - F'\vec{Q}_2^k||_\infty \le \lambda ||\vec{Q}_1^k - \vec{Q}_2^k||_\infty.$$

Since $\lambda < 1$, $F'(.)$ is contractive with respect to the max norm. Then, from Theorem 11.22, the ODE in Eq. (11.78) must have a unique globally asymptotically stable equilibrium.

Condition 5. This is the boundedness condition. We will now show boundedness using a proof based on basic principles from

Gosavi [111]. We will later present two other proofs that exploit Theorem 11.23. The boundedness argument is next provided as a separate lemma.

LEMMA 11.30 *In Q-Learning (for discounted reward MDPs), under synchronous and asynchronous updating, the iterate $Q^k(i,a)$ for any state-action pair (i,a) in its kth update remains bounded, as long as we start with some finite starting values for the Q-factors, where $\lambda \in [0,1)$.*

Proof We first claim that for every state-action pair (i,a):

$$|Q^k(i,a)| \leq M(1+\lambda+\lambda^2+\cdots+\lambda^k), \tag{11.79}$$

where λ is the discounting factor and M is a positive finite number defined as follows:

$$M = \max\left\{r_{\max}, \max_{i\in\mathcal{S},a\in\mathcal{A}(i)} Q^1(i,a)\right\}, \tag{11.80}$$

$$\text{where } r_{\max} = \max_{i,j\in\mathcal{S},a\in\mathcal{A}(i)} |r(i,a,j)|. \tag{11.81}$$

Since the immediate rewards are finite by definition, $r_{\max}$ must be a finite scalar. Since we start with finite values for the Q-factors, then M too has to be finite. Then, from the above claim (11.79), boundedness follows since if $k \to \infty$,

$$\limsup_{k\to\infty} |Q^k(i,a)| \leq M\frac{1}{1-\lambda}$$

for all $i \in \mathcal{S}$ and $a \in \mathcal{A}(i)$, since $0 \leq \lambda < 1$. (In the above, we have used the formula for a convergent infinite geometric series.) The right hand side of the above is a finite scalar, and so $Q^k(i,a)$ will always be finite, thus establishing the result. Thus, all we need to do is to prove our claim in (11.79). We will use an induction argument.

Note that in asynchronous updating, in one iteration of the algorithm, the Q-factor for only one state-action pair is updated, and the other Q-factors remain unchanged. Hence, in general, in the kth iteration of the asynchronous algorithm, the update for $Q^k(i,a)$ is either according to Case 1 or Case 2.

Case 1: The state-action pair is updated in the kth iteration:

$$Q^{k+1}(i,a) = (1-\alpha)Q^k(i,a) + \alpha\left[r(i,a,j) + \lambda \max_{b\in\mathcal{A}(j)} Q^k(j,b)\right].$$

Case 2: The state-action pair is not updated in the kth iteration:

$$Q^{k+1}(i,a) = Q^k(i,a).$$

Now, if the update is carried out as in Case 1:

$$\begin{aligned}
|Q^2(i,a)| &\leq (1-\alpha)|Q^1(i,a)| + \alpha|r(i,a,j) + \lambda \max_{b\in\mathcal{A}(j)} Q^1(j,b)| \\
&\leq (1-\alpha)M + \alpha M + \alpha\lambda M \text{ (from (11.81) and (11.80))} \\
&\leq (1-\alpha)M + \alpha M + \lambda M \text{ (from the fact that } \alpha \leq 1) \\
&= M(1+\lambda)
\end{aligned}$$

Now, if the update is carried out as in Case 2:

$$\begin{aligned}
|Q^2(i,a)| &= |Q^1(i,a)| \\
&\leq M \leq M(1+\lambda).
\end{aligned}$$

From the above, our claim in (11.79) is true for $k = 1$. Now assuming that the claim is true when $k = m$, we have that for all $(i,a) \in (\mathcal{S} \times \mathcal{A}(i))$.

$$|Q^m(i,a)| \leq M(1+\lambda+\lambda^2+\cdots+\lambda^m). \qquad (11.82)$$

Now, if the update is carried out as in Case 1:

$$\begin{aligned}
|Q^{m+1}(i,a)| &\leq (1-\alpha)|Q^m(i,a)| + \alpha|r(i,a,j) + \lambda \max_{j\in\mathcal{A}(j)} Q^m(j,b)| \\
&\leq (1-\alpha)M(1+\lambda+\lambda^2+\cdots+\lambda^m) \\
&\quad +\alpha M + \alpha\lambda M(1+\lambda+\lambda^2+\cdots+\lambda^m) \text{ (from (11.82))} \\
&= M(1+\lambda+\lambda^2+\cdots+\lambda^m) \\
&\quad -\alpha M(1+\lambda+\lambda^2+\cdots+\lambda^m) \\
&\quad +\alpha M + \alpha\lambda M(1+\lambda+\lambda^2+\cdots+\lambda^m) \\
&= M(1+\lambda+\lambda^2+\cdots+\lambda^m) \\
&\quad -\alpha M(1+\lambda+\lambda^2+\cdots+\lambda^m) \\
&\quad +\alpha M + \alpha M(\lambda+\lambda^2+\cdots+\lambda^{m+1}) \\
&= M(1+\lambda+\lambda^2+\cdots+\lambda^m) \\
&\quad -\alpha M(1+\lambda+\lambda^2+\cdots+\lambda^m) + \alpha M \\
&\quad +\alpha M(\lambda+\lambda^2+\cdots+\lambda^m) + \alpha M\lambda^{m+1} \\
&= M(1+\lambda+\lambda^2+\cdots+\lambda^m) - \alpha M(1+\lambda+\lambda^2+\cdots+\lambda^m) \\
&\quad +\alpha M(1+\lambda+\lambda^2+\cdots+\lambda^m) + \alpha M\lambda^{m+1} \\
&= M(1+\lambda+\lambda^2+\cdots+\lambda^m) + \alpha M\lambda^{m+1}
\end{aligned}$$

$$\begin{aligned} &\leq M(1+\lambda+\lambda^2+\cdots+\lambda^m)+M\lambda^{m+1} \text{ (since } 0<\alpha \leq 1) \\ &= M(1+\lambda+\lambda^2+\cdots+\lambda^m+\lambda^{m+1}) \end{aligned}$$

Now, if the update is carried out as in Case 2:

$$\begin{aligned} |Q^{m+1}(i,a)| &= |Q^m(i,a)| \\ &\leq M(1+\lambda+\lambda^2+\cdots+\lambda^m) \\ &\leq M(1+\lambda+\lambda^2+\cdots+\lambda^m+\lambda^{m+1}) \end{aligned}$$

From the above, the claim in (11.79) is proved for $k = m+1$. ▮

Conditions 6 and 7. These conditions are ensured with appropriate step sizes and by appropriate exploration, as discussed above.

Then, by Theorem 11.21, we have convergence with probability 1 to the unique globally asymptotically stable equilibrium of the ODE in Eq. (11.78); denote the equilibrium by $\vec{Q}^\infty$. From the definition of equilibrium (see Definition 9.10 from Chap. 9), it should be clear that the iterates converge with probability 1 to the solution of the following: For all (i,a) pairs,

$$F\left(\vec{Q}^\infty\right)(i,a) = 0, \text{ or } F'\left(\vec{Q}^\infty\right)(i,a) = Q^\infty(i,a).$$

But by Proposition 11.25, the above is the optimal solution (solution of the Q-factor version of the Bellman optimality equation), and we are done. ▮

8.1.1 Alternative Proofs for Boundedness

It is important to note that Lemma 11.30 holds for any sample path chosen in the simulator. We will now show boundedness of the Q-factors in Q-Learning via two other techniques, but these results will be true with probability 1. The first one will be based on the eigenvalues of an associated matrix, while the second is the most general of proofs, and all it needs is the contraction property underlying the transformation. Both results will rely on using Theorem 11.23.

Eigenvalue Proof. For this proof, we need a basic result from ODEs (see Theorem 4.1 on page 151 of [54]), which is as follows.

THEOREM 11.31 *Consider the ODE:*

$$\frac{d\vec{x}}{dt} = \mathbf{A}\vec{x}, \text{ where } \mathbf{A} \text{ is a real constant square matrix.}$$

The critical (equilibrium) point of the ODE must be asymptotically stable if all the eigenvalues of $\mathbf{A}$ *have strictly negative, real parts.*

In order to invoke this result, we need some work. We first compute the scaled function (see Definition 11.6) for the Q-Learning transformation as follows: For all (i,a) pairs and any $c > 0$,

$$F_c\left(\vec{Q}^k\right)(i,a) = \sum_{j=1}^{|\mathcal{S}|} p(i,a,j)\left[\frac{r(i,a,j)}{c} + \lambda \max_{b \in \mathcal{A}(j)} Q^k(j,b)\right] - Q^k(i,a). \tag{11.83}$$

The above implies that for all (i,a) pairs:

$$F_\infty\left(\vec{Q}^k\right)(i,a) \equiv \lim_{c\to\infty} F_c\left(\vec{Q}^k\right)(i,a) = \sum_{j=1}^{|\mathcal{S}|} p(i,a,j)\left[\lambda \max_{b\in\mathcal{A}(j)} Q^k(j,b)\right] - Q^k(i,a).$$

Then, from simple matrix algebra, it can be shown that we can write the matrix $F_\infty\left(\vec{Q}^k\right)$ as follows:

$$F_\infty\left(\vec{Q}^k\right) = \lambda \mathbf{L}\vec{Q}^k - \vec{Q}^k, \tag{11.84}$$

where $\mathbf{L}$ is a square matrix of size $N \times N$ of which each element is either a transition probability ($p(i,a,j)$ term) or 0. The above will be useful in our result below.

Lemma 11.32 *In Q-Learning (for discounted reward MDPs), under synchronous and asynchronous updating, the sequence* $\{\vec{Q}^k\}_{k=1}^{\infty}$ *remains bounded with probability 1.*

Proof We re-write Eq. (11.84) as: $F_\infty\left(\vec{Q}^k\right) = (\lambda\mathbf{L} - \mathbf{I})\,\vec{Q}^k$, where $\mathbf{I}$ is the identity matrix of size N. Since $\lambda < 1$, we have that $||\lambda\mathbf{L}||_\infty < 1$ (where $||.||_\infty$ is the max norm of a matrix; see Appendix for matrix norms, eigenvalues and spectral radii), and hence $\nu(\lambda\mathbf{L}) \le ||\lambda\mathbf{L}||_\infty < 1$, where $\nu(.)$ denotes the spectral radius of a matrix. If ψ_s denotes the sth eigenvalue of $\lambda\mathbf{L}$, then the definition of spectral radius implies: $0 < |\psi_s| < 1$ for all s. Now, the eigenvalue of $\lambda\mathbf{L} - \mathbf{I}$ must equal the eigenvalue of $\lambda\mathbf{L}$ minus 1 (see spectral shift property in Appendix), and hence every eigenvalue of $\lambda\mathbf{L}-\mathbf{I}$ must be strictly negative. Then, Theorem 11.31 implies that the ODE $\frac{d\vec{q}}{dt} = F_\infty(\vec{q})$ must have an asymptotically stable equilibrium. But, since the origin is the only equilibrium for this ODE, applying Theorem 11.23, we are done. ∎

Note that it may be tempting to use the result for the case $\lambda = 1$. But when $\lambda = 1$, the eigenvalues will not be strictly non-negative, and hence it is not possible to show boundedness via the approach above.

Contraction Property Proof. The proof we now present will invoke the contraction property of the transformation $F'(.)$. Again, it will establish boundedness with probability 1.

LEMMA 11.33 *In Q-Learning (for discounted reward MDPs), under synchronous and asynchronous updating, the sequence* $\{\vec{Q}^k\}_{k=1}^{\infty}$ *remains bounded with probability 1.*

Proof It has been shown in proving Condition 4 in the proof of Proposition 11.29 that $F'(.)$ is contractive with respect to the max norm (a special case of that same with respect to the weighted max norm). Then, Theorem 11.22 implies that the ODE in Eq. (11.78) must have a unique globally asymptotically stable equilibrium.

Now if we compute the scaled function $F_c(.)$, as defined in Definition 11.6, we can show (see Eq. (11.83) and its accompanying discussion) that:

$$\begin{aligned} F_{\infty}\left(\vec{Q}^k\right)(i,a) &= \lim_{c\to\infty} F_c\left(\vec{Q}^k\right)(i,a) \\ &= \lambda\sum_{j=1}^{|\mathcal{S}|} p(i,a,j)\left[\max_{b\in\mathcal{A}(j)} Q^k(j,b)\right] - Q^k(i,a); \end{aligned}$$

it is not hard to see that $F_{\infty}(.)$ is a special case of the transformation $F(.)$ with the immediate reward set to 0, i.e., $r(i,a,j)=0$ for all $i\in\mathcal{S}$, $j\in\mathcal{S}$, and $a\in\mathcal{A}(i)$. Thus, the ODE $\frac{d\vec{q}}{dt}=F_{\infty}(\vec{q})$ must have a globally asymptotically stable equilibrium. But note that the origin is the only equilibrium point for this ODE. Then, Theorem 11.23 implies that the iterates must be bounded with probability 1. ∎

8.1.2 Finite Convergence of Q-Learning

A feature of the convergence analysis above that we cannot overlook is the attainment of convergence after an *infinite* number of iterations. The question that naturally arises now is: is there is a finite number of iterations after which the algorithm will generate the optimal solution? The other question is how many samples should be collected (or iterations performed) for the estimates of the iterates to be ϵ-close to their limiting values.

We will now prove using analysis from [109] that it is possible to terminate Q-Learning in a finite number of iterations to obtain the optimal solution. Our analysis, however, will not place a bound on the number of samples needed for convergence; for that the interested reader is referred to [47, 83]. The analysis follows from the fact that Q-factors converge and should hold whenever convergence can be shown in an algorithm with probability 1.

We first provide some notation needed for our analysis. For each state $i \in \mathcal{S}$, let $Q_1^*(i)$ and $Q_2^*(i)$ denote the highest value (maximum value) and the second-highest value (value lower than the maximum but higher than all of the other values) of the Q-factors, respectively, which are obtained from running the Q-Learning algorithm for an infinite number of iterations; i.e., these are the optimal values of the Q-factors obtained theoretically in the limit. That these limiting (optimal) values exist with probability 1 is a consequence of Proposition 11.29. (Note that the case where all actions are equally good is trivial and is not considered.)

Further, for each $i \in \mathcal{S}$, let

$$\mathcal{M}_1(i) = \arg\max_{c \in \mathcal{A}(i)} Q^*(i,c),$$

where $Q^*(i,c)$ denotes the limiting Q-factor for the state-action pair (i,c). Thus, the set $\mathcal{M}_1(i)$ contains all the actions that maximize the optimal Q-factor in state i. Similarly, for each $i \in \mathcal{S}$, let

$$\mathcal{M}_2(i) = \text{arg2max}_{c \in \mathcal{A}(i)} Q^*(i,c),$$

where the set $\mathcal{M}_2(i)$ contains all the actions that produce the second-highest value for the limiting Q-factor in state i. These sets will be non-empty (provided we have at least two actions, which we assume to be true) and can be possibly singletons. (For the sake of simplicity, the reader may assume these sets to be singletons and generalize later).

Now, for any state $i \in \mathcal{S}$ and for any $a_1 \in \mathcal{M}_1(i)$, let $\tilde{Q}_1^k(i,a_1)$ denote the value of the Q-factor of state i and action a_1 in the kth iteration. Similarly, for each state $i \in \mathcal{S}$ and for $a_2 \in \mathcal{M}_2(i)$, let $\tilde{Q}_2^k(i,a_2)$ denote the value of the Q-factor of state i and action a_2 in the kth iteration. Note that here a_1 and a_2 assume values from the sets defined above. Let $\mathcal{J}$ be the set of positive integers and let $k \in \mathcal{J}$.

PROPOSITION 11.34 *With probability 1, there exists a positive integer K such that for $k \geq K$, for each $i \in \mathcal{S}$, and for every (a_1, a_2)-pair,*

$$\tilde{Q}_1^k(i,a_1) > \tilde{Q}_2^k(i,a_2), \textit{ where } a_1 \in \mathcal{M}_1(i) \textit{ and } a_2 \in \mathcal{M}_2(i). \quad (11.85)$$

The above implies that the algorithm can be terminated in a finite number of iterations with probability 1. This is because when for every state, the estimate of what is the highest Q-factor starts exceeding that of what is the second-highest Q-factor, the algorithm should generate the optimal policy.

Proof We will first assume that we are working with one specific (a_1, a_2)-pair. This allows us to conceal a_1 and a_2 from the notation thereby easing the latter. Thus, $\tilde{Q}_1^k(i, a_1)$ will be replaced by $\tilde{Q}_1^k(i)$ and $\tilde{Q}_2^k(i, a_2)$ will be replaced by $\tilde{Q}_2^k(i)$. The result can be shown to hold for every such pair.

We first define the absolute value of the difference between the limiting value and the estimate in the kth iteration. To this end, for each $i \in \mathcal{S}$, let

$$e_1^k(i) = |\tilde{Q}_1^k(i) - Q_1^*(i)|, \text{ and } e_2^k(i) = |\tilde{Q}_2^k(i) - Q_2^*(i)|.$$

From the above, it is easy to see that we can have four different cases for the values of the estimates, depending on the value of the difference:

$$\text{Case 1: } \tilde{Q}_1^k(i) = Q_1^*(i) - e_1^k(i), \text{ and } \tilde{Q}_2^k = Q_2^*(i) + e_2^k(i);$$

$$\text{Case 2: } \tilde{Q}_1^k(i) = Q_1^*(i) - e_1^k(i), \text{ and } \tilde{Q}_2^k = Q_2^*(i) - e_2^k(i);$$
$$\text{Case 3: } \tilde{Q}_1^k(i) = Q_1^*(i) + e_1^k(i), \text{ and } \tilde{Q}_2^k = Q_2^*(i) + e_2^k(i);$$
$$\text{Case 4: } \tilde{Q}_1^k(i) = Q_1^*(i) + e_1^k(i), \text{ and } \tilde{Q}_2^k = Q_2^*(i) - e_2^k(i).$$
$$\text{Let } D(i) \equiv Q_1^*(i) - Q_2^*(i) \text{ for all } i \in S.$$

$D(i) > 0$ for all $i \in \mathcal{S}$ because of the following. By its definition, $D(i) >= 0$ for any i, but $D(i) = 0$ for the situation in which all actions are equally good for the state in question, in which case there is nothing to be proved.

We will now *assume* that there exists a value K for k such that for each i, both $e_1^k(i)$ and $e_2^k(i)$ are less than $D(i)/2$ when $k \geq K$. We will prove later that this assumption holds in Q-Learning. Thus, our immediate goal is to show inequality (11.85) for each case, under this assumption.

We first consider Case 1.

$$\tilde{Q}_1^k(i) - \tilde{Q}_2^k(i)$$

$$= Q_1^*(i) - Q_2^*(i) - e_2^k(i) - e_1^k(i) \text{ from Case 1.}$$
$$= D(i) - (e_1^k(i) + e_2^k(i)) > D(i) - D(i) = 0.$$

Then, $\tilde{Q}_1^k(i) > \tilde{Q}_2^k(i) \quad \forall i, k \geq K$, proving inequality (11.85). The rest of the cases should be obvious from drawing a simple figure, but we present details. Like in Case 1, for Case 4, we can show that $\tilde{Q}_1^k(i) - \tilde{Q}_2^k(i) = D(i) + (e_1^k(i) + e_2^k(i)) > 0$, since $e_l^k(i) \geq 0$ for $l = 1, 2$. For Case 2, since $e_2^k(i) \geq 0$, we have that

$$\frac{D(i)}{2} + e_2^k(i) \geq \frac{D(i)}{2}.$$

Also, since $e_1^k(i) < \frac{D(i)}{2}$, we have that

$$\frac{D(i)}{2} - e_1^k(i) > 0.$$

Combining the two inequalities above, we have that $D(i) - (e_1^k(i) - e_2^k(i)) > \frac{D(i)}{2}$. Then, we have that $\tilde{Q}_1^k(i) - \tilde{Q}_2^k(i) = D(i) - (e_1^k(i) - e_2^k(i)) > \frac{D(i)}{2} > 0$. Case 3 can be proved in a manner very similar to that of Case 2.

What remains to be shown is the assumption we made above. Now, Proposition 11.29 implies that for every $i \in \mathcal{S}$, with probability 1,

$$\lim_{k\to\infty} \tilde{Q}_1^k(i) = Q_1^*(i) \text{ and } \lim_{k\to\infty} \tilde{Q}_2^k(i) = Q_2^*(i).$$

Hence, for any given $\epsilon > 0$, there exists a value $k_1 \in \mathcal{J}$ for which $|\tilde{Q}_1^k(i) - Q_1^*(i)| < \epsilon$ when $k \geq k_1$. Similarly, for any given $\epsilon > 0$, there exists a value $k_2 \in \mathcal{J}$ for which $|\tilde{Q}_2^k(i) - Q_2^*(i)| < \epsilon$ when $k \geq k_2$. Selecting $\epsilon = D(i)/2$ (that $D(i) > 0$ has been shown above) and $K \equiv \max\{k_1, k_2\}$, we have that for $k \geq K$, with probability 1,

$$|\tilde{Q}_1^k(i) - Q_1^*(i)| < \epsilon = D(i)/2, \text{ i.e., } e_1^k(i) < D(i)/2.$$

Similarly, using the same value for ϵ, one can show that for $k \geq K$, with probability 1, $e_2^k(i) < D(i)/2$, which proves our assumption. The result above did not depend on any specific value of a_1 or a_2, and can be similarly shown for every (a_1, a_2)-pair. ∎

An issue related to the above is: how should the algorithm be terminated? The reader may recall that like Q-Learning, convergence occurs in the limit for value iteration, i.e., when the number of iterations tends to infinity. In value iteration, however, we can use the norm of a difference vector to terminate the algorithm. Unfortunately in RL, this is not true for the following reason: Only one Q-factor gets updated in each iteration, leading to a situation where the number of times a state-action pair has been updated thus far (updating frequency) is unlikely to be the same for all state-action pairs at any given (algorithm) iteration. Hence, computing the norm or span of the difference vector is essentially not useful.

In practice, for termination, we run the algorithm for as long as we can, i.e., for a pre-specified, fixed number of iterations. It makes sense to use an appropriate step size such that the step size remains reasonably large until the pre-specified number of iterations are complete.

When the step size becomes too small, e.g., 10^{-6}, due to computer-roundoff errors, the Q-factors cease to change, and there is no point in continuing further. Another way is to terminate the algorithm if the policy has not changed in the last several iterations. But this requires checking the policy after every iteration (or at least after a few iterations), which may be computationally burdensome (and impossible for huge state-action spaces).

8.2. Relative Q-Learning: Average Reward MDPs

We will now prove that Relative Q-Learning for average reward MDPs converges under asynchronous conditions. The core of the Relative Q-Learning algorithm can be expressed by the following transformation:

$$Q^{k+1}(i,a) \leftarrow Q^k(i,a) + \alpha\left[r(i,a,\xi^k) + \max_{b\in\mathcal{A}(\xi^k)} Q^k(\xi^k,b) - Q^k(i,a)\right] - Q^k(i^*,a^*), \tag{11.86}$$

where ξ^k is a random variable that depends on (i,a) and k, and $Q^k(i^*,a^*)$ denotes the Q-factor of the distinguished state-action pair in the kth iteration. Let the policy generated in the kth iteration be defined by:

$$\mu^k(i) \in \arg\max_{a\in\mathcal{A}(i)} Q^k(i,a) \text{ for all } i \in \mathcal{S}.$$

Let the optimal policy be denoted by $\hat{\mu}^*$. The convergence result for algorithm is as follows.

Proposition 11.35 *When the step sizes and action selection used in the algorithm satisfy Conditions 3, 6, and 7 of Theorem 11.21, with probability 1, the sequence of policies generated by the Relative Q-Learning algorithm, $\{\hat{\mu}^k\}_{k=1}^{\infty}$, converges to $\hat{\mu}^*$.*

The proof will be along the lines of that of Q-Learning; however, showing the existence of the asymptotically stable critical point (Condition 4) is not as straightforward here, since the associated transformation $F'(.)$ is not contractive and hence we may not use Theorem 11.22. To show Condition 4, we will need a critical result from [2] (the proof of which is beyond our scope here). The boundedness condition (Condition 5) will be based on Theorem 11.23.

Proof Like in the proof of Q-Learning, we will first define some of the underlying transformations and then invoke an ODE. Hence, we define the transformations $F'(.)$ on the vector $\vec{Q}^k$ as follows: For all (i,a) pairs,

$$F'\left(\vec{Q}^k\right)(i,a) = \sum_{j=1}^{|\mathcal{S}|} p(i,a,j)\left[r(i,a,j) + \max_{b\in\mathcal{A}(j)} Q^k(j,b)\right] - Q^k(i^*,a^*).$$

Then, we can define $F(.)$ via Eq. (11.76). We further define a transformation $f'(.)$ as follows:

$$f'\left(\vec{Q}^k\right)(i,a) = \left[r(i,a,\xi^k) + \max_{b\in\mathcal{A}(\xi^k)} Q^k(\xi^k,b)\right] - Q^k(i^*,a^*),$$

which allows us to define the noise term $w^k(i,a)$ as in Eq. (11.77). Then, like in the case of Q-Learning, we can write the updating transformation in our algorithm, i.e., Eq. (11.86) as:

$$Q^{k+1}(i,a) = Q^k(i,a) + \alpha^k\left[F\left(\vec{Q}^k\right)(i,a) + w^k(i,a)\right],$$

which is of the same form as the updating scheme defined for Theorem 11.21 (replace X^k by Q^k and l by (i,a)). Then, we can invoke the following ODE as in Theorem 11.21:

$$\frac{d\vec{q}}{dt} = F(\vec{q}), \tag{11.87}$$

where $\vec{q}$ denotes the continuous-valued variable underlying the iterate Q. We now need to evaluate the conditions of Theorem 11.21.

Conditions 1, 2, 3, 6, and 7 follow in a manner identical (or very similar) to that shown for Q-Learning (Theorem 11.29). However, Conditions 4 and 5 need additional work.

Conditions 4 and 5. To show these conditions, we first define $\vec{Q}_*$ as the solution for Eq. (11.74) generated when ρ in Eq. (11.74) is replaced by $Q(i^*,a^*)$, where (i^*,a^*) is some distinguished state-action pair. That this solution is unique can be shown (see Lemma 3.2 in [2]).

We will now invoke a very useful result (presented as Theorem 3.4 in [2]). We skip the proof because of its involved nature.

LEMMA 11.36 *$\vec{Q}_*$ is the globally asymptotically stable equilibrium for the ODE in Eq. (11.87).*

The above implies Condition 4.

To show Condition 5, note that the above theorem also holds for the special case when all the immediate rewards are set to 0, i.e., $r(i,a,j)=0$ for all $i \in \mathcal{S}$, $j \in \mathcal{S}$, and $a \in \mathcal{A}(i)$. Now if we compute the scaled function $F_c(.)$, as defined in Definition 11.6, we can show (see Eq. (11.83) and its accompanying discussion) that:

$$\begin{aligned} F_\infty\left(\vec{Q}^k\right)(i,a) &= \lim_{c\to\infty} F_c\left(\vec{Q}^k\right)(i,a) \\ &= \sum_{j=1}^{|\mathcal{S}|} p(i,a,j)\left[\max_{b\in\mathcal{A}(j)} Q^k(j,b)\right] - Q^k(i^*,a^*) - Q^k(i,a); \end{aligned}$$

it is not hard to see that $F_\infty(.)$ is a special case of the transformation $F(.)$ with the immediate reward set to 0. Hence, Lemma 11.36 will imply that the ODE $\frac{d\vec{q}}{dt} = F_\infty(\vec{q})$ has a globally asymptotically stable equilibrium. But note that the origin is the only equilibrium point for this ODE. Then, from Theorem 11.23, it follows that the sequence $\left\{\vec{Q}^k\right\}_{k=1}^{\infty}$ must be bounded with probability 1 (i.e., Condition 5).

Then, by Theorem 11.21, we have convergence of $\left\{\vec{Q}^k\right\}_{k=1}^{\infty}$ with probability 1 to the unique globally asymptotically stable equilibrium of the ODE in Eq. (11.87), i.e., $\vec{Q}_*$. But by Proposition 11.27, this must be the optimal solution (solution of the Q-factor version of the Bellman optimality equation), and hence the sequence of policies must converge to the optimal policy with probability 1. ∎

8.3. CAP-I: Discounted Reward MDPs

We remind the reader that conservative approximate policy iteration (CAP-I) is based on classical policy iteration. It has been already argued via Eq. (7.23) in Chap. 7 that the policy improvement step in an RL algorithm that uses the Q-factors of a policy to generate a new policy is equivalent to that in the classical policy iteration algorithm (based on the value function and transition probabilities). Hence, our analysis here will be restricted to the first two stages in the algorithm and to showing that the algorithm generates the value function vector and the Q-factors associated to the policy being evaluated, i.e., the solution of Bellman policy equation (Poisson equation).

Recall that the first stage of CAP-I is geared towards generating the solution of the Bellman policy equation (value function version) for a given policy. The core of CAP-I in Step 2b can be expressed by the following transformation:

$$J^{n+1}(i) \leftarrow J^n(i) + \alpha\left[r(i,\mu(i),\xi^n) + \lambda J^n(\xi^n) - J^n(i)\right], \qquad (11.88)$$

where ξ^n is a random variable that depends on (i, a) and n, the iteration number within Step 2, and μ denotes the policy being evaluated. The next result shows that the sequence of iterates defined above converges to the solution of Equation (11.22), the Bellman policy equation.

Proposition 11.37 *When the step sizes and the action selection satisfy Conditions 3, 6, and 7 of Theorem 11.21, with probability 1, the sequence of iterates generated within Step 2 of the CAP-I algorithm, $\{\vec{J}^n\}_{n=1}^{\infty}$, converges to the unique solution of the Bellman policy equation, i.e., to the value function vector associated to policy $\hat{\mu}$.*

Proof The main theme underlying this proof is very similar to that of Q-Learning. As usual, we first define some transformations: For all $i \in \mathcal{S}$,

$$F'\left(\vec{J}^n\right)(i) = \sum_{j=1}^{|\mathcal{S}|} p(i, \mu(i), j)\left[r(i, \mu(i), j) + \lambda J^n(j)\right];$$

$$F\left(\vec{J}^n\right)(i) = \sum_{j=1}^{|\mathcal{S}|} p(i, \mu(i), j)\left[r(i, \mu(i), j) + \lambda J^n(j)\right] - J^n(i);$$

$$f'\left(\vec{J}^n\right)(i) = \left[r(i, \mu(i), \xi^n) + \lambda J^n(\xi^n)\right];$$

which allow us to define the noise term $w^n(i, \mu(i))$ as:

$$w^n(i, \mu(i)) = f'\left(\vec{J}^n\right)(i) - F'\left(\vec{J}^n\right)(i).$$

Then, we can write the updating transformation in our algorithm, i.e., Eq. (11.88) as:

$$J^{n+1}(i) = J^n(i) + \alpha^k\left[F\left(\vec{J}^n\right)(i) + w^n(i, \mu(i))\right],$$

which is of the same form as the updating scheme defined for Theorem 11.21 (replace X^n by J^n and l by $(i, \mu(i))$). Then, we can invoke the following ODE as in Theorem 11.21:

$$\frac{d\vec{\mathrm{j}}}{dt} = F\left(\vec{\mathrm{j}}\right), \tag{11.89}$$

where $\vec{\mathrm{j}}$ denotes the continuous-valued variable underlying the iterate $\vec{J}$. We now need to evaluate the conditions of Theorem 11.21.

Conditions 1, 2, 3, 6, and 7 follow in a manner identical (or very similar) to that shown for Q-Learning (Theorem 11.29). However, Conditions 4 and 5 need additional work.

Conditions 4 and 5. Note that transformation $F'(.)$ is contractive with respect to the max norm (see Proposition 11.4). Hence, Theorem 11.22 applies, implying that ODE in (11.89) has a unique globally asymptotically stable equilibrium (i.e., Condition 4 holds).

To show Condition 5, note that the above finding also holds for the special case when all the immediate rewards are set to 0, i.e., $r(.,.,.) = 0$. Now if we compute the scaled function $F_c(.)$, as defined in Definition 11.6, we can show (see Eq. (11.83) and its accompanying discussion) that:

$$\begin{aligned} F_\infty\left(\vec{J}^n\right)(i) &= \lim_{c\to\infty} F_c\left(\vec{J}^n\right)(i) \\ &= \lambda \sum_{j=1}^{|\mathcal{S}|} p(i,\mu(i),j)\left[J^n(j)\right]; \end{aligned}$$

it is not hard to see that $F_\infty(.)$ is a special case of the transformation $F(.)$ with the immediate reward set to 0, and hence the ODE $\frac{d\vec{\mathsf{j}}}{dt} = F_\infty\left(\vec{\mathsf{j}}\right)$ must also have a unique globally asymptotically stable equilibrium. But note that the origin is the only equilibrium point for this ODE. Then, from Theorem 11.23, it follows that the sequence $\left\{\vec{J}^n\right\}_{n=1}^{\infty}$ must be bounded with probability 1 (i.e., Condition 5 holds).

Then, by Theorem 11.21, we have convergence of $\left\{\vec{J}^n\right\}_{n=1}^{\infty}$ with probability 1 to the unique globally asymptotically stable equilibrium of the ODE in Eq. (11.89), which, by the definition of the equilibrium point, *must* be the solution of the Bellman policy equation, i.e., Eq. (11.22). Then, by Proposition 11.21, the solution must be the value function vector associated to the policy $\hat{\mu}$. ∎

We now analyze the convergence of the Q-factors in Step 3 of CAP-I. The core of Step 3b in CAP-I can be expressed by the following transformation:

$$Q^{m+1}(i,a) \leftarrow Q^m(i,a) - \alpha^m\left[Q^m(i,a) - r(i,\mu(i),\xi^m) - \lambda J(\xi^m)\right], \tag{11.90}$$

where ξ^m is a random variable that depends on (i,a) and m, the iteration number within Step 3, and the vector $\vec{J}$ is a constant. The goal here is for the Q-factor to estimate the following mean:

$$\sum_{j\in\mathcal{S}} p(i,a,j)\left[r(i,a,j) + \lambda J(j)\right] \equiv \mathsf{E}[Z(i,a)], \tag{11.91}$$

where $Z(i,a)$ is a random variable and $\vec{J}$ denotes the vector obtained in Step 2 of the algorithm. Note that since $\vec{J}$ is fixed in Step 3, the terms within the squared brackets in the LHS of the above are constants, and hence the update in (11.90) becomes a straightforward Robbins-Monro update in which the mean of a random variable is estimated from its values. Our main result is as follows:

Proposition 11.38 *When the step sizes and the action selection satisfy Condition 3 of Theorem 11.21 and $Q^m(i,a)$ denotes the iterate for the (i,a)-th pair in the mth iteration of Step 3 in the CAP-I algorithm, with probability 1, for every state-action pair (i,a), $\lim_{m\to\infty} Q^m(i,a) = \mathsf{E}[Z(i,a)]$, where $[Z(i,a)]$ is defined in (11.91).*

The result implies that the limit point $\vec{Q}^{\infty}$ will equal $\mathsf{E}[Z(i,a)]$, i.e., for all (i,a) pairs:

$$Q^{\infty}(i,a) = \sum_{j\in\mathcal{S}} p(i,a,j)\left[r(i,a,j) + \lambda J(j)\right],$$

which satisfies the definition of Q-factors in Eq. (11.71) (note: that $\vec{J}$ is the unique solution of the Bellman policy equation has already been established in Proposition 11.37), and hence from the discussion accompanying Eq. (11.71), we can claim that the above is the unique solution of the Q-factor version of the Bellman equation for a given policy, i.e., Eq. (11.72).

Proof For this proof, we first note that since (i) the update of a given Q-factor is unrelated to that of any other Q-factor (this is because the updating equation does not contain any other Q-factor) and (ii) each state-action pair is tried with the same frequency (this is because each action is tried with the same probability in every state), we can analyze the convergence of each Q-factor independently (separately).

For the proof, we will rely on the standard arguments used in the literature to show convergence of a Robbins-Monro algorithm (see e.g., [33]). The latter can be shown to be a special case of the stochastic gradient algorithm discussed in Chap. 10. As such, the result associated to stochastic gradients (Theorem 10.8) can be exploited for the analysis after finding a so-called "potential function" needed in the stochastic gradient algorithm.

We define the potential function $g : \Re^N \to \Re$ such that for any (i,a) pair:

$$g\left(\vec{Q}^m\right)(i,a) = \left(Q^m(i,a) - \mathsf{E}[Z(i,a)]\right)^2/2.$$

Then, for any (i,a)-pair, $$\frac{d\left(g\left(\vec{Q}^m\right)(i,a)\right)}{dQ^m(i,a)} = \left[Q^m(i,a) - \mathsf{E}[Z(i,a)]\right]. \tag{11.92}$$

We further define the noise term as follows for every (i,a)-pair:

$$\begin{aligned} w^m(i,a) &= \sum_{j\in\mathcal{S}} p(i,a,j)\left[r(i,a,j) + \lambda J(j)\right] - \left[r(i,\mu(i),\xi^m) + \lambda J(\xi^m)\right] \\ &= \mathsf{E}[Z(i,a)] - \left[r(i,\mu(i),\xi^m) + \lambda J(\xi^m)\right] \end{aligned} \tag{11.93}$$

Combining (11.92) and (11.93), we can write the update in (11.90) as:

$$Q^{m+1}(i,a) \leftarrow Q^m(i,a) - \alpha^m\left[\frac{d\left(g\left(\vec{Q}^m\right)(i,a)\right)}{dQ^m(i,a)} + w^m(i,a)\right],$$

which is of the form of the stochastic gradient algorithm considered in Eq. (10.16). In order to invoke the associated result (Theorem 10.8), we must show that all the conditions invoked in the theorem (Assumptions 10.5–10.7) are satisfied.

From the definition of $g(.)$, it is easy to see that it is non-negative everywhere. Also, from the first derivative calculated above, it can be shown to be continuously differentiable; further, the second derivative is bounded:

$$\frac{d^2\left(g\left(\vec{Q}^m\right)(i,a)\right)}{d\left(Q^m(i,a)\right)^2} = 1;$$

hence $\nabla g(.)$ must be Lipschitz, establishing Assumption 10.5.

The condition in Assumption 10.6 follows from Condition 1 of Theorem 11.21. Note that the condition in (10.18) within Assumption 10.7 follows from the fact that the noise has a conditional zero-mean (in a manner similar to arguments provided above in other algorithms); further note that the second moment of the noise for (i,a), i.e.,

$$\mathsf{E}\left[\left[w^m(i,a)\right]^2\middle|\,\mathcal{F}^m\right],$$

is finite from its definition in (11.93) (noting that $J(.)$ is a constant). Now, if the iterates $(Q^m(.,.))$ can be shown to be finite, the norm of the derivative defined in (11.92) must also be finite. Then, it should be possible to bound the norm of the second moment (a finite scalar) by a function of the norm of the derivative (another finite scalar), i.e., condition in (10.19) will hold. Hence, we now show that the iterates remain bounded.

LEMMA 11.39 *The sequence* $\left\{\vec{Q}^m\right\}_{m=1}^{\infty}$ *remains bounded.*

Proof The proof is very similar to that of Lemma 11.30; hence we present only the main ideas. The claim is that for every state-action pair (i, a):

$$|Q^m(i,a)| \leq M,$$

where M is a positive scalar defined as follows:

$$M = \max\left\{\max_{i,j\in\mathcal{S},a\in\mathcal{A}(i)} |r(i,a,j) + \lambda J(j)|, \max_{i\in\mathcal{S},a\in\mathcal{A}(i)} Q^0(i,a)\right\},$$

Since $J(.)$ is bounded and since clearly since we start with finite values for the Q-factors, M has to be finite. We will use an induction argument. We show the case for $m = 1$ as follows:

$$|Q^1(i,a)| \leq (1-\alpha)|Q^0(i,a)|+\alpha|r(i,a,j)+\lambda J(j)| \leq (1-\alpha)M+\alpha M = M$$

Now assuming that the claim is true when $m = P$, we have that for all (i, a): $|Q^P(i,a)| \leq M$. Then,

$$|Q^{P+1}(i,a)| \leq (1-\alpha)|Q^P(i,a)|+\alpha|r(i,a,j)+\lambda J(j)| \leq (1-\alpha)M+\alpha M=M.$$

The asynchronous update can be handled as in Lemma 11.30. ∎

Having shown all the conditions in Theorem 10.8, we can invoke **R2** in the theorem, which implies that with probability 1, the sequence of iterates will converge to the zero of the gradient, i.e., from (11.92), $\lim_{m\to\infty} Q^m(i,a) = \mathsf{E}[Z(i,a)]$ for any (i, a) pair. ∎

8.4. *Q*-*P*-Learning: Discounted Reward MDPs

The Q-P-Learning algorithm can be viewed as a Q-factor version of conservative approximate policy iteration (CAP-I). However, its convergence analysis is different because here Q-factors are evaluated on their own, unlike in CAP-I where the value function is first estimated and the Q-factors from the value function. However, the central idea is similar. In one policy evaluation, one seeks to estimate the Q-factors for a given policy, which are stored in the P-factors. As argued in the context of Eq. (7.23) in Chap. 7, if policy evaluation converges to the solution of the Bellman policy equation, the policy improvement step performed in Step 3 will behave as it should. Hence, we restrict our attention to showing that the Q-factors indeed converge to the solution of the Q-factor version of the Bellman policy equation (Poisson equation). Our main result is as follows.

PROPOSITION 11.40 *When step sizes and the action selection satisfy Conditions 3, 6, and 7 of Theorem 11.21, with probability 1, the sequence of iterates generated within Step 2 of the Q-P-Learning algorithm for discounted reward MDPs, $\{\vec{Q}^n\}_{n=1}^{\infty}$, converges to the unique solution of the Q-factor version of the Bellman policy equation, i.e., to the value function vector associated to policy $\hat{\mu}$.*

Proof The proof will be very similar to that of Q-Learning, since the policy evaluation phase in Q-P-Learning can essentially be viewed as Q-Learning performed for a fixed policy. We will use $\hat{\mu}$ to denote the policy being evaluated. Hence, using the notation of the algorithm, for all $i \in \mathcal{S}$:

$$\mu(i) \in \arg\max_{c \in \mathcal{A}(i)} P(i,c).$$

We will use n to denote the iteration within the policy evaluation phase. As usual, we begin with defining some functions: For all (i,a) pairs,

$$F'\left(\vec{Q}^n\right)(i,a) = \sum_{j=1}^{|\mathcal{S}|} p(i,a,j)\left[r(i,a,j) + \lambda Q^n(j,\mu(j))\right].$$

Then, we can define $F(.)$ via Eq. (11.76). We further define a transformation $f'(.)$ as follows:

$$f'\left(\vec{Q}^n\right)(i,a) = \left[r(i,a,\xi^n) + \lambda Q^k(\xi^n, \mu(\xi^n))\right],$$

which allows us to define the noise term $w^n(i,a)$ as in Eq. (11.77). Then, like in the case of Q-Learning, we can write the updating transformation in our algorithm, i.e., Eq. (7.21), as:

$$Q^{n+1}(i,a) = Q^n(i,a) + \alpha^n\left[F\left(\vec{Q}^n\right)(i,a) + w^n(i,a)\right],$$

which is of the same form as the updating scheme defined for Theorem 11.21 (replace X^n by Q^n and l by (i,a)). Then, we can invoke the following ODE as in Theorem 11.21:

$$\frac{d\vec{q}}{dt} = F(\vec{q}), \tag{11.94}$$

where $\vec{q}$ denotes the continuous-valued variable underlying the iterate Q. We now need to evaluate the conditions of Theorem 11.21.

Conditions 1, 2, 3, 6, and 7 follow in a manner identical (or very similar) to that shown for Q-Learning (Theorem 11.29). As argued in the case of Q-Learning, Conditions 4 and 5 will stem from showing that $F'(.)$ is contractive, which we next show. Consider two vectors $\vec{Q}_1^n$ and $\vec{Q}_2^n$ in $\Re^N$. From the definition of $F'(.)$ above, it follows that:

$$F'\left(\vec{Q}_1^n\right)(i,a) - F'\left(\vec{Q}_2^n\right)(i,a) = \lambda \sum_{j=1}^{|\mathcal{S}|} p(i,a,j)\left[Q_1^n(j,\mu(j)) - Q_2^n(j,\mu(j))\right].$$

For any (i,a) pair,

$$\begin{aligned}
\left|F'\left(\vec{Q}_1^n\right)(i,a) - F'\left(\vec{Q}_2^n\right)(i,a)\right| &\leq \lambda \sum_{j=1}^{|\mathcal{S}|} p(i,a,j)\left|Q_1^n(j,\mu(j)) - Q_2^n(j,\mu(j))\right| \\
&\quad \text{(from triangle inequality; page 285)} \\
&\leq \lambda \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) \max_{j\in\mathcal{S}, b\in\mathcal{A}(j)} \left|Q_1^n(j,b) - Q_2^n(j,b)\right| \\
&= \lambda \sum_{j=1}^{|\mathcal{S}|} p(i,a,j)||\vec{Q}_1^n - \vec{Q}_2^n||_\infty \\
&= \lambda ||\vec{Q}_1^n - \vec{Q}_2^n||_\infty \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) \\
&= \lambda ||\vec{Q}_1^n - \vec{Q}_2^n||_\infty \cdot 1.
\end{aligned}$$

Since the above holds for all values of (i,a), it also holds for the values that maximize the left hand side of the above. Therefore

$$||F'\vec{Q}_1^n - F'\vec{Q}_2^n||_\infty \leq \lambda ||\vec{Q}_1^n - \vec{Q}_2^n||_\infty. \quad \blacksquare$$

9. Reinforcement Learning for SMDPs: Convergence

In this section, we collect together results applicable to convergence of a subset of RL algorithms for solving SMDPs. As stated previously, many real-world problems are SMDPs rather than MDPs, and in many cases the results from MDPs do not extend naturally to SMDPs.

9.1. *Q*-Learning: Discounted Reward SMDPs

Discounted reward SMDPs form a special case where the results from the MDP extend naturally to the SMDP. Hence, we only present the main ideas.

The core of Q-Learning algorithm in this context for the generalized SMDP model was presented in Eq. (7.26). We define $F'\left(\vec{Q}^k\right)(i,a) =$

$$\sum_{j\in\mathcal{S}} p(i,a,j)\left[r_L(i,a,j) + \frac{1-e^{-\gamma\bar{t}(i,a,j)}}{\gamma} r_C(i,a,j) + e^{-\gamma\bar{t}(i,a,j)} \max_{b\in\mathcal{A}(j)} Q^k(j,b)\right].$$

The following result holds the key to showing Conditions 4 and 5 of Theorem 11.21; the remaining analysis for this algorithm follows in a manner identical to that of Q-Learning for discounted reward MDPs and is hence skipped.

LEMMA 11.41 *The transformation $F'(.)$ defined above is contractive with respect to the max norm.*

Proof First note that since the time term is always strictly non-negative ($\bar{t}(.,.,.) > 0$) and since $\gamma > 0$, there exists a scalar $\bar{\lambda}$ in the interval $(0, 1)$ such that

$$\max_{i,j \in \mathcal{S}; a \in \mathcal{A}(j)} \left| e^{-\gamma \bar{t}(i,a,j)} \right| \leq \bar{\lambda}.$$

Consider two vectors $\vec{Q}_1^k$ and $\vec{Q}_2^k$ in $\Re^N$. From the definition of $F'(.)$ above, it follows that: $F'\left(\vec{Q}_1^k\right)(i,a) - F'\left(\vec{Q}_2^k\right)(i,a) =$

$$\sum_{j=1}^{|\mathcal{S}|} e^{-\gamma \bar{t}(i,a,j)} p(i,a,j) \left[\max_{b \in \mathcal{A}(j)} Q_1^k(j,b) - \max_{b \in \mathcal{A}(j)} Q_2^k(j,b) \right].$$

From this, we can write that for any (i,a) pair: $\left| F'\left(\vec{Q}_1^k\right)(i,a) - F'\left(\vec{Q}_2^k\right)(i,a) \right|$

$$\leq \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) e^{-\gamma \bar{t}(i,a,j)} \left| \max_{b \in \mathcal{A}(j)} Q_1^k(j,b) - \max_{b \in \mathcal{A}(j)} Q_2^k(j,b) \right|$$

$$\leq \max_{i,j \in \mathcal{S}; a \in \mathcal{A}(j)} \left| e^{-\gamma \bar{t}(i,a,j)} \right| \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) \left| \max_{b \in \mathcal{A}(j)} Q_1^k(j,b) - \max_{b \in \mathcal{A}(j)} Q_2^k(j,b) \right|$$

$$\leq \bar{\lambda} \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) \max_{j \in \mathcal{S}, b \in \mathcal{A}(j)} |Q_1^k(j,b) - Q_2^k(j,b)|$$

$$= \bar{\lambda} \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) ||\vec{Q}_1^k - \vec{Q}_2^k||_\infty = \bar{\lambda} ||\vec{Q}_1^k - \vec{Q}_2^k||_\infty \sum_{j=1}^{|\mathcal{S}|} p(i,a,j) = \bar{\lambda} ||\vec{Q}_1^k - \vec{Q}_2^k||_\infty \cdot 1.$$

Then using standard arguments, $||F'\vec{Q}_1^k - F'\vec{Q}_2^k||_\infty \leq \bar{\lambda} ||\vec{Q}_1^k - \vec{Q}_2^k||_\infty$. ∎

9.2. Average Reward SMDPs

In the case of average reward, SMDP algorithms do not always have direct extensions from the MDP. We will use the stochastic shortest path and two-time-scale updating extensively here for developing the extensions. We first present some basic theory underlying these two topics. Thereafter, we apply these topics for showing convergence of RL algorithms based on value- and policy-iteration.

9.2.1 Stochastic Shortest-Path Problem

We begin with a discussion on the stochastic shortest-path problem (SSP). As stated above, we study the SSP because it forms a convenient link to the SMDP and the finite horizon problem.

The objective in the SSP is identical to that in a discounted reward MDP with the following very important differences:

1. There is a reward-free termination state. When the system reaches this state, it remains there, i.e., the state is absorbing.

2. The discount factor is set to 1.

Thus, the value function vector for the SSP for any given policy has the same definition as that of the discounted reward MDP (see Definition 11.1) with the understanding that $\lambda = 1$ and that there is a reward-free termination state into which the system is absorbed when it reaches the state. The optimal value function vector for the SSP, similarly, has the same definition as that for the discounted reward MDP (see Definition 11.2) with the same understanding.

Consistent with our assumption throughout this book, we will assume that all states in the problem are recurrent under every policy. In the context of the SSP, this implies that under any policy, the reward-free termination state is eventually reached with probability 1 regardless of the starting state. (The termination state is also an absorbing state.) This property is also known as "properness" of the policy. Thus, we will assume in our analysis that all policies in the SSP are *proper*.

Another assumption that we will make throughout in our analysis of the SSP is that there is a unique *starting* state, which is the state from which all trajectories of the SSP begin. From the perspective of simulations in which trajectories of states must be generated repeatedly, after the termination state is reached, the system is restarted at the starting state.

The following result, which we state without proof, establishes that the optimal value function vector for the SSP solves a suitable form of a Bellman optimality equation. We will call this equation the Bellman optimality equation for the SSP. The following result also relates the SSP's value function for a given policy $\hat{\mu}$ to a suitable form of the Bellman policy equation (Poisson equation).

PROPOSITION 11.42 [30] *When all admissible policies are proper and the starting state is unique, the optimal value function vector $\vec{J}^*$ for the SSP satisfies the following equation: For all $i \in \mathcal{S}'$,*

$$J^*(i) = \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) + \sum_{j \in \mathcal{S}'} p(i,a,j) J^*(j) \right], \qquad (11.95)$$

where $\mathcal{S}' = \mathcal{S} \setminus \{s^*\}$, where s^* denotes the termination (absorbing) state; the equation has a unique solution.

Further, for any given stationary policy $\hat{\mu}$, there exists a unique solution to the following equation:

$$J_{\hat{\mu}}(i) = \left[\bar{r}(i,\mu(i)) + \sum_{j \in \mathcal{S}'} p(i,\mu(i),j) J_{\hat{\mu}}(j) \right], \text{ such that for each } i \in \mathcal{S}',$$

the scalar $J_{\hat{\mu}}(i)$ will equal the expected value of the total reward earned in a trajectory starting in state i until the termination state is reached.

Note that in the above, the summation in the RHS is over $\mathcal{S}'$ which excludes the termination state. The termination state is a part of the state space $\mathcal{S}$. Because the termination state is not a part of the summation, we will have that, for one or more $i \in \mathcal{S}$,

$$\sum_{j \in \mathcal{S}'} p(i,\mu(i),j) < 1 \text{ under any given policy } \hat{\mu}.$$

The above implies that the transition probability matrix used in the Bellman equation is not stochastic, i.e., in at least one row, the elements do *not* sum to 1. This is an important property of the SSP's transition probabilities that the reader needs to keep in mind.

A Q-factor version of Eq. (11.95) would be as follows:

$$Q^*(i,a) = \left[\bar{r}(i,a) + \sum_{j \in \mathcal{S}'} p(i,a,j) \max_{b \in \mathcal{A}(j)} Q^*(j,b) \right]. \qquad (11.96)$$

We will now present another remarkable property of the SSP's transition probabilities, derived by [30] originally, but adapted in [109] to the Q-factor setting needed here. The property will be useful later in showing that the SSP's main transformation is contractive with respect to a weighted max norm.

LEMMA 11.43 *Define a scalar ϑ as follows:*

$$\vartheta \equiv \max_{i\in\mathcal{S}';a\in\mathcal{A}(i)} \frac{\upsilon(i,a)-1}{\upsilon(i,a)}, \tag{11.97}$$

where $\upsilon(i,a)$ for every $i \in \mathcal{S}'$ and $a \in \mathcal{A}(i)$ is a positive scalar. When all the stationary policies in an SSP are proper, there exist positive values for $\upsilon(.,.)$ such that $\vartheta \in [0,1)$ and the following is true:

$$\sum_{j\in\mathcal{S}'} p(i,a,j)\upsilon(j,\mu(j)) \leq \vartheta\upsilon(i,a), \text{ where } \hat{\mu} \text{ is any given stationary policy.}$$

Proof Consider a new SSP in which all the transition probabilities are identical to our SSP, but all the immediate rewards are set to 1 (except for those to the termination state, which are 0). Let $\tilde{Q}(.,.)$ be the optimal Q-factor for this new SSP. Adapting Proposition 2.1(a) in [33] to Q-factors, we have that for any given policy $\hat{\mu}$ and for all $i \in \mathcal{S}'$ and all $a \in \mathcal{A}(i)$:

$$\begin{aligned}\tilde{Q}(i,a) &= 1 + \sum_{j\in\mathcal{S}'} p(i,a,j) \max_{b\in\mathcal{A}(j)} \tilde{Q}(j,b) \\ &\geq 1 + \sum_{j\in\mathcal{S}'} p(i,a,j)\tilde{Q}(j,\mu(j)).\end{aligned} \tag{11.98}$$

Now for all (i,a)-pairs, we define $\upsilon(i,a) = \tilde{Q}(i,a)$.

From (11.98), we have that $\upsilon(i,a) \geq 1$ for all (i,a)-pairs. Clearly then, $0 \leq \vartheta < 1$. Then, for any policy $\hat{\mu}$, we have from (11.98), that

$$\sum_{j\in\mathcal{S}'} p(i,a,j)\tilde{Q}(j,\mu(j)) = \sum_{j\in\mathcal{S}'} p(i,a,j)\upsilon(j,\mu(j)) \leq \upsilon(i,a) - 1 \leq \vartheta\upsilon(i,a). \quad \blacksquare$$

We now present a key result which shows that the average reward SMDP can be viewed as a special case of the SSP under some conditions. The motivation for this is that the SSP has some attractive properties that can be used to solve the SMDP.

Transformation of SMDP to a fictitious κ-SSP: Consider any recurrent state in the SMDP, and number it K. We will call this state the *distinguished* state in the SMDP. Define the immediate rewards for a transition from i to j (where $i, j \in \mathcal{S}$) under any action $a \in \mathcal{A}(i)$ in the new problem to be:

$$r(i,a,j) - \kappa t(i,a,j),$$

where κ is any scalar (the value will be defined later). In this problem, the distinguished state will serve as the termination state as well as the starting state. This implies that once the system enters K, no further transitions are possible in that trajectory, and hence the above problem is an SSP. The fictitious SSP so generated will be called a κ-SSP. We now present the associated result which establishes the equivalence.

PROPOSITION 11.44 *Consider an average reward SMDP whose optimal average reward is ρ^* in which (i) all states are recurrent and (ii) one of the stationary deterministic policies is optimal. Construct a fictitious ρ^*-SSP associated to the SMDP using any one of the recurrent states as the termination state as well as the starting state. Then, a solution of the Bellman optimality equation for the ρ^*-SSP will also solve the Bellman optimality equation for the average reward SMDP.*

The implication of this result may not be immediately obvious. What the result implies is that (somehow) if we knew the value of ρ^* for the SMDP, we could solve the SMDP by just solving the associated fictitious ρ^*-SSP. Of course, how one can determine ρ^* *before* solving the SMDP is an important issue, but we will consider that later; for the time being, assume that there exists a mechanism to reach that value (i.e., ρ^*). As stated before, there is a strong motivation for solving the SSP in place of the SMDP: a critical transformation underlying the SSP is contractive, which leads us to a convergent RL algorithm.

Proof The proof will be presented via a succession of lemmas. We will first derive the solution of the Bellman optimality equation for a suitably derived κ-SSP (via Lemma 11.45). Thereafter, we will show that the solution of this Bellman optimality equation will also solve the associated SMDP (via Lemma 11.46).

In what follows, we will use the notion of a "cycle" within the SSP, which is to be understood as a trajectory (sequence) of states starting from any given state and ending in the first return to the same state. (Clearly, when the underlying Markov chains have random transitions, this trajectory will also be random.) Let K be the distinguished state in the SMDP, used in deriving the SSP, from where the cycle starts and where it ends. Let $K = |\mathcal{S}|$ without loss of generality. Define $\mathcal{S}' = \mathcal{S} \setminus \{K\}$.

LEMMA 11.45 *Let $R_K(\hat{\mu})$ and $T_K(\hat{\mu})$ denote the expected value of the total reward and the expected value of the total time, respectively, in one "cycle" from K to K when the policy pursued in the cycle is $\hat{\mu}$. Further, define*

$$\tilde{\rho} \equiv \max_{\hat{\mu}} \frac{R_K(\hat{\mu})}{T_K(\hat{\mu})}. \tag{11.99}$$

Now consider the associated κ-SSP where $\kappa = \tilde{\rho}$. If a bounded function $h : \mathcal{S}' \to \Re$ forms the unique solution of the Bellman optimality equation for this SSP, then $h(.)$ also solves the following equation for the SMDP: For all $i \in \mathcal{S}$, i.e., for $i = 1, 2, \ldots, K$:

$$h(i) = \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) - \tilde{\rho}\bar{t}(i,a) + \sum_{j=1}^{K} p(i,a,j)h(j) \right]; \quad h(K) = 0. \tag{11.100}$$

Proof Via Proposition 11.42, we have that some bounded function $h(.)$ solves the Bellman equation for the SSP such that for $i = 1, 2, \ldots, K-1$,

$$h(i) = \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) - \tilde{\rho}\bar{t}(i,a) + \sum_{j=1}^{K-1} p(i,a,j)h(j) \right]. \tag{11.101}$$

Note that in the RHS of the above, we have omitted $h(K)$ since K is the termination state for the SSP. Now, for any given policy $\hat{\mu}$, the value function of the state K in the SSP can be written as:

$$h_{\hat{\mu}}(K) = R_K(\hat{\mu}) - \tilde{\rho}T_K(\hat{\mu}).$$

The above follows from the definition of the value function, which says that it equals the expected value of the sum of the reward function (in this case $r(.,.,.) - \tilde{\rho}t(.,.,.)$) starting from K and ending at K. Now, again by definition,

$$\begin{aligned} h(K) &\equiv \max_{\hat{\mu}} h_{\hat{\mu}}(K) \\ &= \max_{\hat{\mu}} [R_K(\hat{\mu}) - \tilde{\rho}T_K(\hat{\mu})] \\ &= \max_{\hat{\mu}} \left[\frac{R_K(\hat{\mu})}{T_K(\hat{\mu})} - \tilde{\rho} \right] T_K(\hat{\mu}) \\ &= 0 \text{ (follows from (11.99))} \end{aligned}$$

i.e., $h(K) = 0$. Then, from Eq. (11.101), for $i = 1, 2, \ldots, K$, we have that

$$h(i) = \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) - \tilde{\rho}\bar{t}(i,a) + \sum_{j=1}^{K} p(i,a,j)h(j) \right]. \quad \blacksquare \tag{11.102}$$

LEMMA 11.46 *$\tilde{\rho} = \rho^*$.*

For the proof of this lemma, we need the following fundamental result [30, 242] (we will use the notation $\mathbf{A}^k$ to mean matrix $\mathbf{A}$ raised to the kth power):

LEMMA 11.47 *Let $\mathbf{P}_{\hat{\mu}}$ denote the transition probability matrix of a policy $\hat{\mu}$ with n states, where the matrix is stochastic. Then,*

$$\lim_{m\to\infty} \frac{\sum_{k=1}^{m} \mathbf{P}_{\hat{\mu}}^k}{m} = \mathbf{P}_{\hat{\mu}}^{**}, \text{ where } \mathbf{P}_{\hat{\mu}}^{**} \text{ is an } n \times n \text{ matrix}$$

such that $P_{\hat{\mu}}^{**}(i,j)$ denotes the steady-state probability of being in state j provided the system started in state i and policy $\hat{\mu}$ is being used in all states.

For an SMDP where all states are recurrent, each row in the matrix $\mathbf{P}_{\hat{\mu}}^{**}$ will be identical, and the (i,j)th term in every row of the matrix will equal $\Pi_{\hat{\mu}}(j)$, i.e., the steady-state probability of being in state j under $\hat{\mu}$.

Proof (of Lemma 11.46) We now define $\vec{J}_0 = \vec{h}$, and for some stationary policy $\hat{\mu}$, using $\vec{r}_{\hat{\mu}}$ to denote the vector whose ith element is $\bar{r}(i,\mu(i))$, we define the sequence $\left\{\vec{J}_k\right\}_{k=1}^{\infty}$:

$$\vec{J}_{k+1} = \vec{r}_{\hat{\mu}} + \mathbf{P}_{\hat{\mu}}\vec{J}_k, \tag{11.103}$$

where $\mathbf{P}_{\hat{\mu}}$ denotes the transition probability matrix in the SMDP associated to the policy $\hat{\mu}$. Note that this matrix is stochastic, and so is the transition probability matrix underlying any action defined in Eq. (11.102). Then, if $\vec{\tau}_{\hat{\mu}}$ denotes that vector whose ith element is $\bar{t}(i,\mu(i))$ for all $i \in \mathcal{S}$, we will show via induction that:

$$\tilde{\rho}\sum_{k=1}^{m} \mathbf{P}_{\hat{\mu}}^k \vec{\tau}_{\hat{\mu}} + \vec{J}_0 \geq \vec{J}_m. \tag{11.104}$$

Now, from Eq. (11.102), for any given stationary policy $\hat{\mu}$,

$$\vec{h} \geq \vec{r}_{\hat{\mu}} - \tilde{\rho}\mathbf{P}_{\hat{\mu}}\vec{\tau}_{\hat{\mu}} + \mathbf{P}_{\hat{\mu}}\vec{h}, \text{i.e., } \vec{J}_0 \geq \vec{r}_{\hat{\mu}} - \tilde{\rho}\mathbf{P}_{\hat{\mu}}\vec{\tau}_{\mu} + \mathbf{P}_{\hat{\mu}}\vec{J}_0$$

from which we have, using (11.103), $\tilde{\rho}\mathbf{P}_{\hat{\mu}}\vec{\tau}_{\hat{\mu}} + \vec{J}_0 \geq \vec{r}_{\hat{\mu}} + \mathbf{P}_{\hat{\mu}}\vec{J}_0 = \vec{J}_1$;

$$\text{i.e., } \hat{\rho}\mathbf{P}_{\hat{\mu}}\vec{\tau}_{\hat{\mu}} + \vec{J}_0 \geq \vec{J}_1. \tag{11.105}$$

We now assume (11.104) to be true for $m = n$, multiply both of its sides by $\mathbf{P}_{\hat{\mu}}$, and then add $\bar{r}_{\hat{\mu}}$ to both sides to obtain:

$$\tilde{\rho}\sum_{k=1}^{n}\mathbf{P}_{\hat{\mu}}^{k+1}\vec{\tau}_{\hat{\mu}} + \mathbf{P}_{\hat{\mu}}\vec{J}_0 + \bar{r}_{\hat{\mu}} \geq \mathbf{P}_{\hat{\mu}}\vec{J}_n + \bar{r}_{\hat{\mu}}, \text{ which results in}$$

$$\tilde{\rho}\sum_{k=1}^{n}\mathbf{P}_{\hat{\mu}}^{k+1}\vec{\tau}_{\hat{\mu}} + \vec{J}_1 \geq \vec{J}_{n+1}. \tag{11.106}$$

Adding (11.106) and (11.105), we have $\tilde{\rho}\sum_{k=1}^{n+1}\mathbf{P}_{\hat{\mu}}^{k}\vec{\tau}_{\hat{\mu}} + \vec{J}_0 \geq \vec{J}_{n+1}$, which completes the induction. Then dividing both sides of (11.104) by m and taking the limit as $m \to \infty$, we have from Theorem 9.7:

$$\tilde{\rho}\lim_{m\to\infty}\frac{\sum_{k=1}^{m}\mathbf{P}_{\hat{\mu}}^{k}\vec{\tau}_{\hat{\mu}}}{m} + \lim_{m\to\infty}\frac{\vec{J}_0}{m} \geq \lim_{m\to\infty}\frac{\vec{J}_m}{m}. \tag{11.107}$$

Then, using $\vec{e}$ to denote a column vector whose every element is 1,

from Lemma 11.47, we have that $\lim_{m\to\infty}\frac{\sum_{k=1}^{m}\mathbf{P}_{\hat{\mu}}^{k}\vec{\tau}_{\hat{\mu}}}{m} = \left(\sum_{i\in\mathcal{S}}\Pi_{\hat{\mu}}\tau_{\hat{\mu}}(i)\right)\vec{e} \equiv \bar{T}_{\hat{\mu}}\vec{e}.$

Also, note that from its definition $J_m(i)$ denotes the total expected reward earned starting from state i, and hence $\lim_{m\to\infty}\frac{\vec{J}_m}{m} = \bar{R}_{\hat{\mu}}\vec{e}$, where $\bar{R}_{\hat{\mu}}$ denotes the expected reward in one state transition under policy $\hat{\mu}$. Since $J_0(i)$ is bounded for every i, $\lim_{m\to\infty}\frac{\vec{J}_0}{m} = 0\vec{e}$. Then, we can write (11.107) as:

$$\tilde{\rho}\bar{T}_{\hat{\mu}}\vec{e} \geq \bar{R}_{\hat{\mu}}\vec{e}, \text{ i.e., } \tilde{\rho}\vec{e} \geq \frac{\bar{R}_{\hat{\mu}}}{\bar{T}_{\hat{\mu}}}\vec{e}, \tag{11.108}$$

where $\bar{T}_{\hat{\mu}}$ equals the expected time spent in one transition under $\hat{\mu}$; the renewal reward theorem (see Eq. (6.23); [155, 251]) implies that $\bar{R}_{\hat{\mu}}/\bar{T}_{\hat{\mu}}$ equals the average reward of the policy $\hat{\mu}$, i.e., $\tilde{\rho} \geq \rho_{\hat{\mu}}$. The equality in (11.108) applies only when one uses the policy $\hat{\mu}^*$ that uses the max operator in (11.102); i.e., only when $\tilde{\rho}$ equals the average reward of that policy in particular. Now, what (11.108) implies is that the average reward of *every* policy other than $\hat{\mu}^*$ will be less than $\tilde{\rho}$. Clearly then $\hat{\mu}^*$ must be optimal, i.e., $\tilde{\rho} = \rho_{\hat{\mu}^*} = \rho^*$. ∎

Equation (11.102) with $\tilde{\rho} = \rho^*$ is the Bellman equation for the average reward SMDP with $h(K) = 0$. ∎

Note that the distinguished state, K, in the SSP is actually a regular state in the SMDP whose value function may not necessarily equal zero. This becomes critical when designing an algorithm. Hence, we will design an algorithm based on Eq. (11.102)—with the understanding that when the value of $h(K)$ is to be used from the RHS of the equation, it is replaced by zero. Since, one cannot have two values for $h(K)$ within the same equation, it is convenient to think of a fictitious state $\bar{K}$ as the termination state and its twin K as the actual state in the SMDP. The value function of $\bar{K}$, $h(\bar{K})$, will always equal zero. In other words, the version of Eq. (11.102) useful in deriving algorithms will be: For $i = 1, 2, \ldots, K$:

$$h(i) = \max_{a \in \mathcal{A}(i)} \left[\bar{r}(i,a) - \tilde{\rho}\bar{t}(i,a) + \sum_{j=1}^{K} p(i,a,j) I(j \neq K) h(j) \right] \tag{11.109}$$

where $I(.)$, the indicator function, will return a 1 when the condition inside the brackets is true and a zero otherwise. The attractive feature of the above is that the associated transformation is fortunately contractive (with respect to the weighted max norm; which can be proved) and behaves gracefully in numerical computations.

The following result is the counterpart of Proposition 11.44 for a given policy.

PROPOSITION 11.48 *Consider an average reward SMDP in which all states are recurrent and the average reward of a policy $\hat{\mu}$ is denoted by $\rho_{\hat{\mu}}$. Construct a fictitious $\rho_{\hat{\mu}}$-SSP associated to the SMDP using any one of the recurrent states as the termination state as well as the starting state. Then, a solution of the Bellman policy equation for the $\rho_{\hat{\mu}}$-SSP, i.e.,*

$$h_{\hat{\mu}}(i) = \left[\bar{r}(i,\mu(i)) - \rho_{\hat{\mu}}\bar{t}(i,\mu(i)) + \sum_{j \in \mathcal{S}'} p(i,\mu(i),j) h_{\hat{\mu}}(j) \right], \ \forall i, \tag{11.110}$$

will also solve the Bellman policy equation for the average reward SMDP.

Proof The proof is similar to that of Lemma 11.45, using the second part of Proposition 11.42. Note that the renewal reward theorem implies that

$$\rho_{\hat{\mu}} = \frac{R_K(\hat{\mu})}{T_K(\hat{\mu})}.$$

Then, as discussed above, $h_{\hat{\mu}}(K) = R_K(\hat{\mu}) - \rho_{\hat{\mu}} T_K(\hat{\mu})$, which from the definition of $\rho_{\hat{\mu}}$ implies that $h_{\hat{\mu}}(K) = 0$. Then, we can write Eq. (11.110) as follows: For $i = 1, 2, \ldots, K$:

$$h_{\hat{\mu}}(i) = \left[\bar{r}(i, \mu(i)) - \rho_{\hat{\mu}} \bar{t}(i, \mu(i)) + \sum_{j=1}^{K} p(i, \mu(i), j) h_{\hat{\mu}}(j) \right]. \quad \blacksquare$$

The version of the Bellman policy equation useful in deriving algorithms will be: For $i = 1, 2, \ldots, K$:

$$h_{\hat{\mu}}(i) = \left[\bar{r}(i, \mu(i)) - \rho_{\hat{\mu}} \bar{t}(i, \mu(i)) + \sum_{j=1}^{K} p(i, \mu(i), j) I(j \neq K) h_{\hat{\mu}}(j) \right] \tag{11.111}$$

9.2.2 Two Time Scales

Two-time-scale algorithms are designed keeping Condition 4a of Theorem 11.24 in mind, i.e., Condition 4a is usually satisfied (obviously, it still has to be proved). However, showing Condition 4b, which is related to showing convergence of the iterates on the second time scale, usually requires significant work. We will now discuss how Condition 4b holds in a special case.

The conditions of this special case can appear rather restrictive, but fortunately hold, under a mild assumption, for R-SMART. These conditions, which will show that Condition 4b in Theorem 11.24 holds, can be briefly described as follows: (i) there is a single iterate on the slower time scale, (ii) (lockstep condition) when the slower iterate in the update on the faster time scale is fixed to some value and the algorithm is run, the slower iterate converges to the same value, and (iii) (derivative condition) the partial derivative of $G(.,.)$ with respect to the slower iterate's value is bounded and is strictly negative in every iteration.

Essentially, satisfying the lockstep condition ensures that the slower iterates "march" in a manner that maintains some sort of "order"—which allows the iterates (fast and slow) to converge gracefully to some solution (which we desire to be the optimal solution). In particular, if a fixed value of the slower iterate were used in the faster iterates (instead of the actual value of the slower iterate), the condition needs that in the resulting algorithm, the slower iterate converges to that fixed value. We now present these conditions more formally.

Condition $4b'$**:** This can be used to establish Condition 4b in Theorem 11.24.

(i) $N_2 = 1$, i.e., there is a single iterate, $Y^k \in \Re$, on the slower time scale.

(ii) **(Lockstep Condition)** There exists a unique value $y_* \in \Re$ such that if we set $Y^k = y_*$ in the update on the faster time scale for every k, i.e., the update of the iterate $\vec{X}^k$ in Eq. (11.66), then, with probability 1,

$$\lim_{k\to\infty} Y^k = y_*.$$

(iii) **(Derivative Condition)** In every iteration, the partial derivative of $G(.,.)$ with respect to the slower iterate's value is bounded and is strictly negative, i.e., for all k,

$$\frac{\partial G\left(\vec{X}^k, Y^k\right)}{\partial Y^k} < 0 \text{ and } \left|\frac{\partial G\left(\vec{X}^k, Y^k\right)}{\partial Y^k}\right| < \infty, \text{ where } Y^k \in \Re.$$

Note that the boundedness of the derivative is already ensured by the Lipschitz continuity of the function $G(.,.)$ imposed above; however, we restate it here to clarify that the derivative cannot equal negative infinity.

It is important to understand that in evaluating the lockstep condition, the value of the slower iterate is fixed to some value y_* in the faster update, while the slower update is updated as usual. Under these circumstances, the slower iterate should converge to y_*. In general, the two time scale algorithm is designed to ensure this, but this may not be true in general of all two-time-scale algorithms. We now show how Condition $4b'$ leads to 4b.

Proposition 11.49 *Consider the two-time-scale asynchronous algorithm defined via Eq. (11.66). Assume all conditions except Condition 4b in Theorem 11.24 to hold. When Condition* $4b'$ *defined above holds, Condition 4b also holds and then* y_* *is the globally asymptotically stable equilibrium for the slower ODE in (11.69).*

In other words, what this result shows is that when Condition $4b'$ holds, we have that the slower iterate converges with probability 1 to a globally asymptotically stable equilibrium of the ODE in (11.69). In the proof, we will use the shorthand notation of $x^k \to x_*$ when we mean the sequence $\left\{x^k\right\}_{k=1}^{\infty}$ converges to x_*.

Proof We will drop the iterate index l_2 from the notation of the step size for the unique slower iterate. For the proof, we need the following limits showed in [235].

For any non-negative sequence $\{\beta^n\}_{n=1}^{\infty}$ satisfying Condition 3 of Theorem 11.24, i.e., $\sum_{n=1}^{\infty} \beta^n = \infty$, and for any finite integer K,

$$\lim_{k\to\infty} \prod_{n=K}^{k+1} (1-\beta^n) = 0 \text{ for all } k > K; \tag{11.112}$$

further, for any sequence $\{\delta^k\}_{k=1}^{\infty}$ satisfying $\lim_{k\to\infty} \delta^k = 0$, we have

$$\lim_{k\to\infty} \sum_{n=K}^{k+1} \left(\prod_{m=n+1}^{k} (1-\beta^m) \right) \beta^n \delta^n = 0 \text{ for all } k > K. \tag{11.113}$$

We set up a sequence $\{\Delta^k\}_{k=1}^{\infty}$ where $\Delta^k = Y^k - y_*$ so that our goal becomes to show that $\Delta^k \to 0$. The road-map for our proof is as follows. We will first define another sequence $\{\delta^k\}_{k=1}^{\infty}$ and express Δ^k in terms of δ^k. Then, we will develop upper and lower bounds on the partial derivative of $G(\vec{X}^k, Y^k)$ with respect to Y^k. These bounds, the sequence $\{\delta^k\}_{k=1}^{\infty}$, and the limits shown in (11.112)–(11.113) will together be exploited to show that $\Delta^k \to 0$.

For a given value Y^k of the slower iterate, Condition 4a ensures that with probability 1, the sequence $\{\vec{X}^k\}_{k=1}^{\infty}$ will converge to a globally asymptotically stable critical point of the ODE in (11.68). Denote this critical point by $x_*\left(Y^k\right)$, where this point is a function of Y^k.

$$\text{Let } \delta^k = G\left(\vec{X}^k, Y^k\right) - G\left(x_*\left(Y^k\right), Y^k\right). \tag{11.114}$$

By its definition, Condition 4a ensures that $\delta^k \to 0$, since $\vec{X}^k \to x_*\left(Y^k\right)$.

We now express Δ^k in terms of $G(.,.)$ and δ^k. From the definition of δ^k above in Eq. (11.114) and the fact that the update of Y^k can be expressed as follows:

$$Y^{k+1} = Y^k + \beta^k \left(G\left(\vec{X}^k, Y^k\right)\right), \tag{11.115}$$

$$\text{we have that } \Delta^{k+1} = \Delta^k + \beta^k G\left(x_*\left(Y^k\right), Y^k\right) + \beta^k \delta^k. \tag{11.116}$$

Now the derivative condition, i.e., Condition $4b'$(iii), implies that there exist negative, upper and lower bounds on the derivative, i.e., there exist $C_1, C_2 \in \Re$ where $0 < C_1 \leq C_2$ such that:

$$-C_2\left(Y_1 - Y_2\right) \leq G\left(x_*\left(Y_1\right), Y_1\right) - G\left(x_*\left(Y_2\right), Y_2\right) \leq -C_1\left(Y_1 - Y_2\right) \tag{11.117}$$

for any $Y_1, Y_2 \in \Re$ if $Y_1 > Y_2$. If $Y_2 > Y_1$, we will have the following inequality:

$$-C_2\left(Y_1 - Y_2\right) \geq G\left(x_*\left(Y_1\right), Y_1\right) - G\left(x_*\left(Y_2\right), Y_2\right) \geq -C_1\left(Y_1 - Y_2\right). \tag{11.118}$$

We first consider the case $Y_1 > Y_2$. Now, the lockstep condition (Condition $4b'$(iii)) implies that if the faster iterates in the algorithm are updated using $Y^k \equiv y_*$ in $F\left(\vec{X}^k, Y^k\right)$, then $Y^k \to y_*$. This implies from (11.115) that $G\left(x_*\left(y_*\right), y_*\right) = 0$. Thus if, $Y_2 = y_*$ and $Y_1 = Y^k$, inequality (11.117) will lead to:

$$-C_2\Delta^k \leq G\left(x_*\left(Y^k\right), Y^k\right) \leq -C_1\Delta^k.$$

Because $\beta^k > 0$, the above leads to:

$$-C_2\Delta^k\beta^k \leq G\left(x_*\left(Y^k\right), Y^k\right)\beta^k \leq -C_1\Delta^k\beta^k.$$

The above combined with (11.116) leads to:

$$(1 - C_2\beta^k)\Delta^k + \beta^k\delta^k \leq \Delta^{k+1} \leq (1 - C_1\beta^k)\Delta^k + \beta^k\delta^k. \tag{11.119}$$

Since, the above is true for any finite integral value of k, we have that for $k = K$ and $k = K + 1$,

$$(1 - C_2\beta^K)\Delta^K + \beta^K\delta^K \leq \Delta^{K+1}; \tag{11.120}$$

$$(1 - C_2\beta^{K+1})\Delta^{K+1} + \beta^{K+1}\delta^{K+1} \leq \Delta^{K+2}. \tag{11.121}$$

Multiplying both sides of (11.120) by $(1 - C_2\beta^{K+1})$ and adding $\beta^{K+1}\delta^{K+1}$ to both sides, we have:

$$(1 - C_2\beta^K)(1 - C_2\beta^{K+1})\Delta^K + (1 - C_2\beta^{K+1})\beta^K\delta^K + \beta^{K+1}\delta^{K+1} \leq$$
$$(1 - C_2\beta^{K+1})\Delta^{K+1} + \beta^{K+1}\delta^{K+1}.$$

The above in combination with (11.121) leads to:

$$(1-C_2\beta^K)(1-C_2\beta^{K+1})\Delta^K + (1-C_2\beta^{K+1})\beta^K\delta^K + \beta^{K+1}\delta^{K+1} \leq \Delta^{K+2}.$$

Using similar arguments on the inequality on the other side of Δ^{k+1} of (11.119), we have that

$$(1 - C_2\beta^K)(1 - C_2\beta^{K+1})\Delta^K + (1 - C_2\beta^{K+1})\beta^K\delta^K + \beta^{K+1}\delta^{K+1}$$
$$\leq \Delta^{K+2} \leq (1-C_2\beta^K)(1-C_2\beta^{K+1})\Delta^K + (1-C_2\beta^{K+1})\beta^K\delta^K + \beta^{K+1}\delta^{K+1}.$$

In this style, we can also show the above result when the sandwiched term is $\Delta^{K+3}, \Delta^{K+4} \ldots$. In general, then, for any $M > K$, we obtain:

$$\prod_{n=K}^{M+1}(1-C_2\beta^n)\Delta^K + \sum_{n=K}^{M}\left(\prod_{m=n+1}^{M}(1-C_2\beta^m)\right)\beta^n\delta^n \leq \Delta^{M+1}$$

$$\leq \prod_{n=K}^{M+1}(1-C_1\beta^n)\Delta^K + \sum_{n=K}^{M}\left(\prod_{m=n+1}^{M}(1-C_1\beta^m)\right)\beta^n\delta^n.$$

We now take the limits as $M \to \infty$ on the above. Then, via Theorem 9.7 and using (11.112) and (11.113), $0 \leq \lim_{m\to\infty}\Delta^{M+1} \leq 0$. Then, Theorem 9.8 implies that $\Delta^{M+1} \to 0$. Identical arguments can now be repeated for the case $Y_2 > Y_1$, i.e., for (11.118), to obtain the same conclusion. ∎

9.2.3 R-SMART

We now discuss the convergence properties of R-SMART under some conditions. R-SMART is a two-time-scale algorithm, and we will use Theorem 11.24 to establish convergence.

We will first consider the CF-version. The core of the CF-version of R-SMART can be expressed by the following transformations. On the faster time scale we have:

$$Q^{k+1}(i,a) = Q^k(i,a) + \alpha\left[r(i,a,\xi^k) - \rho^k t(i,a,\xi^k) + \eta \max_{b\in\mathcal{A}(\xi^k)} Q^k(\xi^k,b) - Q^k(i,a)\right], \tag{11.122}$$

where ξ^k is a random variable that depends on (i,a) and k; on the slower time scale we have:

$$\rho^{k+1} = \rho^k + \beta^k I\left(a \in \arg\max_{u\in\mathcal{A}(i)} Q^k(i,u)\right)\left[\frac{TR^k}{TT^k} - \rho^k\right];$$

$$TR^{k+1} = TR^k + I\left(a \in \arg\max_{u\in\mathcal{A}(i)} Q^k(i,u)\right) r(i,a,\xi^k); \tag{11.123}$$

$$TT^{k+1} = TT^k + I\left(a \in \arg\max_{u\in\mathcal{A}(i)} Q^k(i,u)\right) t(i,a,\xi^k);$$

note that in the above, we use the indicator function in order to account for the fact that TR^k, TT^k and ρ^k are updated only when a greedy action is chosen in the simulator. We denote the optimal policy by $\hat{\mu}^*$ and that generated in the kth iteration by:

$$\mu^k(i) \in \arg\max_{a\in\mathcal{A}(i)} Q^k(i,a) \text{ for all } i \in \mathcal{S}. \tag{11.124}$$

A Note About Assumptions. The policy used for simulations in R-SMART is an exploratory policy greedy in the limit (GLIE) (discussed in Chap. 7). With a GLIE policy, in the limit, all state-action pairs are visited infinitely often; this ensures that the noise in the updates has a conditional mean of zero and that Condition 7 in Theorem 11.24 is satisfied. For R-SMART, we will also assume that part (iii) of Condition $4b'$ of Sect. 9.2.2 holds. Further, note that the CF-version works only when Assumption 7.1 from Chap. 7 is true and $\eta \in (\bar{\eta}, 1)$. Finally, all policies produce regular Markov chains (this is an assumption we make throughout this book and omit from the statements of theorems).

Before we present the convergence result, we will define some of the underlying functions necessary to invoke the ODE in question. For all (i, a) pairs,

$$F'\left(\vec{Q}^k, \rho^k\right)(i,a) = \sum_{j=1}^{|\mathcal{S}|} p(i,a,j)\left[r(i,a,j) - \rho^k \bar{t}(i,a,j) + \eta \max_{b \in \mathcal{A}(j)} Q^k(j,b)\right].$$

Then, for all (i, a) pairs, $F\left(\vec{Q}^k, \rho^k\right)(i,a) = F'\left(\vec{Q}^k, \rho^k\right)(i,a) - Q^k(i,a)$. We further define the noise term as follows: For all (i, a) pairs,

$$w_1^k(i,a) = \left[r(i,a,\xi^k) - \rho^k t(i,a,\xi^k) + \eta \max_{b \in \mathcal{A}(\xi^k)} Q^k(\xi^k, b)\right] - F'\left(\vec{Q}^k, \rho^k\right)(i,a).$$

Then, like in the case of Q-Learning, we can write the updating transformation on the faster time scale in our algorithm, (11.122), as:

$$Q^{k+1}(i,a) = Q^k(i,a) + \alpha^k\left[F\left(\vec{Q}^k, \rho^k\right)(i,a) + w_1^k(i,a)\right],$$

which is of the same form as the updating scheme for the faster time scale defined for Theorem 11.24 (replace X^k by Q^k and l by (i, a)). Then, if we fix the value of ρ^k to some constant, $\breve{\rho}$, we can invoke the following ODE as in Theorem 11.24:

$$\frac{d\vec{q}}{dt} = F(\vec{q}, \breve{\rho}), \tag{11.125}$$

where $\vec{q}$ denotes the continuous-valued variable underlying the iterate Q.

We now define the functions underlying the iterate on the slower time scale. For all (i, a) pairs,

$$G\left(\vec{Q}^k, \rho^k\right)(i,a) = \sum_{j=1}^{|\mathcal{S}|} p(i,a,j)\left[TR^k/TT^k\right] - \rho^k;$$

$$G'\left(\vec{Q}^k, \rho^k\right)(i,a) = G\left(\vec{Q}^k, \rho^k\right)(i,a) + \rho^k;$$

$$w_2^k(i,a) = \left[TR^k/TT^k\right] - G'\left(\vec{Q}^k, \rho^k\right)(i,a);$$

the above allows us to express the update on the slower time scale in the algorithm, Eq. (11.123), as:

$$\rho^{k+1} = \rho^k + \beta^k I\left(a \in \arg\max_{u\in\mathcal{A}(i)} Q^k(i,u)\right)\left[G\left(\vec{Q}^k, \rho^k\right)(i,a) + w_2^k(i,a)\right].$$

We now present our convergence result.

Proposition 11.50 *Assume that the step sizes used in the algorithm satisfy Conditions 3 and 6 of Theorem 11.24 and that GLIE policies are used in the learning. Further, assume that Assumption 7.1 from Chap. 7 holds such that $\eta \in (\bar{\eta}, 1)$. Finally, assume that part (iii) of Condition 4b′ from Sect. 9.2.2 holds. Then, with probability 1, the sequence of policies generated by the CF-version of R-SMART, $\{\hat{\mu}^k\}_{k=1}^{\infty}$, converges to $\hat{\mu}^*$.*

Proof We now need to evaluate the conditions of Theorem 11.24. Condition 1 results from the fact that (i) the partial derivative of $F(.,.)$ with Q^k is bounded and (ii) $G(.,.)$ is a linear function of ρ^k. Conditions 3 and 6 are satisfied by appropriate step-size selection. The GLIE policies ensure that Conditions 2 and 7 are met. As usual, Conditions 4 and 5 need additional work.

Condition 5: We show Condition 5 via the following result.

Lemma 11.51 *The sequence $\left\{\vec{Q}^k, \rho^k\right\}_{k=1}^{\infty}$ remains bounded with probability 1.*

Proof We will first analyze the iterate on the slower time scale and claim that:

$$|\rho^k| \leq M \text{ for all } k, \tag{11.126}$$

where M, a positive finite scalar, is defined as:

$$M = \max\left\{\frac{\max_{i,j\in\mathcal{S},a\in\mathcal{A}(i)} |r(i,a,j)|}{\min_{i,j\in\mathcal{S},a\in\mathcal{A}(i)} t(i,a,j)}, \rho^1\right\}.$$

We prove the claim for $k = 1$ as follows:

$$|\rho^2| \leq (1-\beta^1)|\rho^1| + \beta^1 |r(i,a,j)/t(i,a,j)| \leq (1-\beta^1)M + \beta^1 M = M$$

Now assuming the claim when $k = P$, we have that $|\rho^P| \leq M$. Then,

$$\begin{aligned}
|\rho^{P+1}| &\leq (1-\beta^P)|\rho^P| + \beta^P \left|\frac{TR^P}{TT^P}\right| \\
&\leq (1-\beta^P)|\rho^P| + \beta^P \frac{P \max_{i,j\in\mathcal{S},a\in\mathcal{A}(i)} |r(i,a,j)|}{P \min_{i,j\in\mathcal{S},a\in\mathcal{A}(i)} t(i,a,j)} \\
&= (1-\beta^P)|\rho^P| + \beta^P \frac{\max_{i,j\in\mathcal{S},a\in\mathcal{A}(i)} |r(i,a,j)|}{\min_{i,j\in\mathcal{S},a\in\mathcal{A}(i)} t(i,a,j)} \\
&= (1-\beta^P)M + \beta^P(M) = M.
\end{aligned}$$

In the above, we have used: $|TR^P| \leq P \max_{i,j\in\mathcal{S},a\in\mathcal{A}(i)} |r(i,a,j)|$ and $1/TT^P \leq 1/(P \min_{i,j\in\mathcal{S},a\in\mathcal{A}(i)} t(i,a,j))$. We now analyze the iterate on the faster time scale. We first note that when ρ^k is fixed to any value $\breve{\rho} \in \Re$, the transformation $F(.,\breve{\rho})$ is contractive. When $0 < \eta < 1$, the proof of this fact follows in a manner very similar to that used in showing Condition 4 in the proof of Proposition 11.29. When all the immediate rewards and times are set to 0, i.e., $r(i,a,j) = t(i,a,j) = 0$ for all $i \in \mathcal{S}$, $j \in \mathcal{S}$, and $a \in \mathcal{A}(i)$, the contractive property still holds. Now if we compute the scaled function $F_c(.,.)$, as defined in Definition 11.6, we can show (see Eq. (11.83) and its accompanying discussion) that: $F_\infty\left(\vec{Q}^k, \rho^k\right)(i,a) = \lim_{c\to\infty} F_c\left(\vec{Q}^k, \rho^k\right)(i,a)$

$$\begin{aligned}
&= \lim_{c\to\infty} \sum_{j=1}^{|\mathcal{S}|} p(i,a,j)\left[\frac{r(i,a,j) - \rho^k \bar{t}(i,a,j)}{c} + \eta \frac{\max_{b\in\mathcal{A}(j)} cQ^k(j,b)}{c}\right] \\
&\quad - \lim_{c\to\infty} \frac{cQ^k(i,a)}{c} \\
&= \eta \sum_{j=1}^{|\mathcal{S}|} p(i,a,j)\left[\max_{b\in\mathcal{A}(j)} Q^k(j,b)\right] - Q^k(i,a) \text{ (since } \rho^k \text{ is bounded);}
\end{aligned}$$

it is not hard to see that $F_\infty\left(\vec{Q}^k, \rho^k\right)$ is a special case of the transformation $F\left(\vec{Q}^k, \mathsf{a}\right)$ with the immediate rewards and times set to 0, where a is any fixed scalar. But $F\left(\vec{Q}^k, \mathsf{a}\right)$ is contractive, and hence via Theorem 11.22, the ODE $\frac{d\vec{q}}{dt} = F_\infty(\vec{q}, \mathsf{a})$ has a globally asymptotically stable equilibrium. But note that the origin is the only equilibrium point for this ODE. Then, from Theorem 11.23, it follows that the sequence $\left\{\vec{Q}^k\right\}_{k=1}^{\infty}$ must be bounded with probability 1. ∎

Condition 4: When ρ^k is fixed to any scalar $\breve{\rho}$, we have already argued that $F'(.,\breve{\rho})$ is contractive, and hence the ODE in Eq. (11.125) must have a globally asymptotically stable equilibrium. Further, the equilibrium solution is:

$$\sum_{j=1}^{|\mathcal{S}|} p(i,a,j)\left[r(i,a,j) - \rho^k \bar{t}(i,a,j) + \eta \max_{b\in\mathcal{A}(j)} Q^k(j,b)\right] \quad \forall (i,a),$$

which is clearly Lipschitz in $Q^k(.,.)$. We have thus shown that Condition 4a holds.

To show Condition 4b, consider Condition $4b'$ of Sect. 9.2.2 setting $y_* = \rho^*$, where ρ^* is the optimal average reward of the SMDP. Note that $N_2 = 1$ for our algorithm (part (i) of Condition $4b'$). Now, under Assumption 7.1, when the value of ρ^k in the faster iterate is fixed to ρ^*, the faster iterates will converge to $\vec{Q}^*$, a solution of the Bellman optimality equation. Since the slower iterate updates only when a greedy policy is chosen, in the limit, the slower iterate must converge to the average reward of the policy contained in $\vec{Q}^*$, which must be optimal. Thus, the lockstep condition (part (ii) in Condition $4b'$) holds for $y_* = \rho^*$. The derivative condition (part (iii) of Condition $4b'$) is true by assumption. Thus, Condition $4b'$ is true for our two-time-scale algorithm. Then, Proposition 11.49 implies that Condition 4b in Theorem 11.24 must hold. Theorem 11.24 can now be invoked to ensure convergence to the optimal solution of the SMDP with probability 1. ∎

We now consider the SSP-version of R-SMART. The main update on the faster time scale is: $Q^{k+1}(i,a) \leftarrow (1-\alpha)Q^k(i,a)+$

$$\alpha\left[r(i,a,\xi^k) - \rho^k t(i,a,\xi^k) + I(j \neq i^*) \max_{b\in\mathcal{A}(\xi^k)} Q^k(\xi^k,b)\right].$$

The main transformations related to the faster time scale will be: For all $i \in \mathcal{S}$ and $a \in \mathcal{A}(i)$, $F'\left(\vec{Q}^k, \rho^k\right)(i,a)$

$$= \sum_{j\in\mathcal{S}} p(i,a,j)\left[r(i,a,j) - \rho^k \bar{t}(i,a,j) + I(j \neq i^*) \max_{b\in\mathcal{A}(j)} Q^k(j,b)\right]$$

$$= \sum_{j \in \mathcal{S}'} p(i,a,j) \left[r(i,a,j) - \rho^k \bar{t}(i,a,j) + \max_{b \in \mathcal{A}(j)} Q^k(j,b) \right]. \qquad (11.27)$$

The definition in (11.27) will be used in our analysis, since by summing over the set of states $\mathcal{S}' = \mathcal{S} \setminus \{i^*\}$, which excludes the termination state i^*, we can exploit a contractive property of $F'(.)$. Also, all $i \in \mathcal{S}$ and $a \in \mathcal{A}(i)$:

$$F\left(\vec{Q}^k, \rho^k\right)(i,a) = F'\left(\vec{Q}^k, \rho^k\right)(i,a) - Q^k(i,a).$$

$$w_1^k(i,a) = \left[r(i,a,\xi^k) - \rho^k t(i,a,\xi^k) + I(j \neq i^*) \max_{b \in \mathcal{A}(\xi^k)} Q^k(\xi^k, b) \right] - F'\left(\vec{Q}^k, \rho^k\right)(i,a).$$

Then, like in the case of Q-Learning, we can write the updating transformation on the faster time scale in our algorithm as:

$$Q^{k+1}(i,a) = Q^k(i,a) + \alpha^k \left[F\left(\vec{Q}^k, \rho^k\right)(i,a) + w_1^k(i,a) \right] \quad \forall (i,a);$$

the updates on the slower time scale and the associated functions will be identical to those for the CF-version. Also, the policy generated by the algorithm in the kth iteration will be given as in Eq. (11.124). Our main convergence result is as follows:

PROPOSITION 11.52 *Assume that the step sizes used in the algorithm satisfy Conditions 3 and 6 of Theorem 11.24 and that GLIE policies are used in the learning. Further assume that part (iii) of Condition* $4b'$ *from Sect. 9.2.2 holds. Then, with probability 1, the sequence of policies generated by the SSP-version of R-SMART,* $\{\hat{\mu}^k\}_{k=1}^{\infty}$*, converges to* $\hat{\mu}^*$.

The result above assumes that all states are recurrent under every policy and that one of the stationary deterministic policies is optimal.

Proof We will first show (via Lemma 11.53) that the transformation $F'(.)$ underlying the faster iterate, as defined in (11.27), is contractive with respect to a weighted max norm. The rest of the proof will be very similar to that of Proposition 11.50. Note, however, that because we use a distinguished state i^* as an absorbing state in the algorithm, we will essentially be solving an SSP here; but Proposition 11.44 will ensure that the SSP's solution will also solve the Bellman optimality equation for the SMDP concerned and we will be done. We first show the contractive property.

LEMMA 11.53 *When ρ^k is fixed to any constant $\breve{\rho} \in \Re$, the transformation $F'\left(\vec{Q}^k, \rho^k\right)$, as defined in (11.27), is contractive with respect to a weighted max norm.*

Proof Since ρ^k will essentially be a constant, we will drop ρ^k from our notation $F'\left(\vec{Q}^k, \rho^k\right)$ and use $F'\left(\vec{Q}^k\right)$ instead. Consider two vectors $\vec{Q}_1^k$ and $\vec{Q}_2^k$ in $\Re^N$. From the definition of $F'(.)$:

$$F'\left(\vec{Q}_1^k\right)(i,a) - F'\left(\vec{Q}_2^k\right)(i,a) = \sum_{j \in \mathcal{S}'} p(i,a,j)\left[\max_{b \in \mathcal{A}(j)} Q_1^k(j,b) - \max_{b \in \mathcal{A}(j)} Q_2^k(j,b)\right].$$

Then, for any (i,a)-pair: $\left|F'\left(\vec{Q}_1^k\right)(i,a) - F'\left(\vec{Q}_2^k\right)(i,a)\right|$

$$\begin{aligned}
&\leq \sum_{j \in \mathcal{S}'} p(i,a,j)\left|\max_{b \in \mathcal{A}(j)} Q_1^k(j,b) - \max_{b \in \mathcal{A}(j)} Q_2^k(j,b)\right| \\
&\leq \sum_{j \in \mathcal{S}'} p(i,a,j) \max_{b \in \mathcal{A}(j)} |Q_1^k(j,b) - Q_2^k(j,b)| \\
&\leq \sum_{j \in \mathcal{S}'} p(i,a,j) \max_{j \in \mathcal{S}, b \in \mathcal{A}(j)} |Q_1^k(j,b) - Q_2^k(j,b)| \\
&= \sum_{j \in \mathcal{S}'} p(i,a,j) \upsilon(j,b) ||\vec{Q}_1^k - \vec{Q}_2^k||_\upsilon \text{ for any } b \in \mathcal{A}(j) \\
&\leq \vartheta \upsilon(i,a) ||\vec{Q}_1^k - \vec{Q}_2^k||_\upsilon
\end{aligned}$$

with $0 \leq \vartheta < 1$, where the last but one line follows from the definition of the weighted max norm (see Appendix) and the last line from Lemma 11.43 and the definition of ϑ in Eq. (11.97). Then, we have that $\frac{\left|F'\left(\vec{Q}_1^k\right)(i,a) - F'\left(\vec{Q}_2^k\right)(i,a)\right|}{\upsilon(i,a)} \leq \vartheta ||\vec{Q}_1^k - \vec{Q}_2^k||_\upsilon$. Via usual arguments, $||F'\vec{Q}_1^k - F'\vec{Q}_2^k||_\upsilon \leq \vartheta ||\vec{Q}_1^k - \vec{Q}_2^k||_\upsilon$. ∎

Conditions 1, 2, 3, and 6 and the boundedness of the slower iterate follow in a manner similar to that used in the previous result. Also, the contractive property can then be similarly used to show boundedness of the faster iterate. Since the transformation, $F'(.,.)$ is contractive, we have that for a fixed value of ρ^k, the associated ODE has a globally asymptotically stable equilibrium. Also, it can argued as before that the equilibrium will be Lipschitz, thus proving Condition 4a.

To show Condition 4a, we will use Condition $4b'$. We first note that the Q-factor version of Eq. (11.109), which is the value-function version of the SSP's Bellman optimality equation, can be derived as follows when K in the notation of that equation is replaced by i^*:

$$Q^{k+1}(i,a) = \bar{r}(i,a) - \tilde{\rho}\bar{t}(i,a) + \sum_{j\in\mathcal{S}} p(i,a,j)\left[I(j\neq i^*)\max_{b\in\mathcal{A}(i)} Q^k(j,b)\right] \quad \forall (i,a).$$

When $\tilde{\rho}$ is fixed to ρ^* in the faster iterate, it can be argued (as in the previous result) that the faster iterates will converge to the solution of the above equation (under the condition that $\tilde{\rho} = \rho^*$). Via Proposition 11.44, we know that the solution of the above equation with $\tilde{\rho} = \rho^*$ will also generate a solution to the Bellman optimality equation for SMDPs; consequently the slower iterates will converge to ρ^*. This establishes the lockstep condition in $4b'$. The derivative condition is true by assumption and $N_2 = 1$. Then, by invoking Proposition 11.49, we have via Theorem 11.24 that ρ^k converges to ρ^* with probability 1. As a result, repeating the argument above, the Q-factors must converge with probability 1 to their optimal solution. ∎

9.2.4 Q-P-Learning

We now analyze the Q-P-learning algorithm for average reward SMDPs. As usual, in the case of algorithms based on policy iteration, our analysis will be restricted to the policy evaluation phase. We begin with analyzing the CF-version. The equation that this algorithm seeks to solve is the η-version of the Bellman policy equation for a given policy $\hat{\mu}$, i.e., Eq. (7.36).

PROPOSITION 11.54 *Assume the step sizes and the action selection to satisfy Conditions 3, 6, and 7 of Theorem 11.21. Further assume that Assumption 7.2 (Chap. 7) is true for the SMDP concerned and η is chosen such that $\eta \in (\bar{\eta}, 1)$. Then, with probability 1, the sequence of iterates generated within Step 3 of the CF-version of Q-P-Learning for average reward SMDPs, $\{\vec{Q}^n\}_{n=1}^{\infty}$, converges to the unique solution of Equation (7.36), i.e., to the value function vector associated to policy $\hat{\mu}$.*

Proof The proof will be very similar to that of Q-P-Learning for discounted reward MDPs. In Step 2, an estimate of the average reward of the policy being evaluated, $\hat{\mu}$ (the policy is contained in the current values of the P-factors), is generated. The transformation $F'(.)$ for this algorithm will be as follows:

$$F'\left(\vec{Q}^n\right)(i,a) = \sum_{j=1}^{|\mathcal{S}|} p(i,a,j)\left[r(i,a,j) - \rho_{\hat{\mu}}\bar{t}(i,a,j) + \eta Q^n(j,\mu(j))\right].$$

Now $F'(.)$ is contractive, using arguments similar to those in the proof of Proposition 11.40; then, under Assumption 7.2, the rest of this proof is very similar to that of Proposition 11.40. ∎

In the SSP-version, the algorithm seeks to solve is as follows:

$$\begin{aligned} Q^{n+1}(i,a) &= \sum_{j\in\mathcal{S}} p(i,a,j)\left[r(i,a,j) - \rho_{\hat{\mu}}\bar{t}(i,a,j) + I(j\neq i^*)Q^n(j,\mu(j))\right] \\ &= \sum_{j\in\mathcal{S}'} p(i,a,j)\left[r(i,a,j) - \rho_{\hat{\mu}}\bar{t}(i,a,j) + Q^n(j,\mu(j))\right], \end{aligned} \tag{11.128}$$

where $\mathcal{S}' = \mathcal{S}\setminus\{i^*\}$. This is the Q-factor version of the Bellman policy equation for SMDPs that uses the SSP connection (see Eq. (11.111)). As usual, we will assume that all states are recurrent.

Proposition 11.55 *Assume the step sizes and the action selection to satisfy Conditions 3, 6, and 7 of Theorem 11.21. Then, with probability 1, the sequence of iterates generated within Step 3 of the CF-version of Q-P-Learning for average reward SMDPs, $\{\vec{Q}^n\}_{n=1}^{\infty}$, converges to the unique solution of Equation (11.128), i.e., to the value function vector associated to policy $\hat{\mu}$.*

Proof The proof will be along the lines of that for Q-P-Learning for discounted reward MDPs (see Proposition 11.40). We will first show that the transformation $F'(.)$ is contractive. We define $F'(.)$ as follows:

$$F'\left(\vec{Q}^n\right) = \sum_{j\in\mathcal{S}'} p(i,a,j)\left[r(i,a,j) - \rho_{\hat{\mu}}\bar{t}(i,a,j) + Q^n(j,\mu(j))\right].$$

The following result shows the contraction property.

Lemma 11.56 *The transformation $F'(.)$ is contractive.*

Proof From the definition of $F'(.)$:

$$F'\left(\vec{Q}^n_1\right)(i,a) - F'\left(\vec{Q}^n_2\right)(i,a) = \sum_{j\in\mathcal{S}'} p(i,a,j)\left[Q^n_1(j,\mu(j)) - Q^n_2(j,\mu(j))\right].$$

Then, for any (i,a)-pair: $\left|F'\left(\vec{Q}^n_1\right)(i,a) - F'\left(\vec{Q}^n_2\right)(i,a)\right|$

$$\leq \sum_{j\in\mathcal{S}'} p(i,a,j)\left|Q^n_1(j,\mu(j)) - Q^n_2(j,\mu(j))\right|$$

$$
\begin{aligned}
&\leq \sum_{j \in \mathcal{S}'} p(i,a,j) \max_{j \in \mathcal{S}, b \in \mathcal{A}(j)} |Q_1^n(j,b) - Q_2^n(j,b)| \\
&= \sum_{j \in \mathcal{S}'} p(i,a,j) v(j,b) ||\vec{Q}_1^n - \vec{Q}_2^n||_v \text{ for any } b \in \mathcal{A}(j) \\
&\leq \vartheta v(i,a) ||\vec{Q}_1^n - \vec{Q}_2^n||_v
\end{aligned}
$$

with $0 \leq \vartheta < 1$, where the last but one line follows from the definition of the weighted max norm (see Appendix) and the last line from Lemma 11.43 and the definition of ϑ in Eq. (11.97). Then, we have that $\frac{|F'(\vec{Q}_1^n)(i,a) - F'(\vec{Q}_2^n)(i,a)|}{v(i,a)} \leq \vartheta ||\vec{Q}_1^n - \vec{Q}_2^n||_v$. Via usual arguments, $||F'\vec{Q}_1^n - F'\vec{Q}_2^n||_v \leq \vartheta ||\vec{Q}_1^n - \vec{Q}_2^n||_v$. ∎

The solution to which the algorithm converges is that of an associated SSP, but via Proposition 11.48, we have that the solution also solves the SMDP's Bellman policy equation. ∎

10. Reinforcement Learning for Finite Horizon: Convergence

We will analyze the finite horizon Q-Learning algorithm when the discounting factor is 1, there is a unique starting state, and all states are recurrent under every policy. Under these conditions, the finite horizon problem can be viewed as a special case of the Stochastic Shortest Path (SSP) problem.

In the finite horizon problem, the so-called stage, s, is incremented by one after every state transition, and there is a finite number, T, of stages in the problem. When an action a is chosen in state i at the sth stage, the core of the algorithm in its kth iteration can be written as:

$$
\begin{aligned}
Q^k(i,s,a) &\leftarrow (1-\alpha^k) Q^k(i,s,a) \\
&\quad + \alpha^k \left[r(i,s,a,\xi^k,s+1) + \max_{b \in \mathcal{A}(\xi^k,s+1)} Q^k(\xi^k,s+1,b) \right],
\end{aligned}
$$

where ξ^k is the random state to which the system transitions; it depends on i, s and a. Further, note that we have the boundary condition: $Q^k(i,T+1,b) = 0$ for all $i \in \mathcal{S}$ and $b \in \mathcal{A}(i,T+1)$. Let the policy generated in the kth iteration be defined by:

$$
\mu^k(i,s) \in \arg\max_{a \in \mathcal{A}(i,s)} Q^k(i,a,s) \text{ for all } i \in \mathcal{S} \text{ and for } s = 1,2,\ldots,T.
$$

Let the optimal policy be denoted by $\hat{\mu}^*$. For the convergence result, we define $F'(.)$ as follows: $F'\left(Q^k(i,s,a)\right) =$

$$\sum_{j\in\mathcal{S}} p(i,s,a,j,s+1)\left[r(i,s,a,j,s+1) + \max_{b\in\mathcal{A}(j,s+1)} Q^k(j,s+1,b)\right].$$

where $p(i,s,a,j,l) = 0$ whenever $l \neq (s+1)$. Then, if a fixed point exists for $F'(\vec{x}) = \vec{x}$, with the boundary condition, we are interested in obtaining convergence to it, since it should generate an optimal solution to our problem [30]. The convergence result for finite-horizon Q-Learning is as follows.

Proposition 11.57 *When the step sizes and action selection used in the algorithm satisfy Conditions 3, 6, and 7 of Theorem 11.21, all the states in the system are recurrent under every policy, and there is a unique starting state, with probability 1, the sequence of policies generated by the finite-horizon Q-Learning algorithm, $\{\hat{\mu}^k\}_{k=1}^{\infty}$, converges to $\hat{\mu}^*$.*

Proof The proof will be very similar to that of infinite-horizon Q-Learning. Conditions 1, 2, 3, 6, and 7 follow in a manner identical (or very similar) to that shown for Q-Learning (Theorem 11.29). However, Conditions 4 and 5 need additional work. We now define $F(.)$ as follows: $F\left(\vec{Q}^k\right)(i,s,a) = F'\left(\vec{Q}^k\right)(i,s,a) - Q^k(i,s,a)$, where the terms $Q^k(.,.,.)$, we will assume, are mapped to a vector. We further define a transformation $f'(.)$ as follows:

$$f'\left(\vec{Q}^k\right)(i,s,a) = \left[r(i,s,a,\xi^k,s+1) + \max_{b\in\mathcal{A}(\xi^k,s+1)} Q^k(\xi^k,s+1,b)\right].$$

Now, if we define the noise term as:

$$w^k(i,s,a) = f'\left(\vec{Q}^k\right)(i,s,a) - F'\left(\vec{Q}^k\right)(i,s,a),$$

then, we can write the updating transformation in our algorithm as:

$$Q^{k+1}(i,s,a) = Q^k(i,s,a) + \alpha^k\left[F\left(\vec{Q}^k\right)(i,s,a) + w^k(i,s,a)\right],$$

which is of the standard form. Then, we can invoke the following ODE as in Condition 4 of Theorem 11.21:

$$\frac{d\vec{q}}{dt} = F(\vec{q}), \tag{11.129}$$

where $\vec{q}$ denotes the continuous-valued variable underlying the iterate Q.

If we treat the state-stage pair as the state, i.e., $\mathsf{I} \equiv (i, s)$ and $\mathsf{J} \equiv (j, s+1)$, then the problem can be viewed as a special case of an SSP problem in which every policy is proper and the starting state is unique; the transition probability of the SSP will be defined as $p(\mathsf{I}, a, \mathsf{J})$ and the transition reward as $r(\mathsf{I}, a, \mathsf{J})$. Setting $\rho^k = 0$ in Lemma 11.53, we have from that result that $F'(.)$ is contractive (with respect to some weighted max norm). This implies from Theorem 11.22 that the associated ODE in Eq. (11.129) has a unique globally asymptotically stable equilibrium (Condition 4) and that the equilibrium is Lipschitz; the existence of the equilibrium can be used to show via Theorem 11.23, using arguments familiar to the reader by now, that the iterates remain bounded with probability 1 (Condition 5). Then, we have convergence to the desired fixed point with probability 1. ∎

11. Conclusions

This chapter was meant to introduce the reader to some basic results in the convergence theory of DP and RL. While the material was not meant to be comprehensive, it is hoped that the reader has gained an appreciation for the formal ideas underlying the convergence theory. Our goal for DP was to show that the solutions of the Bellman equation are useful and that the algorithms of policy and value iteration converge. In RL, our goal was very modest—only that of presenting convergence of some algorithms via key results from stochastic approximation theory (based on ODEs and two-time- scale updating).

Bibliographic Remarks. In what follows, we have summarized some references to convergence analysis of DP and RL theory. The following account is not comprehensive. Also, our discussion is heavily borrowed from the literature, and despite our best efforts, we may have missed some references for which we apologize in advance.

DP theory. Our accounts, which show that a solution of the Bellman optimality equation for both discounted and the average reward case is indeed optimal, follow Vol II of Bertsekas [30]. The convergence of value iteration, via the fixed point theorem, is due to Blackwell [42]. The convergence proof for policy iteration for discounted reward, presented in this book, follows from Vol II of Bertsekas [30] and the references therein. The analysis of policy iteration for average reward is from Howard [144]. The discussion on span semi-norms and the statement of Theorem 11.15 is from Puterman [242]. Our account of convergence of value iteration and relative value iteration for average reward is based on the results in Gosavi [119].

RL theory. For RL, the main result (Proposition 11.21) related to synchronous conditions follows from [46, 136, 49]; see [48] for a textbook-based treatment of this topic. Two-time-scale asynchronous convergence result is based on results from Borkar [45, 46].

Convergence of Q-Learning has appeared in a number of papers. Some of the earliest proofs can be found in [300, 151, 290]. A proof based on ODEs was developed later in [49], which used a result from [46]. The ODE analysis requires showing boundedness of iterates. In our account, the proof based on basic principles for showing boundedness of Q-Learning is from [111]. See also [300] for yet another boundedness proof for Q-Learning. The general approach to show boundedness (that works for many RL algorithm and has been used extensively here) is based on showing the link between a contraction and a globally asymptotically stable equilibrium of the ODE concerned (Theorem 11.22 above is from [48]) and a link between the asymptotically stable equilibrium and boundedness (Theorem 11.23 above is from [49]). The eigenvalue-based analysis for showing boundedness, which exploits these results, can be found in [119]. The analysis of finite convergence of Q-Learning is from [109]. The convergence of Relative Q-Learning can be found in [2, 49].

The "lockstep" condition in Condition $4b'$ of Proposition 11.49 is not our original work; it can be found in many two time scale algorithms, and has been explicitly used in the proofs of an SSP algorithm in [2] and R-SMART [119]. However, it was never presented *in the general format* that we present here, which makes it a candidate for application in two time scale stochastic approximation algorithms; when the condition holds, it should further ease the analysis of a two-time-scale algorithm. The derivative condition in Condition $4b'$ was formally used in [119], but is also exploited (indirectly) in [2].

SSP. The connection between the SSP and the MDP was made via a remarkable result in Bertsekas [30, vol I]. The result connecting the SSP to the SMDP (Proposition 11.44), which is the basis of the SSP-versions of R-SMART and Q-P-Learning, is from Gosavi [119]. The analysis of R-SMART for the SSP-version and the CF-version can be found in [119]. An analysis which assumed that ρ starts in the vicinity of ρ^* can be found in [110]. The convergence of the SSP-versions and regular versions of Q-P-learning for average reward can be collectively found in Gosavi [109, 118]. The contraction property of the SSP's transformation, shown here via Lemma 11.53 for the Bellman optimality equation and Lemma 11.56 for the Bellman policy equation, are extensions of results for the value function from [33] to the Q-factor and are based on [109]. Lemma 11.43 is also a Q-factor extension of a result in [33], and can be found in [109]. Lemma 11.46 for SMDPs, based on the renewal reward theorem, is from Gosavi [119]. Our analysis of finite-horizon Q-Learning, under the conditions of a unique starting state and proper policies, is based on the contraction argument in Lemma 11.53. See [332] for a more recent analysis of the SSP in which some of these conditions can be relaxed. See also [38] for an analysis of the finite horizon algorithm under conditions weaker than those imposed here.

Miscellaneous. The convergence of API for MDPs (discounted reward) and Q-P-Learning for MDPs (average and discounted reward) is from [120]. See [33] and references therein for convergence analysis with function approximation. The convergence of SARSA has been established in [277].

Chapter 12

CASE STUDIES

1. Chapter Overview

In this chapter, we will describe some case studies related to simulation-based optimization. We will provide a general description of the problem and of the approach used in the solution process. For more specific numeric details, the readers are referred to the references provided. We present three case studies for model-free simulation optimization related to airline revenue management, preventive maintenance of machines, and buffer allocation in production lines in detail. We also present a heuristic rule in each case, which can be used for benchmarking the simulation-optimization performance. Such heuristics are typically problem-dependent and may produce high-quality solutions. Also, without a benchmark, it is difficult to gage the performance of the simulation-optimization algorithm on large-scale problems where the optimal solution cannot be determined. We enumerate numerous other case studies, pointing the reader to appropriate references for further reading.

2. Airline Revenue Management

Revenue management (or yield management) is the science underlying maximization of profits via selling seats on an airplane in the most effective manner [212, 229, 294]. Seat allocation and overbooking are two critical aspects of the revenue management problem,

A. Gosavi, *Simulation-Based Optimization*, Operations Research/Computer Science Interfaces Series 55,
DOI 10.1007/978-1-4899-7491-4_12

and the key to making profit lies in controlling seat allocation and overbooking properly. Our discussion here will follow [123, 127, 109].

It is known to airlines that not every customer has the same expectation from the service provided. For instance, some customers like direct flights, while some are willing to fly with a few stopovers if it means a cheaper ticket. More importantly, some customers book tickets considerably in advance of their journey, while some (usually business related travelers) tend to book a few days before the flight's departure. Airline companies take advantage of these differences by selling seats of a flight at different prices. Thus for instance, a customer who desires fewer stopovers or arrives late in the booking process is charged a higher fare. A customer (generally a business traveler) who needs a ticket that is refundable, usually because of a higher likelihood of the cancellation of his/her trip, is also charged a higher fare.

All of the above factors lead to a situation where airlines *internally* (without telling the customers) divide passengers into different **fare classes** or **products** based on their needs and the circumstances. Passengers within the same fare class (or product) pay the same (or roughly the same) fare.

It makes business sense to place upper limits on the number of seats to be sold in each fare class. This ensures that some seats are reserved for higher fare class passengers (that provide higher revenues), who tend to arrive late in the booking period. Also, it is usually the case that demand for lower fare classes is higher than that for the higher fare classes, and unless some limits are imposed, it is likely that the plane will be primarily occupied by the lower fare-class passengers. This is an undesirable situation for airline profits. On the other hand, if the limits imposed are very high, it is quite possible that the plane will not be full at takeoff. Thus finding the right values for these limits becomes an important problem for the carrier.

The "overbooking" aspect adds to the complexity of this problem. Some customers cancel their tickets and some fail to show up at the flight time (no-shows). As a result, airlines tend to overbook (sell more tickets than the number of seats available) flights, anticipating such events. This can minimize the chances of flying planes with empty seats. It may be noted that a seat in an airplane, like a hotel room or vegetables in a supermarket, is a perishable item, and loses all its value as soon as the flight takes off. However, the downside of excessive overbooking is the risk of not having sufficient capacity for all the ticket-holders at takeoff. When this happens, i.e., when the number of passengers who show up exceeds the capacity because of overbooking, airlines are forced to deny (bump) boarding request to the extra ticket-holders and pay a penalty in two ways: directly in the form of financial compensations to the inconvenienced passengers, and sometimes indirectly through loss of customer goodwill. Usually loss of customer goodwill can be avoided by finding volunteers, who are willing to fly on a later flight; nobody is inconvenienced this way, but the volunteers get monetary benefits and the company loses money. Hence, a careful choice of overbooking policy that maximizes the revenue is required.

A major factor used in classification is the time of the request for reservation. Passengers, who book in advance, get discount fares; in other words they are made to belong to lower fare classes (nobody has complaints with this, of course). Those who come later have to pay higher fares, that is, they get allocated to the higher fare classes. (Note that a higher fare class does not mean that there is any seating advantage.) If classification is carried out by the airline company on the basis of this factor alone, passengers in the the real-world system should arrive *sequentially*, i.e., passengers in the lowest classes arrive first, followed by passengers in the next higher class, and so on. This is however not necessarily true, and hence our model should account for *concurrent* arrivals of different types of passengers.

We now consider another important factor that affects the fare class (product). To understand how an itinerary-based passenger classification works, consider Fig. 12.1, which shows a small hub-and-spoke network with Chicago as the hub (center) and Miami, Boston, and Denver as spokes. In such a network, passengers who want to fly from one spoke city (origin) to another spoke city (destination) take two flights: one from the origin to the hub and then a second from the hub to the destination. With larger networks, it is possible that one may have to take three flights. Each individual flight in the network is called a **leg**. Thus in the flight from Chicago to Miami, it is possible that there are passengers whose origin was Chicago, and there are passengers whose origin was Denver or Boston. Thus on the same flight, one may have passengers who are either flying directly from the origin to their destination *and* those for whom this is just one leg in their longer itinerary. Thus, in summary, the number of stop-overs in their entire journey for passengers on a given leg may not be the same. This number can affect the fare class (product).

A third factor that can affect the fare class (product) is the ability of the traveler to cancel a ticket. If a ticket is refundable, which means it can be canceled with little or no penalty, it usually costs more. Thus for instance, for the same origin-destination itinerary, one traveler may have paid a non-refundable fare of \$200 and another may have paid a refundable fare of \$300.

All of these factors are together used to allocate the customer to a unique fare class (product). The problem faced by the airline is to determine whether to accept or deny the boarding request when a customer "arrives." Usually, this means that the customer logs on to a web-site, such as Travelocity or the website of the carrier (or calls a travel agent), and enters data related to time and place of origin and destination. The web-site, which is backed by a computer program, then determines the appropriate product that the customer is seeking and then provides all the available options the customer has.

The computer program that we mention, which is usually a sophisticated data-base system with a web interface, has access to the optimal booking limits for each type of product, and it offers information to the customer based on that. The optimization for the booking limits is oftentimes performed every night. Of course, since with every accepted request, the status of the system changes (i.e., the number of seats sold of each product), updates of this nature must be done continuously for the system to work properly.

Simultaneous perturbation was used in [127] to solve the network problem for booking-limit optimization, while the leg-based problem was solved via RL in [123, 109]. Other works that use simulation directly or indirectly include [65, 175, 76, 159, 306, 36]. Comprehensive overviews of the revenue management problem can be found in the literature. A recent survey is [66]; see also [201]. This is an area that has attracted significant research interest in the operations management community recently. We now discuss some additional aspects of modeling this problem.

An SMDP model. For a control-theoretic (dynamic) RL-based model to this problem, one must first define the state space properly. Usually, some kind of approximation of the state space using appropriate features and basis function is needed (see [109] for one example). The state space must account for the seats sold for each product and perhaps the current time in the booking period. (The booking period is the time during which requests for tickets to a particular product can be entertained.) The action space is: {*Accept, Deny*}. The time horizon for the underlying problem could be finite and it could equal the booking period. Otherwise, the time horizon could be infinite in which when the flight takes off, the system reverts to the start of the booking period. The objective function depends on the type of time horizon.

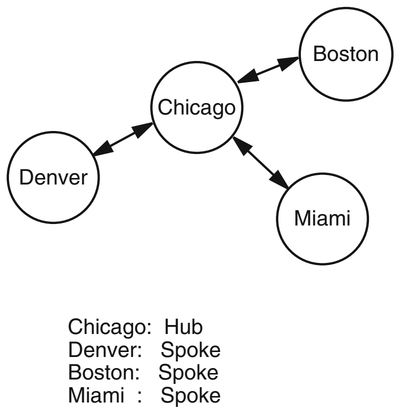

Figure 12.1. A typical hub-and-spoke network in which Chicago serves as the hub and Miami, Boston, and Denver serve as the spokes

A Parametric Optimization Model. The seat-allocation problem can be solved as a parametric-optimization problem. The objective function is the average revenue per unit time (for infinite time horizon) or the average revenue per flight (for finite time horizon). If BL_i denotes the booking limiting for the ith fare class, then the problem is: Maximize $f(BL_i, BL_2, \ldots, BL_n)$ where $f(.)$, the objective function, is estimated using simulation. The simulator is combined with an optimization algorithm, e.g., some meta-heuristic or simultaneous perturbation.

We now present some heuristic algorithms (not based on simulation optimization) that have been used successfully in the industry to solve these problems. Naturally, these heuristics form great benchmarks for simulation-optimization approaches.

EMSR-a. The EMSR (Expected Marginal Seat Revenue) heuristic was developed in Belobaba [26]. It is rooted in an equation by Littlewood [190]. This heuristic is widely used in the industry. According to this heuristic, when a customer requests a ticket, he/she should be given a reservation only if the number of seats sold in the class to which he/she belongs is less than a so-called *booking limit* for that class. The booking limits for any given class can be obtained by solving the following equation:

$$\mathsf{P}(Y_i > S_j^i) = V_j/V_i \quad \text{for} \quad j = 1, 2, \ldots, n-1, \;\; i = j+1, j+2, \ldots, n,$$

where Y_i denotes the random number of requests for class i that will arrive during the booking horizon, S_j^i denotes the number of seats to be protected from class j for a higher class i, V_i and V_j are the fares for the classes i and j, respectively, and n is the number of classes. In order to solve the equation, one must know the distribution of Y_i for $i = 1, 2, \ldots, n$. Examples of distributions here are normal, Poisson, and Erlang. In order to solve this equation, one may use the inverse of the cumulative distribution functions (*cdfs*) of the random variables Y_i.

In the equation above, the numbering convention that we have used implies that if $i > j$, then $V_i > V_j$. *In other words, higher the number (index) of the fare class (product), the greater the revenue; some books adopt the opposite convention.*

The unknown quantities in the equation above are the values of S_j^i, i.e., the protection levels. These protection levels are subsequently used to compute the booking limits as follows: For $j = 1, 2, \ldots, n-1$,

$$BL_j = C - \sum_{i=j+1}^{n} S_j^i \text{ and } BL_n = C,$$

where C is the capacity of the plane. The booking limit for class n (the highest class) is C since these customers are always desirable. Cancellations and no-shows are accounted for by multiplying the capacity of the aircraft with an overbooking factor. Thus, if C is the capacity of the flight and C_p is the probability of cancellation, then to account for that, the modified capacity of the aircraft is calculated to be $C/(1 - C_p)$, which replaces C in the calculations above. This is done so that at take-off the expected number of customers present roughly equals the capacity of the plane.

EMSR-b. A variant of the above heuristic, called EMSR-b (also credited to Belobaba; see also [27]), is also popular in the literature. Under some conditions, it can outperform EMSR-a (see [294] for a comprehensive treatment of this issue). In this heuristic, the attempt is to convert the n-fare-class problem into one with only *two* classes during solution for each class. Let

$$\bar{Y}_i \equiv \sum_{j=i}^{n} Y_j$$

denote the sum of the random demands of all classes above i and including i (again note that a higher class provides higher revenue by our convention). Then, we can define a so-called *aggregate revenue* for the ith class to be the weighted mean revenue of all classes above i and including i as follows:

$$\bar{V}_i = \frac{\sum_{j=i}^{n} V_j \mathsf{E}[Y_j]}{\sum_{j=i}^{n} \mathsf{E}[Y_j]}.$$

Then Littlewood's equation for EMSR-b is:

$$\bar{V}_i = \bar{V}_{i+1} \mathsf{P}\left(\bar{Y}_{i+1} > S_{i+1}\right)$$

for $i = 1, 2, \ldots, n-1$, where S_i denotes the protection level. Note that here the protection levels are defined differently than in EMSR-a. The above equation can be solved using the inverse *cdf*, like in EMSR-a, to obtain the values of $S_2, S_3, \ldots, S_n$. There is no protection level for the lowest class 1. Then, if C denotes the capacity of the plane, the booking limit for the ith class is calculated to be:

$$BL_i = \max\{C - S_{i+1}, 0\} \text{ for } i = 1, 2, \ldots, n-1; BL_n = C.$$

Like in EMSR-a, instead of using C above, we could use $C/(1 - C_p)$.

EMSR-a and EMSR-b are used to determine booking limits for an individual leg. However, since the leg is a part of the entire network, in network problems, ideally, the booking limits on each leg are not computed in isolation. The following approach, popular in industry, can be used for the entire network and has been used as a benchmark for simulation optimization in [127].

DAVN-EMSR-b. DAVN-EMSR-b, as the name suggests, is a combination of DAVN, which stands for Displacement Adjusted Virtual Nesting, and EMSR-b. In its first step, this approach requires the solution of a linear program (LP) [101], which we explain below.

Let $\mathsf{E}[Y_j]$ denote the expected demand and V_j the revenue respectively for the jth product in the network. Let $\mathcal{D}_j$ denote the set of legs used by product j and $\mathcal{A}_l$ denote the set of products that need leg l. Then solve the following LP:

$$\text{Maximize } \sum_{j=1}^{n} V_j z_j, \text{ such that } 0 \le z_j \le \mathsf{E}[Y_j], \text{ for } j = 1, 2 \ldots, n,$$

$$\text{and } \sum_{j \in \mathcal{A}_l} z_j \le C^l, \quad l = 1, 2, \ldots, L, \tag{12.1}$$

where C^l denotes the capacity of the plane on the lth leg, n denotes the number of products, and L denotes the total number of legs. The value of z_j, the decision variable, could be used as the booking limit for product j. However, this value can be significantly improved upon by combining the results of this LP with EMSR-b, as discussed next.

The *D*isplacement *A*djusted *RE*venue (DARE) for the jth product that uses leg l, i.e., $DARE_j^l$, is computed as follows. For $j = 1, 2, \ldots, n$ and every $l \in \mathcal{D}_j$,

$$DARE_j^l = V_j - \sum_{i \ne l; i \in \mathcal{D}_j} B_i,$$

and B_i denotes the dual (shadow) prices associated with the ith capacity constraint (see (12.1)) in the LP. Then $DARE_j^l$ can be treated as the *virtual* revenue of the product j on leg l, i.e., $V_j^l = DARE_j^l$.

Finally, EMSR-b is employed on each leg separately, treating the virtual revenue as the actual revenue. On every leg, products that are relevant may have to be re-ordered according to their DARE values; the higher the DARE value, the higher the class (according to our convention). The demand for each relevant product on every leg has to be determined from that of individual products. Then EMSR-b is applied on each leg in the network. If the booking limit for product j on leg l is denoted by BL_j^l, a customer requesting a given product is accepted *if and only if* the conditions with respect to *all* the booking limits are satisfied, i.e., if at time t in the booking horizon, $\phi_j(t)$ denotes the number of seats sold for product j, then product j is accepted if $\phi_j(t) < BL_j^l$ for *every* leg l used by product j. Otherwise that customer is rejected. It is thus entirely possible that a customer meets the above condition for one leg but not for some other leg. However, if the conditions for *all* the legs are not met, the customer is rejected.

Computational results. We now present some computational results. We consider a case with four cities: the hub, A, and three other cities, denoted by X, Y, and Z (see e.g., Fig. 12.1). Thus, there are 6 legs: A→X, X→A, A→Y, Y→A, A→Z, and Z→A. Table 12.1 shows the 12 itineraries possible in this setting, along with some other data, over which we will optimize. For every itinerary, the airline offers two products (i and j), one that pays a high fare and one that pays a low fare. Thus, there are 24 products in all. The higher fare product has a lower cancellation penalty.

Any customer arrives in the booking horizon of 100 days according to a non-homogeneous Poisson process with an intensity function defined as $9 + 0.03t$. The arriving customer belongs to the product i with probability $Pr(i)$ and has a cancellation probability of $CP(i)$. V_i will denote the fare for the ith product. The cancellation penalty (paid by the customer to the airline for cancellation) for the higher fare paying customer is 5 dollars while the same for the lower fare paying customer is 80 dollars. If a customer has to be bumped, the following assumption was made. Since the customer is likely to travel with the same airline although at a later time, the fare is not returned to the customer (unlike in a cancellation), but a penalty of 500 dollars is charged to the airline.

The DAVN computations are performed using the LP procedure described above. For EMSR-b calculations, the non-homogeneous

Table 12.1. Itineraries in the four-city network

Product (i,j)	Itinerary	$(Pr(i), Pr(j))$	$(CP(i), CP(j))$	(V_i, V_j)
(1,13)	A → X	(0.056,0.014)	(0.025,0.3)	(350,500)
(2,14)	X → A	(0.064,0.016)	(0.025,0.3)	(375,525)
(3,15)	A → Y	(0.048,0.012)	(0.025,0.3)	(400,550)
(4,16)	Y → A	(0.056,0.014)	(0.05,0.3)	(430,585)
(5,17)	A → Z	(0.064,0.016)	(0.05,0.3)	(450,600)
(6,18)	Z → A	(0.048,0.012)	(0.075,0.3)	(500,650)
(7,19)	X → Y via A	(0.08,0.020)	(0.125,0.3)	(600,750)
(8,20)	Y → X via A	(0.096,0.024)	(0.2,0.3)	(610,760)
(9,21)	X → Z via A	(0.08,0.020)	(0.2,0.3)	(620,770)
(10,22)	Z → X via A	(0.072,0.018)	(0.225,0.3)	(630,780)
(11,23)	Y → Z via A	(0.08,0.02)	(0.2,0.3)	(640,790)
(12,24)	Z → Y via A	(0.056,0.014)	(0.2,0.3)	(650,800)

Poisson distribution is approximated by a normal distribution (see [127, 229]) in order to compute the associated booking limits. Then, the solution from DAVN and EMSR-b combination is simulated to obtain its average reward.

The expected value of the total reward summed over all legs using SP (simultaneous perturbation) was obtained to be 229,602.85 dollars, while DAVN-EMSR-b produced a value of 203,190.75 dollars. The improvement is 12.99 %. For additional results with SP, see [127], which also solves single-leg problems. We note that the results from SP were used as a starting point for a simulated annealing to see if the solution improved any further; however, in some cases, simulated annealing did not improve these results any further. RL was used in [123, 109] for single-leg problems.

3. Preventive Maintenance

Preventive maintenance has acquired a special place in modern manufacturing management with the advent of the so-called "lean" philosophy. According to the lean philosophy, an untimely breakdown of a machine is viewed as source of *muda*—a Japanese term for waste. Indeed, an untimely breakdown of a machine can disrupt production schedules and reduce production rates. If a machine happens to be a bottleneck, it is especially important that the machine be kept in a working state almost all the time. Total Productive Maintenance, like many other *management* philosophies, relies on the age-old reliability principle, which states that if a machine is maintained in a preventive manner the up-time (availability) of the machine is raised.

Usually, as a machine ages, the probability of its failure increases. When a machine fails, a **repair** becomes necessary. On the other hand, a machine can be **maintained** before it fails. If one waits too long, one pays for expensive repairs, while maintenance too early in the machine's life can cause maintenance costs to be excessive. As such, one seeks the optimal time for maintaining the machine. See Fig. 12.2 for a graphical demonstration of this concept.

We will discuss a semi-Markov model for solving the problem based on the work in Das and Sarkar [74]. Their maintenance model assumes, as is common in the literature, that after a repair or a maintenance, the machine is as good as new. Exact expressions for the transition probabilities have been developed. The problems they solve have a relatively small state space because of the difficulties in setting up the same expressions for larger state spaces. Also, their expressions are evaluated for some specific distributions. Nevertheless, the results obtained serve as an important source for benchmarking the performance of various RL algorithms. On large problems, since optimal solutions are unavailable, one must turn however to heuristic solutions for benchmarking.

Consider a production-inventory system, i.e., a machine with a finished product buffer. See Fig. 12.3. The buffer keeps the product until the product is taken away by the customer. The customer could be the actual customer to whom the product is sold (in case the

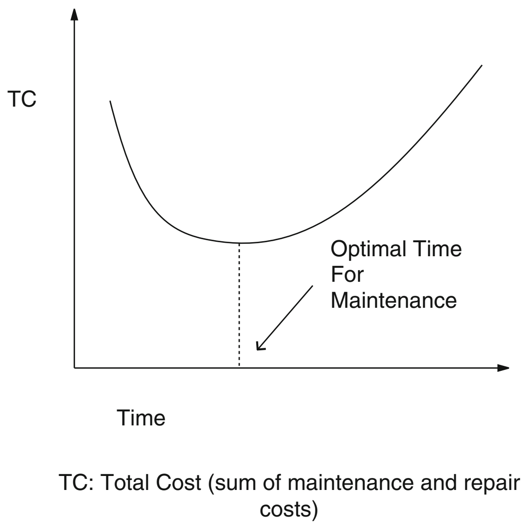

Figure 12.2. The graph shows that there is an optimal time to maintain

factory has a single machine), or else the next machine could serve as the customer. The demand, when it arrives, depletes the buffer by 1, while the machine, when it produces 1 unit, fills the buffer by 1 unit. There is a limit to how much the buffer can hold. When this limit is reached, the machine goes on vacation (stops working) and remains on vacation until the buffer level drops to a predetermined level. The time for producing a part is a random variable; the time between failures, the time for a repair, the time between demand arrivals, and the time for a maintenance are also random variables. The age of the machine can be measured by the number of units produced since last repair or maintenance. (The age can also be measured by the time since last repair or maintenance.)

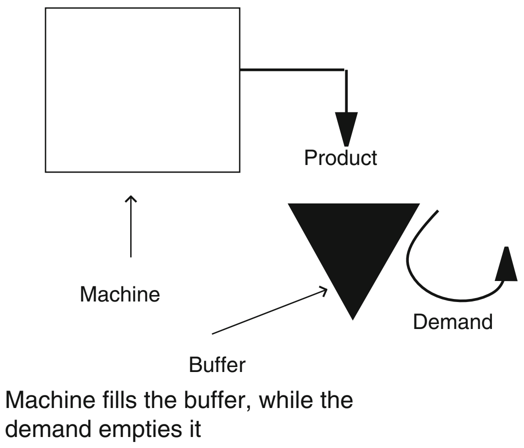

Figure 12.3. A production-inventory system

An SMDP model. The state-space is modeled as: $\hat{\phi} = \{b, c\}$, where c denotes the number of parts produced since last repair or maintenance, and b denotes the number of parts in the buffer. There are two actions that the decision maker can select from: $\{$*Produce, Maintain*$\}$. The action is to be selected at the end of a production cycle, that is, when one unit is produced.

Value-iteration-based RL approaches to solve this problem can be found in [72, 110]. RL is able to re-produce the results obtained by the optimal-seeking approach of the Das and Sarkar model [74]. An MCAT approach to solve this problem can be found in [124].

A parametric optimization model. A static model based on parametric optimization can be set up easily here. Assume that for a given value of $b1$, there is a threshold value for c. When the value of c

exceeds the threshold, maintenance must be performed. For example, if the threshold level for $b = x$ is 10, the machine must be maintained when the system state reaches $(x, 10)$, while the system must keep producing in states $(x, n), n < 10$. Thus, the problem can be solved as follows: Minimize $f(c_0, c_1, \ldots, c_k)$, where c_i denotes the threshold level for maintenance when $b = i$. Here $f(.)$ denotes the average *cost* of running the system using the threshold levels specified inside the round brackets. The optimality of the threshold policy should be established in order for the parametric optimization approach to be useful. Schouten and Vanneste [265] have established the existence of a structure for the optimal policy under certain conditions that are different than those in [74].

An Age-Replacement Heuristic. With large-scale problems, on which optimal solutions are unknown, it is important to benchmark the performance of simulation-optimization techniques in any setting. An important framework, which has been used successfully in reliability applications, goes by the name renewal theory. It provides very robust heuristics in many areas, including in maintenance theory. We now describe the age-replacement heuristic based on renewal theory.

The use of renewal reward theory for developing heuristics is often based on a simple strategy: Identify a cyclical phenomenon in the system. Determine the expected total cost and the expected total time associated with one cycle. The expected total cost divided by the expected total time, according to renewal reward theorem [251], equals the average cost per unit time in the system. Every time the machine fails or is maintained, we can assume that a cycle is completed. Then, an analytical expression for the average cost of maintaining the machine when its age is T can be derived. Thereafter, one can use non-linear optimization to find the optimal value of T that minimizes the average cost [251]. We now obtain the average cost as a function of T. The average cost, $g(T)$, can be written as

$$g(T) = \frac{\mathsf{E}[C]}{\mathsf{E}[\theta]},$$

where C is the (random) cost in one (renewal) cycle and θ is the time consumed by one (renewal) cycle. Let x denote the time for failure of the machine. $\mathsf{E}[C]$ can be written as:

$$\mathsf{E}[C] = (1 - F(T))C_m + F(T)C_r,$$

where T is the age of maintenance, $F(x)$ is the *cdf* of the random variable X, C_m is the cost of one maintenance and C_r is the cost of one repair. Now θ can be written as:

$$\mathsf{E}[\theta] = T_r F(T) + \int_0^T x f(x) dx + (T + T_m)(1 - F(T)),$$

where $f(x)$ denotes the *pdf* of the random variable X, T_m denotes the mean time for maintenance, and T_r denotes the mean time for repair. The age-replacement heuristic was used as the benchmarking technique in [72] and [124], because of its robust performance on complex systems.

Computational results. We now present some computational results obtained with R-SMART on a small-scale problem. The buffer uses an (S, s) policy, i.e., when the inventory in the buffer reaches S, it goes on vacation and remains on vacation until the inventory falls to s. The production time, the time between failures, and the time for repair all have the Erlang distribution: $Erl(n, \lambda)$, whose mean is n/λ (see Appendix). The time for maintenance has a uniform distribution: $Unif(a, b)$. The inter-arrival time for the demand has the exponential distribution: $Expo(\mu)$ whose mean equals $1/\mu$.

We show details for one case: $(S, s) = (3, 2)$. $Expo(1/10)$ for time between arrivals, $Erl(8, 0.08)$ for time between failures, $Erl(2, 0.01)$ for time for repair, $Erl(8, 0.8)$ for production time, and $Unif(5, 20)$ for the maintenance time. $C_r = \$5$, $C_m = \$2$, and profit per unit demand's sale is \$1. CF version of R-SMART with $\eta = 0.999$ produced a near-optimal solution of $\rho = \$0.033$ per unit time. The policy turns out to have a threshold nature (concept defined in parametric optimization model) with the following thresholds: $c_1 = 5$, $c_2 = 6$, and $c_3 = 7$; when buffer is empty, the action is to produce always. Both R-SMART and SMART have been tested successfully on this and numerous other cases [72, 110].

4. Production Line Buffer Optimization

The transfer line buffer optimization problem is, perhaps, one of the most widely-studied problems in serial production lines—both in the academic community and in the industry. The driving force behind this problem is the success attributed to the use of "kanbans" in the Japanese automotive industry. Kanbans are essentially buffers placed in between machines in the serial line, also called flow shop. In a flow shop, unlike a "job shop", there is little variety in the products being produced, and all the products maintain a roughly linear flow.

In other words, products move in a line from one machine to the next. Kanbans reduce the risk of over production, minimize inventory, and maximize flow in the production shop.

Consider Fig. 12.4. In between the first and the second machine, there exists a buffer (kanban) or container. When the machine preceding a buffer completes its operation, the product (essentially a batch of parts) goes into the buffer. The machine following the buffer gets it supply from this buffer, i.e., the preceding buffer in Fig. 12.4. The first machine gets its supply from raw material, which we assume is always available. In a kanban-controlled system, there is a limit on the buffer size. When the limit is reached, the previous machine is not allowed to produce any parts until there is space in the buffer. The previous machine is then said to be *blocked.* If a machine suffers from lack of material, due to an empty buffer, it is said to be *starved.* A machine is idle when it is starved or blocked. Idleness in machines may reduce the throughput rate of the line, i.e., the number of batches produced by the line in unit time.

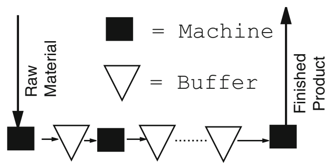

Figure 12.4. A transfer line with buffers

Ideally, starvation and blocking should be minimized. This can be done by placing large buffers to make up for any differences in the speeds of the machines. However, large buffers mean high work-in-process inventory, which should be minimized. This scenario gives rise to a parametric optimization problem—one of finding the sizes of the buffers in between the machines so that the throughput rate is maximized, while the amount of inventory is constrained to a pre-specified limit.

In the real world, production times on the machines are often random variables. Machines fail and it takes time to repair them. Failures and repairs add to the randomness in the system. This creates a challenging problem, and a semi-Markov process may be needed to capture its dynamics. [7], Altiok and Stidham set up expressions for several objective functions using Markov theory, and then optimized the systems using Hooke and Jeeves algorithm. They had to calculate the transition probabilities of the Markov chain underlying the system towards this end. A popular model is:

$$\text{Maximize } f(\vec{x}) \text{ such that } \sum_{i=1}^{k} x(i) = M \text{ or } \sum_{i=1}^{k} x(i) \leq M \qquad (12.2)$$

where $f(.)$ is the expected throughput, M, a pre-specified number, denotes a limit on the total amount of inventory in the system, $x(i)$ denotes the buffer size or number of kanbans for machine i, and k denotes the number of buffers. Another model can be mathematically described as:

$$\text{Maximize } R \cdot f(\vec{x}) - h \cdot I(\vec{x}) \qquad (12.3)$$

where R is the reward for producing a part, $I(.)$ denotes the average inventory in the system, and h is the inventory-holding cost. Both models were considered in [7].

Tezcan and Gosavi [297] simulated a production line taking into account the randomness in the production times, the failures of machines, and the repair times. Then the system was optimized for the model in Eq. (12.3) via LAST (see Chap. 5). For the model in Eq. (12.2) with the equality constraint, a closed-form approximation was used for the objective function in Shi and Men[272] (although simulation can be just as easily used) and optimization was performed via the nested partitions algorithm. In both studies, near optimal solutions were obtained with parametric optimization. For a control optimization approach for a related problem, see [223].

A kanban heuristic. A number of heuristics have appeared for this problem, since it is so widely prevalent in the industry. A simple heuristic rule that provides a lower bound on the number of kanbans is used in many industries [11]. It recommends:

$$x(i) = L(i)\lambda\Omega, \qquad (12.4)$$

where $L(i)$ is the mean lead time (production time on the machine plus waiting time in the queue in front of the machine) of a batch (which is made up of one or more parts) on machine i, λ is the rate of demand for the batch from the line, and $\Omega \geq 1$ (e.g., $\Omega = 2$) is a factor of safety.

(In many texts, the phrase "number of kanbans" is used to mean buffer size. Thus, $x(i)$ denotes the number of kanbans for machine i. Note that one kanban is generally associated with a batch. Thus if $n(i)$ denotes the number of parts associated with a kanban on machine i, then the number of parts in that buffer is actually equal to $x(i)n(i)$. We will confine ourselves to determining $x(i)$ assuming that $n(i)$ is a predetermined number.)

The rate of demand equals the rate of job arrivals at any of the machines in a kanban-controlled line, provided every machine can produce at a rate greater than the demand rate. The distributions for inter-arrival times at subsequent machines and the lead times in each of the machines can be calculated using queueing approximations; e.g., see Askin and Goldberg [11]. We now present the details for computing the lead time of a batch at any given machine. Let μ denote the mean rate of production at a machine. Let σ_s^2 denote the variance in the production time for one batch on a machine and σ_a^2 denote the variance in the inter-arrival time of a batch at a machine. Then, the *squared coefficient of variation*, which is the variance divided by the mean squared, for the inter-arrival time and the production time can be given as:

$$C_a^2 = \frac{\sigma_a^2}{(1/\lambda)^2}; \qquad C_s^2 = \frac{\sigma_s^2}{(1/\mu)^2}.$$

Using Marchal's approximation [198], one has that the *mean* waiting time in the queue in front of the machine for a batch is:

$$W_q \approx \frac{\rho^2(1+C_s^2)(C_a^2+\rho^2 C_s^2)}{2\lambda(1-\rho)(1+\rho^2 C_s^2)}, \tag{12.5}$$

where $\rho = \lambda/\mu$. The mean lead time in front of a machine then becomes: $L = W_q + \frac{1}{\mu}$, since it is the sum of the mean waiting time in front of the queue and the mean production time on the machine $(1/\mu)$. Marchal's formula in Eq. (12.5) works *approximately* for any given distribution for the inter-arrival and production time (see also [180] for another popular approximation). Note that when any of these times are deterministic, their variances should be set to zero in these formulas.

The inter-arrival time's distribution for the batch at the first machine has to be known for any of these models (simulation optimization or heuristic). However, in order to use the heuristic, one must derive the variance and the mean of the inter-arrival time at subsequent machines in the production line, since the heuristic model computes the kanban size separately for each machine. The mean inter-arrival time at every machine can be assumed to the same for every machine, i.e., $1/\lambda$, provided that the production rate of every machine is greater than the arrival rate λ. The squared coefficient of variation for a machine $(i+1)$ can be calculated using the following approximation [11]:

$$C_a^2(i+1) \approx \rho^2(i)C_s^2(i) + (1-\rho^2(i))C_a^2(i).$$

Using the above approximation, one can determine C^2 for the inter-arrival time and the production time for every machine, allowing one to use Marchal's approximation to determine the lead time ($L(i)$) at the ith machine for each i. Then, Eq. (12.4) yields the buffer size for the machine concerned.

Computational results. We now present some computational results from Tezcan and Gosavi [297]. Consider a three-machine production line with two buffers. This is a problem for which optimal solutions are available in [7]. We will use $Expo(\eta)$ to denote the exponential distribution whose mean is $1/\eta$. The time for production ($Expo(\alpha)$), time for repair ($Expo(\beta)$), and the time between failures ($Expo(\gamma)$) for each machine are all exponentially distributed (note that this is the assumption in the theoretical model of [7], but not needed for our simulation-optimization approach). Table 12.2 provides some of the input parameters for the systems studied. The model in Eq. (12.3) was used. Table 12.3 compares the results obtained from the optimal approach and LAST; z denotes the objective function's value with LAST, while z^* denotes the optimal value, and $x(i)$ denotes the LAST solution for the ith buffer, while $x^*(i)$ denotes the optimal solution. As is clear, LAST produces near-optimal solutions on the small-scale problems.

Table 12.2. Input parameters for experiments with LAST

Machine	α	β	γ
1	0.25	0.1	0.01
2	0.2	0.3	0.02
3	0.3	0.5	0.04

Table 12.3. Results with LAST

R	h	$(x^*(1), x^*(2))$	z^*	$(x(1), x(2))$	z
30	0.5	$(1, 5)$	1.92	$(1, 4)$	1.90
30	0.2	$(3, 8)$	2.52	$(3, 6)$	2.51
14	0.2	$(2, 11)$	1.62	$(2, 10)$	1.61

5. Other Case Studies

In this section, we will provide a brief overview of some other case studies involving simulation-based optimization. Some of the earlier case studies (elevator control [70], AGV routing[292], and grid world problems [288]) were from computer science. Here we provide a brief introduction to some more recent cases from communications, quality control, and operations research, and of course, computer science.

Wireless communication. The so-called *voice over packet* technology is an alternative to the older circuit switching technology used for communicating voice in networks (e.g., over telephones). The voice over packet technology uses the mechanism of "packets," which are used to transfer data. While it offers significant advantages over the older technology, it requires frequent (dynamic) adjustment of the capacity of the communicating path (or a pseudo-wire). The problem of dynamically adjusting the capacity of the path in a voice over packet network has been set up as an SMDP in Akar and Sahin [3]. They use dynamic programming for smaller problems and reinforcement learning for larger problems. They consider two different formulations of the problem. In the first formulation, there is a cost associated to changing the capacity, and in the second, there is no cost but a limit is placed on how many times the capacity can be updated in unit time. For the first formulation, they use an algorithm based on the simulation-based asynchronous value iteration algorithm of Jalali and Ferguson [153], while for the second formulation, they use an algorithm based on R-SMART [110]. Other applications include [330]

Supply chain management. The field of supply chains covers numerous computational problems related to raw material ordering, inventory control, and transportation of raw materials and finished products [309]. An inventory control problem was studied via neuro-response surface optimization in [165]. A problem studied in Pontrandolfo et al. [236] is that of selecting a vendor (supplier) dynamically depending on the current inventory levels in the manufacturing plants and retailer warehouses. An RL-based SMDP model was developed in [236] and solved using SMART. The well-known beer game in supply chains that is modeled on the phenomenon of the bull-whip effect has been studied in [60]. See [245] for a study of optimal disassembly in remanufacturing and [128, 284] for material dispatching in vendor-managed inventory systems using SP, neuro-response surfaces, and RL.

Control charts. Control charts are on-line tools used for controlling the quality of parts in a manufacturing process and for diagnosing problems in the production process. The economic design of a control chart, which seeks to minimize the average cost of the operation, requires the optimization of numerous parameters [208]. See [73] for one example that uses Markov chains to construct the closed form of the cost function. Because of the complexity of the cost function, it becomes necessary to use numerical methods of optimization. Fu and Hu [92] apply perturbation analysis in control chart design. Simulation has been used in quality control for several years [15], but it appears that the potential for optimization needed in control charts remains untapped.

Computer Games and Neuro-Science. A number of computer games have been modeled as MDPs or agent-based versions of MDPs. The first reported case study of this nature is that of backgammon [296], which almost beat the then human world champion. More recently, success with games of Othello [303], which uses a discounted algorithm in a multi-agent setting, and robotic soccer [324], which uses a model-building algorithm, has been reported. While these games are purely for entertainment, it is notoriously difficult to develop a computer program that can play well with a human, since it requires thinking like a human. Neuro-science attempts to study the functioning of the brain. Several neuro-science studies in recent times have used reinforcement learning models. Interestingly, model-building algorithms appear to have been favored in human brain studies via fMRI [331, 150].

Stochastic Games. The extension of the MDP to the multi-agent setting, where there are multiple decision-makers, and the transition probabilities and rewards depend on actions of all the decision-makers, is called a stochastic game or a competitive Markov decision process (see [86] for a comprehensive discussion of this topic). An exciting branch of RL is devoted to solving stochastic games. A special case of this stochastic game is related to the famous zero-sum game [308] for which an RL algorithm can be found in [191]. For the non-zero-sum game, two of the key algorithms are from Hu and Wellman [148] for discounted reward and Ravulapati et al. [244] for average reward. The average reward algorithm and its variants have also been used in solving the "matrix" version of the stochastic game [214]. The convergence of these algorithms is an open problem. See Collins [68] for a game-theoretic RL application in airline pricing.

Miscellaneous. A large number of other case studies of model-free simulation optimization can be found in the literature. We enumerate a few.

Parametric optimization: ground water management [310, 129], financial portfolio optimization [69], body shop design [282], communication protocol optimization in wireless sensor networks [276], social security systems [82], and health care [170].

Control optimization: irrigation control [267], re-entrant semiconductor manufacturing lines [243], autonomous helicopter control [218, 1], aircraft taxi-out time prediction [93], sensor placement [28], vehicle cruise control [197], variable speed-limit control [336], and ambulance redeployment [200].

6. Conclusions

We conclude with some final thoughts on computational aspects of both parametric and control optimization.

Parametric optimization. From our review of the literature, the genetic algorithm appears to be one of the most popular techniques for discrete problems, although there is no general agreement on which algorithm is the best. It is the oldest algorithm in this family, and it relatively easy to code, which could be one reason for its popularity. Tabu search, also, has seen a large number of useful applications. Stochastic adaptive search techniques [333] have considerable theoretical backing. Although their convergence guarantees are asymptotic (in the limit) and so they often generate sub-optimal solutions in finite time, their solutions are usually of good quality. In comparison to classical RSM, stochastic adaptive search and meta-heuristics sometimes take less computational time. In the field of continuous optimization, simultaneous perturbation appears to be a remarkable development. Compared to finite differences, it usually takes less computational time. It does get trapped in local optima and may hence require multiple starts. We must remember that parametric optimization techniques developed here do require fine-tuning of several parameters, e.g., temperature, tabu-list length, step-sizes, or some other scalars, in order to obtain the best behavior, which can increase computational time.

Control optimization. In this area, we covered RL and stochastic policy search techniques. The use of both methods on large-scale problems started in the late 1980s and continues to grow in popularity. Getting an RL algorithm to work on a real-life case study usually requires that the simulator be written in a language such as C or MATLAB so that RL-related functions and function approximation routines can be incorporated into the simulator. The reason is that in RL, unlike parametric optimization, the function is not evaluated at fixed scenarios; rather RL functions have to be incorporated into simulation trajectories. Furthermore, each problem has its own unique features. In some cases, value iteration schemes seem to fare better than policy iteration schemes, and in some cases the reverse is true. The objective function in MDPs or SMDPs, usually, cannot be chosen by the analyst because it depends on how it is measured in the real world setting for the problem at hand. Choosing the right scheme for function approximation is, however, up to the analyst, and it can often prove to be a time-consuming task. There also exist numerous ways for function approximation, e.g., neural networks, local regression, and interpolation. Each of these approaches require significant amount of trial and error. A neural network requires testing of several parameters, such as the number of hidden layers, the number of nodes, the learning rates etc.

APPENDIX

Probability Basics: The probability of an event is a measure of the likelihood of the event. Typically in statistics, the event is viewed as an outcome of a statistical experiment. E.g., you inspect a part and it turns out to be defective. Inspecting the part is an experiment, and obtaining the outcome "defective" is called an event.
Regardless of the nature of the experiment, in order to determine the probability of an event, one must make several trials of the related experiment. So if we conduct m trials of an experiment, and an event A occurs n out of m times in that particular experiment, the probability of the event A is mathematically defined as:

$$\mathsf{P}(A) = \lim_{m \to \infty} \frac{n}{m}.$$

In practice, we measure probability with a finite but a large value for m.

Random Variables: Random variables are either discrete or continuous. Let X denote the random variable in question. Discrete random variables are those for which the number of values of X belong to a set that is finite or countably infinite. We say a set is countably infinite if we can put the set into a one-to-one correspondence with a set of integers. When the set to which the values of X belong is an interval or a collection of intervals, X is called a continuous random variable. We denote the value of a random variable called X by x.

Discrete Random Variables: The probability that a discrete random variable, X, will assume a given value x is called the probability mass function (*pmf*). The *pmf* is defined as:

$$f(x) = \mathsf{Pr}(X = x).$$

The cumulative distribution function (*cdf*) for a discrete random variable is defined as:

$$F(x) = \mathsf{Pr}(X \le x).$$

Note that if x_i denotes the ith value of the random variable, then

$$F(x) = \sum_{\text{all } x_i \le x} f(x_i).$$

A. Gosavi, *Simulation-Based Optimization*, Operations Research/Computer Science Interfaces Series 55,
DOI 10.1007/978-1-4899-7491-4

The mean of expected or average value of the discrete random variable is defined as:

$$\mathsf{E}[X] = \sum_{\text{all } i} x_i f(x_i).$$

The variance is defined as: $\mathsf{Var}[X] = \sum_{\text{all } i} (x_i - \mathsf{E}[X])^2 f(x_i)$.

An alternative approach to calculating the variance is to use the following identity:

$$\mathsf{Var}[X] = \mathsf{E}[X^2] - [\mathsf{E}[X]]^2 = \sum_{\text{all } i} (x_i)^2 f(x_i) - \left[\sum_{\text{all } i} x_i f(x_i)\right]^2.$$

The positive square root of the variance is called the standard deviation of the random variable and is often denoted by σ:

$$\sigma(X) = \sqrt{\mathsf{Var}[X]}.$$

Continuous Random Variables: The probability that a continuous random variable, X, assumes values in the interval (a, b) is defined as:

$$\mathsf{Pr}(a < X < b) = \int_a^b f(x)dx,$$

where $f(x)$ denotes the so-called probability density function or *pdf* of X. Clearly, then

$$\mathsf{Pr}(X = x_i) = \int_{x_i}^{x_i} f(x)dx = 0.$$

The cumulative distribution function (*cdf*) for a continuous random variable is defined as:

$$F(x) = \mathsf{Pr}(X < x).$$

Consider a continuous random variable, X, that assumes values from a to b. We now replace $f(x)$ by $f(t)$ to denote the *pdf* of X; this replacement is perfectly legal mathematically. Then the *cdf* of X can be represented in general as:

$$F(x) = \int_a^x f(t)dt.$$

As a result, we can obtain the *pdf* from the *cdf* by differentiation as follows:

$$f(x) = \frac{dF(x)}{dx}.$$

The mean of expected or average value of the continuous random variable defined over an interval (a, b) is defined as:

$$\mathsf{E}[X] = \int_a^b x f(x)dx,$$

while the variance is defined as:

$$\mathsf{Var}[X] = \int_a^b (x - \mathsf{E}[X])^2 f(x)dx.$$

An alternative approach to calculating the variance is to use the following identity:

$$\mathsf{Var}[X] = \mathsf{E}[X^2] - [\mathsf{E}[X]]^2 = \int_a^b x^2 f(x)dx - \left(\int_a^b xf(x)dx\right)^2.$$

The positive square root of the variance is called the standard deviation of the random variable and is often denoted by σ:

$$\sigma(X) = \sqrt{\mathsf{Var}[X]}.$$

Discrete Distributions: We first discuss some well-known discrete distributions. For the following two distributions, assume that a trial results in either a success or a failure.

Bernoulli: If $X = 1$ denotes success and $X = 0$ failure and if $P(X = 1) = p$ and $P(X = 0) = 1 - p$, then X has the Bernoulli distribution.

$$\mathsf{E}[X] = p; \qquad \mathsf{Var}[X] = p(1-p).$$

Geometric: If the random variable denotes the number of trials required until the first success, where the prob. of success is p, then it is said to have geometric distribution.

$$f(x) = (1-p)^{x-1}p; \qquad \mathsf{E}[X] = 1/p; \qquad \mathsf{Var}[X] = (1-p)/p^2.$$

Poisson distribution: If a random variable has the Poisson distribution with parameter λ, it means:

$$f(x) = \frac{e^{-\lambda}\lambda^x}{x!} \text{ for } x = 0, 1, 2, \ldots; \quad \mathsf{E}[X] = \lambda; \quad \mathsf{Var}[X] = \lambda.$$

Note that the mean and variance of the Poisson distribution are equal.

Continuous Distributions: We now discuss some well-known continuous distributions.

Uniform: X has $Unif(a, b)$ implies that

$$f(x) = \frac{1}{b-a} \text{ if } a \le x \le b \text{ and } f(x) = 0 \text{ otherwise.}$$

If $x < a$, $F(x) = 0$, if $x > b$, $F(x) = 1$ and if $a \le x \le b$, $F(x) = \frac{x-a}{b-a}$. Also, $\mathsf{E}[X] = (a + b)/2$ and $\mathsf{Var}[X] = (b - a)^2/12$.

Normal distribution: Denoted by $N(\mu, \sigma^2)$. For $-\infty < x < \infty$:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

Note that μ and σ are parameters used to define the *pdf*. It turns out that:

$$\mathsf{E}[X] = \mu; \qquad \mathsf{Var}[X] = \sigma^2.$$

Erlang distribution: $Erl(n, \lambda)$, where n is a positive integer, has the following properties:

$$f(x) = \frac{\lambda^n x^{n-1} e^{-\lambda x}}{(n-1)!} \text{ for } x \geq 0; \qquad \mathsf{E}[X] = \frac{n}{\lambda}; \qquad \mathsf{Var}[X] = \frac{n}{\lambda^2}.$$

If $n = 1$, we have the exponential distribution as a special case.

Exponential distribution: $Expo(\lambda)$ has the following properties:

$$f(x) = \lambda e^{-\lambda x} \text{ for } x \geq 0; \quad \mathsf{E}[X] = \frac{1}{\lambda}; \quad \mathsf{Var}[X] = \frac{1}{\lambda^2}.$$

Note that the standard deviation and the mean of the exponential distribution are equal.

Matrix Max Norms: If $\mathbf{A}$ denotes a matrix, the max norm of the matrix is defined as:

$$||\mathbf{A}||_\infty = \max_{i,j} |A(i,j)|.$$

Note that the matrix max norm has an important property:

$$||a\mathbf{A}||_\infty = |a| ||\mathbf{A}||_\infty \text{ where } a \text{ is any real-valued scalar.}$$

The **spectral radius** of a square matrix $\mathbf{A}$ is defined as

$$\nu(\mathbf{A}) = \max_i(|\psi_i|)$$

where ψ_i denotes the ith eigenvalue of $\mathbf{A}$. It has an important relationship with the max norm.

$$\nu(\mathbf{A}) \leq ||\mathbf{A}||_\infty.$$

Spectral shift: If $eig(\mathbf{A})$ denotes *any* eigenvalue of the matrix $\mathbf{A}$, then if $\mathbf{B} = \mathbf{C} - a\mathbf{I}$, where a is a scalar, we have that $eig(B) = eig(C) - a$.

Weighted max norm: The notation $||\vec{x}||_\upsilon$ is often used to denote the **weighted max** norm of the vector $\vec{x}$ with N components and is defined as:

$$||\vec{x}||_\upsilon = \max_i \frac{|x(i)|}{\upsilon(i)},$$

where $|a|$ denotes the absolute value of $a \in \Re$ and $\vec{\upsilon} = (\upsilon(1), \upsilon(2), \ldots, \upsilon(N))$ denotes a vector of weights such that all weights are positive.

Contraction with respect to a weighted max norm: A mapping (or transformation) F is said to be a contraction mapping in $\Re^n$ with respect to a weighted max norm if there exists a λ where $0 \leq \lambda < 1$ and a vector $\vec{\upsilon}$ of n positive components such that

$$||F\vec{v} - F\vec{u}||_\upsilon \leq \lambda ||\vec{v} - \vec{u}||_\upsilon \text{ for all } \vec{v}, \vec{u} \text{ in } \Re^n.$$

Bibliography

[1] P. Abbeel, A. Coates, T. Hunter, A.Y. Ng, Autonomous autorotation of an RC helicopter, in *International Symposium on Robotics*, Seoul, 2008

[2] J. Abounadi, D. Bertsekas, V.S. Borkar, Learning algorithms for Markov decision processes with average cost. SIAM J. Control Optim. **40**(3), 681–698 (2001)

[3] N. Akar, S. Sahin, Reinforcement learning as a means of dynamic aggregate QoS provisioning, in *Architectures for Quality of Service in the Internet*, ed. by W. Burakowski, A. Bęben, B. Koch. Lecture Notes in Computer Science, vol. 2698 (Springer, Berlin/Heidelberg, 2003)

[4] J.S. Albus, *Brain, Behavior and Robotics* (Byte Books, Peterborough, 1981)

[5] M.H. Alrefaei, S. Andradóttir, A simulated annealing algorithm with constant temperature for discrete stochastic optimization. Manag. Sci. **45**(5), 748–764 (1999)

[6] M.H. Alrefaei, S. Andradóttir, A modification of the stochastic ruler method for discrete stochastic optimization. Eur. J. Oper. Res. **133**, 160–182 (2001)

[7] T. Altiok, S. Stidham, The allocation of interstage buffer capacities in production lines. IIE Trans. **15**(4), 292–299 (1984)

[8] S. Andradóttir, Simulation optimization, in *Handbook of Simulation*, ed. by J. Banks, chapter 9 (Wiley, New York, 1998)

[9] B. Ankenman, B.L. Nelson, J. Staum, Stochastic kriging for simulation metamodeling. Oper. Res. **58**(2), 371–382 (2010)

[10] H. Arsham, A. Feurvergerd, D.L. McLeish, J. Kreimer, R.Y. Rubinstein, Sensitivity analysis and the what-if problem in simulation analysis. Math. Comput. Model. **12**(2), 193–219 (1989)

[11] R. Askin, J. Goldberg, *Design and Analysis of Lean Production Systems* (Wiley, New York, 2002)

[12] A.B. Badiru, D.B. Sieger, Neural network simulation metamodel in economic analysis of risky projects. Eur. J. Oper. Res. **105**, 130–142 (1998)

[13] L. Baird, Residual algorithms: reinforcement learning with function approximation, in *Proceedings of the Twelfth International Conference on Machine Learning*, Tahoe City (Morgan Kaufmann, 1995), pp. 30–37

[14] J. Banks, J. Carson III, B. Nelson, D. Nicol, *Discrete-Event System Simulation*, 5th edn. (Prentice Hall, Upper Saddle River, 2010)

[15] N. Barish, N. Hauser, Economic design of control decisions. J. Ind. Eng. **14**, 125–134 (1963)

[16] A.G. Barto, P. Anandan, Pattern recognizing stochastic learning automata. IEEE Trans. Syst. Man Cybern. **15**, 360–375 (1985)

[17] A.G. Barto, S.J. Bradtke, S.P. Singh, Learning to act using real-time dynamic programming. Artif. Intell. **72**, 81–138 (1995)

[18] A.G. Barto, R.S. Sutton, C.W. Anderson, Neuronlike elements that can solve difficult learning control problems. IEEE Trans. Syst. Man Cybern. **13**, 835–846 (1983)

[19] R.B. Barton, Simulation optimization using metamodels, in *Proceedings of the Winter Simulation Conference*, Austin (IEEE, 2009)

[20] R.R. Barton, J.S. Ivey, Nelder-Mead simplex modifications for simulation optimization. Manag. Sci. **42**(7), 954–973 (1996)

[21] J. Baxter, P. Bartlett, Infinite-horizon policy-gradient estimation. J. Artif. Intell. **15**, 319–350 (2001)

[22] R.E. Bechhofer, T.J. Santner, D.J. Goldsman, *Design and Analysis of Experiments for Statistical Selection, Screening, and Multiple Comparisons* (Wiley, New York, 1995)

[23] R. Bellman, The theory of dynamic programming. Bull. Am. Math. Soc. **60**, 503–516 (1954)

[24] R.E. Bellman, *Dynamic Programming* (Princeton University Press, Princeton, 1957)

[25] R.E. Bellman, S.E. Dreyfus, *Applied Dynamic Programming* (Princeton University Press, Princeton, 1962)

[26] P.P. Belobaba, Application of a probabilistic decision model to airline seat inventory control. Oper. Res. **37**, 183–197 (1989)

[27] P.P. Belobaba, L.R. Weatherford, Comparing decision rules that incorporate customer diversion in perishable asset revenue management situations. Decis. Sci. **27**, 343–363 (1996)

[28] T. Ben-Zvi, J. Nickerson, Decision analysis: environmental learning automata for sensor placement. IEEE Sens. J. **11**(5), 1206–1207 (2011)

[29] D.P. Bertsekas, *Non-linear Programming* (Athena Scientific, Belmont, 1995)

[30] D.P. Bertsekas, *Dynamic Programming and Optimal Control*, 3rd edn. (Athena Scientific, Belmont, 2007)

[31] D.P. Bertsekas, Approximate policy iteration: a survey and some new methods. J. Control Theory Appl. **9**(3), 310–335 (2011)

[32] D.P. Bertsekas, J.N. Tsitsiklis, An analysis of the shortest stochastic path problems. Math. Oper. Res. **16**, 580–595 (1991)

[33] D.P. Bertsekas, J. Tsitsiklis, *Neuro-Dynamic Programming* (Athena Scientific, Belmont, 1996)

[34] D.P. Bertsekas, H. Yu, Distributed asynchronous policy iteration in dynamic programming, in *Proceedings of the 48th Allerton Conference on Communication, Control, and Computing*, Monticello (IEEE, 2010)

[35] D.P. Bertsekas, H. Yu, Q-learning and enhanced policy iteration in discounted dynamic programming, in *Proceedings of the 49th IEEE Conference on Decision and Control (CDC)*, Atlanta, 2010, pp. 1409–1416

[36] D. Bertsimas, S. de Boer, Simulation-based booking limits for airline revenue management. Oper. Res. **53**(1), 90–106 (2005)

[37] S. Bhatnagar, Adaptive multivariate three-timescale stochastic approximation algorithms for simulation based optimization. ACM Trans. Model. Comput. Simul. **15**(1), 74–107 (2005)

[38] S. Bhatnagar, M.S. Abdulla, Simulation-based optimization algorithms for finite horizon Markov decision processes. Simulation **84**(12), 577–600 (2008)

[39] S. Bhatnagar, V.S. Borkar, A two time scale stochastic approximation scheme for simulation based parametric optimization. Probab. Eng. Inf. Sci. **12**, 519–531 (1998)

[40] S. Bhatnagar, V.S. Borkar, Multiscale chaotic SPSA and smoothed functional algorithms for simulation optimization. Simulation **79**(10), 569–580 (2003)

[41] S. Bhatnagar, H.J. Kowshik, A discrete parameter stochastic approximation algorithm for simulation optimization. Simulation **81**(11), 757–772 (2005)

[42] D. Blackwell, Discrete dynamic programming. Ann. Math. Stat. **33**, 226–235 (1965)

[43] I. Bohachevsky, M. Johnson, M. Stein, Generalized simulated annealing for function approximation. Technometrics **28**, 209–217 (1986)

[44] L.B. Booker, Intelligent behaviour as an adaptation to the task environment, PhD thesis, University of Michigan, Ann Arbor, 1982

[45] V.S. Borkar, Stochastic approximation with two-time scales. Syst. Control Lett. **29**, 291–294 (1997)

[46] V.S. Borkar, Asynchronous stochastic approximation. SIAM J. Control Optim. **36**(3), 840–851 (1998)

[47] V.S. Borkar, On the number of samples required for Q-learning, in *Proceedings of the 38th Allerton Conference on*

Communication, Control and Computing, University of Illinois at Urbana-Champaign, 2000

[48] V.S. Borkar, *Stochastic Approximation: A Dynamical Systems Viewpoint* (Hindusthan Book Agency, New Delhi, 2008)

[49] V.S. Borkar, S.P. Meyn, The ODE method for convergence of stochastic approximation and reinforcement learning. SIAM J. Control Optim. **38**(2), 447–469 (2000)

[50] J.A. Boyan, A.W. Moore, Generalization in reinforcement learning: safely approximating the value function. Adv. Neural Inf. Process. Syst. **7**, 369–376 (1995)

[51] S. Bradtke, A.G. Barto, Linear least squares learning for temporal differences learning. Mach. Learn. **22**, 33–57 (1996)

[52] S.J. Bradtke, M. Duff, Reinforcement learning methods for continuous-time Markov decision problems, in *Advances in Neural Information Processing Systems 7* (MIT, Cambridge, MA, 1995)

[53] R.I. Brafman, M. Tennenholtz, R-max: a general polynomial time algorithm for near-optimal reinforcement learning. J. Mach. Learn. Res. **3**, 213–231 (2002)

[54] F. Brauer, J.A. Nohel, *The Qualitative Theory of Ordinary Differential Equations: An Introduction* (Dover, New York, 1989)

[55] S. Brooks, B. Morgan, Optimization using simulated annealing. The Statistician **44**, 241–257 (1995)

[56] L. Busoniu, R. Babuska, B. De Schutter, D. Ernst, *Reinforcement Learning and Dynamic Programming Using Function Approximators* (CRC, Boca Raton, 2010)

[57] X.R. Cao, *Stochastic Learning and Optimization: A Sensitivity-Based View* (Springer, Boston, 2007)

[58] Y. Carson, A. Maria, Simulation optimization: methods and applications, in *Proceedings of the 1997 Winter Simulation Conference*, Atlanta, 1997, pp. 118–126

[59] A. Cauchy, Méthode génserale pour la résolution des systéms d'équations simultanées. C. R. Sci. Paris **25**, 536–538 (1847)

[60] S.K. Chaharsooghi, J. Heydari, S.H. Zegordi, A reinforcement learning model for supply chain ordering management: an application to the beer game. Decis. Support Syst. **45**, 949–959 (2008)

[61] H.S. Chang, M.C. Fu, J. Hu, S. Marcus, Recursive learning automata approach to Markov decision processes. IEEE Trans. Autom. Control **52**(7), 1349–1355 (2007)

[62] H.S. Chang, M.C. Fu, J. Hu, S.I. Marcus, *Simulation-Based Algorithms for Markov Decision Processes* (Springer, London, 2007)

[63] H.S. Chang, M.C. Fu, J. Hu, S.I. Marcus, An asymptotically efficient simulation-based algorithm for finite horizon stochastic dynamic programming. IEEE Trans. Autom. Control **52**(1), 89–94 (2007)

[64] H.S. Chang, H.-G. Lee, M.C. Fu, S. Marcus, Evolutionary policy iteration for solving Markov decision processes. IEEE Trans. Autom. Control **50**(11), 1804–1808 (2005)

[65] V. Chen, D. Gunther, E. Johnson, Solving for an optimal airline yield management policy via statistical learning. J. R. Res. Soc. Ser. C **52**(1), 19–30 (2003)

[66] C.-H. Chiang, J.C.H. Chen, X. Xu, An overview of research on revenue management: current issues and future research. Int. J. Revenue Manag. **1**(1), 97–128 (2007)

[67] H. Cohn, M.J. Fielding, Simulated annealing: searching for an optimal temperature schedule. SIAM J. Optim. **9**, 779–802 (1999)

[68] A. Collins, Evaluating reinforcement learning for game theory application learning to price airline seats under competition, PhD thesis, School of Management, University of Southampton, Southampton, 2009

[69] A. Consiglio, S.A. Zenios, Designing portfolios of financial products via integrated simulation and optimization models. Oper. Res. **47**(2), 195–208 (1999)

[70] R. Crites, A. Barto, Improving elevator performance using reinforcement learning, in *Neural Information Processing Systems (NIPS)*, **8**, 1017–1023 (1996)

[71] C. Darken, J. Chang, J. Moody, Learning rate schedules for faster stochastic gradient search, in *Neural Networks for Signal Processing 2 – Proceedings of the 1992 IEEE Workshop*, ed. by D.A. White, D.A. Sofge (IEEE, Piscataway, 1992)

[72] T.K. Das, A. Gosavi, S. Mahadevan, N. Marchalleck, Solving semi-Markov decision problems using average reward reinforcement learning. Manag. Sci. **45**(4), 560–574 (1999)

[73] T.K. Das, V. Jain, A. Gosavi, Economic design of dual-sampling-interval policies for x-bar charts with and without run rules. IIE Trans. **29**, 497–506 (1997)

[74] T.K. Das, S. Sarkar, Optimal preventive maintenance in a production inventory system. IIE Trans. **31**, 537–551 (1999)

[75] S. Davies, Multi-dimensional interpolation and triangulation for reinforcement learning. Adv. Neural Inf. Process. Syst. **9**, (1996)

[76] S. de Boer, R. Freling, N. Piersma, Mathematical programming for network revenue management revisited. Eur. J. Oper. Res. **137**, 72–92 (2002)

[77] G. Deng, M.C. Ferris, Variable-number sample path optimization. Math. Program. Ser. B **117**, 81–109 (2009)

[78] L. Devroye, L. Gyorfi, G. Lugosi, *A Probabilistic Theory of Pattern Recognition* (Springer, New York, 1996)

[79] C. Diuk, L. Li, B.R. Leffler, The adaptive k-meteorologists problem and its application to structure learning and feature selection in reinforcement learning, in *Proceedings of the 26th Annual International Conference on Machine Learning*, Montreal, 2009

[80] M. Dorigo, T. Stützle, *Ant Colony Optimization* (MIT, Cambridge, MA, 2004)

[81] S.A. Douglass, *Introduction to Mathematical Analysis* (Addison-Wesley, Reading, 1996)

[82] T. Ermolieva, Simulation-based optimization of social security systems under uncertainty. Eur. J. Oper. Res. **166**, 782–793 (2005)

[83] E. Evan-Dar, Y. Mansour, Learning rates for Q-learning. J. Mach. Learn. Res. **5**, 1–25 (2003)

[84] T. Feo, M. Resende, Greedy randomized adaptive search procedures. J. Global Optim. **6**, 108–133 (1995)

[85] M.J. Fielding, Optimisation by simulated annealing, PhD thesis, Department of Mathematics, The University of Melbourne, 1999

[86] J. Filar, K. Vrieze, *Competitive Markov Decision Processes* (Springer, New York, 1997)

[87] P.A. Fishwick, Neural network models in simulation: a comparison with traditional modeling approaches, in *Proceedings of the 1989 Winter Simulation Conference*, Washington, DC, 1989, pp. 702–710

[88] B.L. Fox, G.W. Heine, Probabilistic search with overrides. Ann. Appl. Probab. **5**, 1087–1094 (1995)

[89] M.C. Fu, Optimization via simulation: theory vs. practice. INFORMS J. Comput. **14**(3), 192–215 (2002)

[90] M.C. Fu, Optimization via simulation: a review. Ann. Oper. Res. **53**, 199–247 (1994)

[91] M.C. Fu, J.Q. Hu, *Conditional Monte Carlo-Gradient Estimation and Optimization Applications* (Kluwer Academic, Boston, 1997)

[92] M.C. Fu, J. Hu, Efficient design and sensitivity analysis of control charts using Monte Carlo simulation. Manag. Sci. **45**(3), 395–413 (1999)

[93] R. Ganesan, P. Balakrishnan, L. Sherry, Improving quality of prediction in highly dynamic environments using approximate dynamic programming. Qual. Reliab. Eng. Int. **26**, 717–732 (2010)

[94] F. Garcia, S. Ndiaye, A learning rate analysis of reinforcement learning algorithms in finite horizon, in *Proceedings of the 15th International Conference on Machine Learning*, Madison (Morgan Kauffmann, 1998)

[95] E.D. Gaughan, *Introduction to Analysis*, 4th edn. (Brooks/Cole, Belmont, 1993)

[96] S.B. Gelfand, S.K. Mitter, Simulated annealing with noisy or imprecise energy measurements. J. Optim. Theory Appl. **62**(1), 49–62 (1989)

[97] M. Gendreau, J.-Y. Potvin, *Handbook of Metaheuristics* (Springer, New York, 2010)

[98] L. Gerencser, S.D. Hill, Z. Vago, Optimization over discrete sets by SPSA, in *Proceedings of the Winter Simulation Conference*, Phoenix, 1999, pp. 466–470

[99] S. German, D. German, Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. IEEE Trans. Pattern Anal. Mach. Intell. (PAMI) **6**, 721–741 (1984)

[100] F. Glover, Tabu search: a tutorial. Interfaces **20**(4), 74–94 (1990)

[101] F. Glover, R. Glover, J. Lorenzo, C. McMillan, The passenger mix problem in scheduled airlines. Interfaces **12**, 73–79 (1982)

[102] F. Glover, S. Hanafi, Tabu search and finite convergence. Discret. Appl. Math. **119**, 3–36 (2002)

[103] F. Glover, J.P. Kelly, M. Laguna, New advances and applications of combining simulation and optimization, in *Proceedings of the 1996 Winter Simulation Conference*, Coronado, 1996, pp. 144–152

[104] F. Glover, M. Laguna, *Tabu Search* (Kluwer Academic, Norwell, 1998)

[105] F. Glover, M. Laguna, R. Marti, Fundamentals of scatter search and path relinking. Control Cybern. **29**(3), 653–684 (2000)

[106] D. Goldsman, B.L. Nelson, Comparing systems via simulation, in *Handbook of Simulation*, ed. by J. Banks, chapter 8 (Wiley, New York, 1998)

[107] W.B. Gong, Y.C. Ho, W. Zhai, Stochastic comparison algorithm for discrete optimization with estimation, in *Proceedings of the 31st Conference on Decision Control*, Tucson, 1992, pp. 795–800

[108] A. Gosavi, The effect of noise on artificial intelligence and meta-heuristic techniques, in *Conference Proceedings of*

Artificial Neural Networks in Engineering (ANNIE), St. Louis, vol. 12. (American Society of Mechanical Engineering Press, 2002), pp. 981–988

[109] A. Gosavi, A reinforcement learning algorithm based on policy iteration for average reward: empirical results with yield management and convergence analysis. Mach. Learn. **55**, 5–29 (2004)

[110] A. Gosavi, Reinforcement learning for long-run average cost. Eur. J. Oper. Res. **155**, 654–674 (2004)

[111] A. Gosavi, Boundedness of iterates in Q-learning. Syst. Control Lett. **55**, 347–349 (2006)

[112] A. Gosavi, A risk-sensitive approach to total productive maintenance. Automatica **42**, 1321–1330 (2006)

[113] A. Gosavi, On step-sizes, stochastic paths, and survival probabilities in reinforcement learning, in *Proceedings of the 2008 Winter Simulation Conference*, Miami (IEEE, 2008)

[114] A. Gosavi, Reinforcement learning: a tutorial survey and recent advances. INFORMS J. Comput. **21**(2), 178–192 (2009)

[115] A. Gosavi, Reinforcement learning for model building and variance-penalized control, in *Proceedings of the 2009 Winter Simulation Conference*, Austin (IEEE, 2009)

[116] A. Gosavi, Finite horizon Markov control with one-step variance penalties, in *Conference Proceedings of the Allerton Conference*, University of Illinois at Urbana-Champaign, 2010

[117] A. Gosavi, Model building for robust reinforcement learning, in *Conference Proceedings of Artificial Neural Networks in Engineering (ANNIE)*, St. Louis (ASME Press, 2010), pp. 65–72

[118] A. Gosavi, Approximate policy iteration for semi-Markov control revisited, in *Procedia Computer Science, Complex Adaptive Systems*, Chicago (Elsevier, 2011), pp. 249–255

[119] A. Gosavi, Target-sensitive control of Markov and semi-Markov processes. Int. J. Control Autom. Syst. **9**(5), 1–11 (2011)

[120] A. Gosavi, Approximate policy iteration for Markov control revisited, in *Procedia Computer Science, Complex Adaptive Systems*, Chicago (Elsevier, 2012)

[121] A. Gosavi, Codes for neural networks, DP, and RL in the C language for this book (2014), http://web.mst.edu/~gosavia/bookcodes.html

[122] A. Gosavi, Using simulation for solving Markov decision processes, in *Handbook of Simulation Optimization* (forthcoming), ed. by M. Fu (Springer, New York, 2014)

[123] A. Gosavi, N. Bandla, T.K. Das, A reinforcement learning approach to a single leg airline revenue management problem with multiple fare classes and overbooking. IIE Trans. **34**(9), 729–742 (2002)

[124] A. Gosavi, T.K. Das, S. Sarkar, A simulation-based learning automata framework for solving semi-Markov decision problems. IIE Trans. **36**, 557–567 (2004)

[125] A. Gosavi, S. Murray, J. Hu, S. Ghosh, Model-building adaptive critics for semi-Markov control. J. Artif. Intell. Soft Comput. Res. **2**(1) (2012)

[126] A. Gosavi, S.L. Murray, V.M. Tirumalasetty, S. Shewade, A budget-sensitive approach to scheduling maintenance in a total productive maintenance (TPM) program. Eng. Manag. J. **23**(3), 46–56 (2011)

[127] A. Gosavi, E. Ozkaya, A. Kahraman, Simulation optimization for revenue management of airlines with cancellations and overbooking. OR Spectr. **29**, 21–38 (2007)

[128] A. Gosavi, G. Subramaniam, Simulation-based optimisation for material dispatching in vendor-managed inventory systems. Int. J. Simul. Process Model. **3**(4), 238–245 (2007)

[129] J. Grundmann, N. Schutze, G.H. Schmitz, S. Al-Shaqsi. Towards an integrated arid zone water management using simulation-based optimisation. Environ. Earth Sci. **65**, 1381–1394 (2012)

[130] B. Hajek, Cooling schedules for optimal annealing. Math. Oper. Res. **13**, 311–329 (1988)

[131] T. Hastie, R. Tibshirani, J. Friedman, *The Elements of Statistical Learning* (Springer, New York, 2001)

[132] S. Haykin, *Neural Networks: A Comprehensive Foundation* (McMillan, New York, 1994)

[133] S.S. Heragu, *Facilities Design*, 3rd edn. (CRC, Boca Raton, 2008)

[134] F.S. Hillier, G.J. Lieberman, *Introduction to Operations Research*, 7th edn. (McGraw Hill, New York, 2001)

[135] G.E. Hinton, Distributed representations. Technical report, CMU-CS-84-157, Carnegie Mellon University, Pittsburgh, 1984

[136] M. Hirsch, Convergent activation dynamics in continuous time networks. Neural Netw. **2**, 331–349 (1989)

[137] Y.C. Ho, X.R. Cao, *Perturbation Analysis of Discrete Event Dynamic Systems* (Springer, Boston, 1991)

[138] Y.C. Ho, R. Sreenivas, P. Vakili, Ordinal optimization of discrete event dynamic systems. J. DEDS **2**(2), 61–88 (1992)

[139] Y. Hochberg, A.C. Tamhane, *Multiple Comparison Procedures* (Wiley, New York, 1987)

[140] J.H. Holland, *Adaptation in Natural and Artificial Systems* (University of Michigan Press, Ann Arbor, 1975)

[141] J.H. Holland, Escaping brittleness: the possibility of general-purpose learning algorithms applied to rule-based systems, in *Machine Learning: An Artificial Intelligence Approach*, ed. by R.S. Michalski, J.G. Carbonell, T.M. Mitchell (Morgan Kaufmann, San Mateo, 1986), pp. 593–623

[142] L.J. Hong, B.L. Nelson, Discrete optimization via simulation using COMPASS. Oper. Res. **54**(1), 115–129 (2006)

[143] R. Hooke, T.A. Jeeves, Direct search of numerical and statistical problems. ACM **8**, 212–229 (1966)

[144] R. Howard, *Dynamic Programming and Markov Processes* (MIT, Cambridge, MA, 1960)

[145] J. Hu, H.S. Chang, Approximate stochastic annealing for online control of infinite horizon Markov decision processes. Automatica **48**(9), 2182–2188 (2012)

[146] J. Hu, M.C. Fu, S.I. Marcus, A model reference adaptive search method for global optimization. Oper. Res. **55**, 549–568 (2007)

[147] J. Hu, M.C. Fu, S.I. Marcus, A model reference adaptive search method for stochastic global optimization. Commun. Inf. Syst. **8**, 245–276 (2008)

[148] J. Hu, M.P. Wellman, Nash Q-Learning for general-sum stochastic games. J. Mach. Learn. Res. **4**, 1039–1069 (2003)

[149] D. Huang, T.T. Allen, W.I. Notz, N. Zeng, Global optimization of stochastic black-box systems via sequential kriging meta-models. J. Global Optim. **34**, 441–466 (2006)

[150] S. Ishii, W. Yoshida, J. Yoshimoto, Control of exploitation-exploration meta-parameter in reinforcement learning. Neural Netw. **15**, 665–687 (2002)

[151] T. Jaakkola, M. Jordan, S. Singh, On the convergence of stochastic iterative dynamic programming algorithms. Neural Comput. **6**(6), 1185–1201 (1994)

[152] S.H. Jacobson, L.W. Schruben, A harmonic analysis approach to simulation sensitivity analysis. IIE Trans. **31**(3), 231–243 (1999)

[153] A. Jalali, M. Ferguson, Computationally efficient adaptive control algorithms for Markov chains, in *Proceedings of the 29th IEEE Conference on Decision and Control*, Honolulu, 1989, pp. 1283–1288

[154] J.-S.R. Jang, C.-T. Sun, E. Mizutani, *Neuro-Fuzzy and Soft Computing* (Prentice Hall, Upper Saddle River, 1997)

[155] M.V. Johns Jr., R.G. Miller Jr., Average renewal loss rates. Ann. Math. Stat. **34**(2), 396–401 (1963)

[156] S.A. Johnson, J.R. Stedinger, C.A. Shoemaker, Y. Li, J.A. Tejada-Guibert, Numerical solution of continuous state dynamic programs using linear and spline interpolation. Oper. Res. **41**(3), 484–500 (1993)

[157] L.P. Kaelbling, M.L. Littman, A.W. Moore, Reinforcement learning: a survey. J. Artif. Intell. Res. **4**, 237–285 (1996)

[158] P. Kanerva, *Sparse Distributed Memory* (MIT, Cambridge, MA, 1988)

[159] I. Karaesmen, G. van Ryzin, Overbooking with substitutable inventory classes. Oper. Res. **52**(1), 83–104 (2004)

[160] M. Kearns, S. Singh, Near-optimal reinforcement learning in polynomial time. Mach. Learn. **49**(2), 209–232 (2002)

[161] W.D. Kelton, R.B. Barton, Experimental design for simulation, in *Proceedings of the Winter Simulation Conference*, New Orleans (IEEE, 2003)

[162] J.G. Kemeny, J.L. Snell, *Finite Markov Chains* (van Nostrand-Reinhold, New York, 1960)

[163] J. Kennedy, R.C. Eberhart, Y. Shi, *Swarm Intelligence* (Morgan Kaufmann, San Francisco, 2001)

[164] J. Kiefer, J. Wolfowitz, Stochastic estimation of the maximum of a regression function. Ann. Math. Stat. **23**, 462–466 (1952)

[165] R.A. Kilmer, A.E. Smith, L.J. Shuman, Computing confidence intervals for stochastic simulation using neural network metamodels. Comput. Ind. Eng. **36**(2), 391–407 (1998)

[166] S.-H. Kim, B.L. Nelson, A fully sequential procedure for indifference-zone selection in simulation. ACM Trans. Model. Comput. Simul. **11**, 251–273 (2001)

[167] H. Kimura, K. Miyazaki, S. Kobayashi, Reinforcement learning in POMDPs with function approximation, in *Proceedings of the 14th International Conference on Machine Learning*, Nashville, 1997, pp. 152–160

[168] H. Kimura, M. Yamamura, S. Kobayashi, Reinforcement learning by stochastic hill climbing on discounted reward, in *Proceedings of the 12th International Conference on Machine Learning*, Tahoe City, 1995, pp. 295–303

[169] S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, Optimization by simulated annealing. Science **220**, 671–680 (1983)

[170] K.K. Klassen, R. Yoogalingam, Improving performance in outpatient appointment services with a simulation optimization approach. Prod. Oper. Manag. **18**(4), 447–458 (2009)

[171] J.P.C. Kleijnen, Sensitivity analysis and optimization in simulation: design of experiments and case studies, in

Proceedings of the 1995 Winter Simulation Conference, Arlington, 1995, pp. 133–140

[172] J.P.C. Kleijnen, *Design and Analysis of Simulation Experiments* (Springer, New York, 2007)

[173] J.P.C. Kleijnen, R.G. Sargent, A methodology for fitting and validating metamodels in simulation. Eur. J. Oper. Res. **120**, 14–29 (2000)

[174] J.P.C. Kleijnen, W.C.M. van Beers, Robustness of kriging when interpolating in random simulation with heterogeneous variances: some experiments. Eur. J. Oper. Res. **165**, 826–834 (2005)

[175] R. Klein, Network capacity control using self-adjusting prices. OR Spectr. **29**, 39–60 (2007)

[176] A.H. Klopf, Brain function and adaptive systems—a heterostatic theory. Technical report AFCRL-72-0164, 1972

[177] D.E. Knuth, *The Art of Computer Programming.* Volume 1: Seminumerical Algorithms, 3rd edn. (Addison-Wesley, Reading, 1998)

[178] V.R. Konda, V.S. Borkar, Actor-critic type learning algorithms for Markov decision processes. SIAM J. Control Optim. **38**(1), 94–123 (1999)

[179] R. Koppejan, S. Whiteson, Neuroevolutionary reinforcement learning for generalized helicopter control, in *GECCO: Proceedings of the Genetic and Evolutionary Computation Conference*, Montreal, 2009, pp. 145–152

[180] W. Kramer, M. Lagenbach-Belz. Approximate formulae for the delay in the queueing system GI/G/1, in *Proceedings of 8th International Teletraffic Congress*, Melbourne, 1976, pp. 1–8

[181] E. Kreyszig, *Advanced Engineering Mathematics* (Wiley, New York, 1998)

[182] B. Kristinsdottir, Z.B. Zabinsky, G. Wood, Discrete backtracking adaptive search for global optimization, in *Stochastic and Global Optimization*, ed. by G. Dzemyda, V. Saltenis, A. Zilinskas (Kluwer Academic, Dordrecht/London, 2002), pp. 147–174

[183] K. Kulkarni, A. Gosavi, S. Murray, K. Grantham, Semi-Markov adaptive critic heuristics with application to airline revenue management. J. Control Theory Appl. **9**(3), 421–430 (2011)

[184] H.J. Kushner, D.S. Clark, *Stochastic Approximation Methods for Constrained and Unconstrained Systems* (Springer, New York, 1978)

[185] H.J. Kushner, G. Yin, *Stochastic Approximation and Recursive Algorithms and Applications*, 2nd edn. (Springer, New York, 2003)

[186] J.C. Lagarias, J.A. Reeds, M.H. Wright, P.E. Wright, Convergence properties of the Nelder–Mead simplex method in low dimensions. SIAM J. Optim. **9**(1), 112–147 (1998)

[187] M. Lagoudakis, R. Parr, Least-squares policy iteration. J. Mach. Learn. Res. **4**, 1107–1149 (2003)

[188] A.M. Law, W.D. Kelton, *Simulation Modeling and Analysis* (McGraw Hill, New York, 1999)

[189] P. L'Ecuyer, Good parameters for combined multiple recursive random number generators. Oper. Res. **47**, 159–164 (1999)

[190] K. Littlewood, Forecasting and control of passenger bookings, in *Proceedings of the 12th AGIFORS (Airline Group of the International Federation of Operational Research Societies Symposium)*, Nathanya, 1972, pp. 95–117

[191] M.L. Littman, Markov games as a framework for multi-agent reinforcement learning, in *The Eleventh International Conference on Machine Learning*, New Brunswick (Morgan Kaufmann, 1994), pp. 157–163

[192] L. Ljung, Analysis of recursive stochastic algorithms. IEEE Trans. Autom. Control **22**, 551–575 (1977)

[193] M. Lundy, A. Mees, Convergence of the annealing algorithm. Math. Program. **34**, 111–124 (1986)

[194] G.R. Madey, J. Wienroth, V. Shah, Integration of neurocomputing and system simulation for modeling continuous improvement systems in manufacturing. J. Intell. Manuf. **3**, 193–204 (1992)

[195] S. Mahadevan, Average reward reinforcement learning: foundations, algorithms, and empirical results. Mach. Learn. **22**(1), 159–195 (1996)

[196] S. Mahadevan, Learning representation and control in Markov decision processes: new frontiers, in *Foundations and Trends in Machine Learning*, vol. I(4) (Now Publishers, Boston, 2009), pp. 403–565

[197] A. Malikopoulos, P. Papalambros, D. Assanis, A real-time computational learning model for sequential decision-making problems under uncertainty. J. Dyn. Syst. Measur. Control (ASME) **131**, 041010–1–041010–8 (2009)

[198] W.G. Marchal, An approximate formula for waiting time in single server queues. AIIE Trans. **8**(4), 473–474 (1976)

[199] S. Marsland, *Machine Learning: An Algorithmic Perspective* (CRC, Boca Raton, 2009)

[200] M. Maxwell, M. Restrepo, H. Topaloglu, S.G. Henderson, Approximate dynamic programming for ambulance redeployment. INFORMS J. Comput. **22**, 266–281 (2010)

[201] J.I. McGill, G.J. van Ryzin. Revenue management: research overview and prospects. Transp. Sci. **33**(2), 233–256 (1999)

[202] G. Meghabghab, G. Nasr, Iterative RBF neural networks as metamodels of stochastic simulations, in *2nd International Conference on Intelligent Processing and Manufacturing of Materials*, Honolulu, vol. 2, 1999, pp. 729–734

[203] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, E. Teller, Equation of state calculations by fast computing machines. J. Chem. Phys **21**, 1087–1092 (1953)

[204] J. Michels, A. Saxena, A.Y. Ng, High speed obstacle avoidance using monocular vision and reinforcement learning, in *Proceedings of the 22nd International Conference on Machine Learning*, Bonn, 2005

[205] T.M. Mitchell, *Machine Learning* (McGraw Hill, Boston, 1997)

[206] D. Mitra, F. Romeo, A. Sangiovanni-Vincentelli, Convergence of finite time behavior of simulated annealing. Adv. Appl. Probab. **18**, 747–771 (1986)

[207] O. Molvalioglu, Z.B. Zabinsky, W. Kohn, The interacting particle algorithm with dynamic heating and cooling. J. Global Optim. **43**, 329–356 (2019)

[208] D.C. Montgomery, *Introduction to Statistical Quality Control*, 4th edn. (Wiley, New York, 2001)

[209] D.G. Montgomery, G.C. Runger, N.A. Hubele, *Engineering Statistics*, 2nd edn. (Wiley, New York, 2001)

[210] J.E. Moody, C. Darken, Fast learning in networks of locally-tuned processing units. Neural Comput. **1**, 281–294 (1989)

[211] A.W. Moore, C.G. Atkeson, Prioritized sweeping: reinforcement learning with less data and less real time. Mach. Learn. **13**, 103–130 (1993)

[212] M. Müller-Bungart, *Revenue Management with Flexible Products* (Springer, Berlin, 2010)

[213] R.H. Myers, D.C. Montgomery, *Response Surface Methodology: Process and Product Optimization Using Designed Experiments* (Wiley, New York, 1995)

[214] V. Nanduri, T.K. Das, A reinforcement learning approach for obtaining the Nash equilibrium of multiplayer matrix games. IIE Trans. **41**, 158–167 (2009)

[215] K.S. Narendra, M.A.L. Thathachar, *Learning Automata: An Introduction* (Prentice Hall, Englewood Cliffs, 1989)

[216] J.A. Nelder, R. Mead, A simplex method for function minimization. Comput. J. **7**, 308–313 (1965)

[217] A. Ng, M. Jordan, PEGASUS: a policy search method for large MDPs and POMDPs, in *Proceedings of 16th Conference on Uncertainty in Artificial Intelligence*, Stanford, 2000

[218] A.Y. Ng, H.J. Kim, M.I. Jordan, S. Sastry, Autonomous helicopter flight via reinforcement learning. Adv. Neural Inf. Process. Syst. **17** (2004). MIT

[219] D. Ormoneit, S. Sen, Kernel-based reinforcement learning. Mach. Learn. **49**(2–3), 161–178 (2002)

[220] M.L. Padgett, T.A. Roppel, Neural networks and simulation: modeling for applications. Simulation **58**, 295–305 (1992)

[221] S.K. Park, K.W. Miller, Random number generators: good ones are hard to find. Commun. ACM **31**(10), 1192–1201 (1988)

[222] R. Pasupathy, B.W. Schmeiser, Retrospective-approximation algorithms for multidimensional stochastic root-finding problems. ACM TOMACS **19**(2), 5:1–5:36 (2009)

[223] C.D. Paternina, T.K. Das, Intelligent dynamic control policies for serial production lines. IIE Trans. **33**(1), 65–77 (2001)

[224] J. Peng, R.J. Williams, Incremental multi-step Q-learning. Mach. Learn. **22**, 226–232 (1996). Morgan Kaufmann

[225] L. Peshkin, Reinforcement learning by policy search, PhD thesis, MIT, Cambridge, MA, 2001

[226] G.C. Pflug, *Optimization of Stochastic Models: The Interface Between Simulation and Optimization* (Kluwer Academic, Boston, 1996)

[227] D.T. Pham, D. Karaboga, *Intelligent Optimisation Techniques: Genetic Algorithms, Tabu Search, Simulated Annealing, Neural Networks* (Springer, New York, 1998)

[228] C.R. Philbrick, P.K. Kitanidis, Improved dynamic programming methods for optimal control of lumped-parameter stochastic systems. Oper. Res. **49**(3), 398–412 (2001)

[229] R.L. Phillips, *Pricing and Revenue Optimization* (Stanford University Press, Stanford, 2005)

[230] H. Pierreval, Training a neural network by simulation for dispatching problems, in *Proceedings of the Third Rensselaer International Conference on Computer Integrated Manufacturing*, Troy, 1992, pp. 332–336

[231] H. Pierreval, R.C. Huntsinger, An investigation on neural network capabilities as simulation metamodels, in *Proceedings of the 1992 Summer Computer Simulation Conference*, Reno, 1992, pp. 413–417

[232] E.L. Plambeck, B.R. Fu, S.M. Robinson, R. Suri, Sample path optimization of convex stochastic performance functions. Math. Program. **75**, 137–176 (1996)

[233] B.T. Poljak, Y.Z. Tsypkin, Pseudogradient adaptation and training algorithms. Autom. Remote Control **12**, 83–94 (1973)

[234] M.A. Pollatschek, *Programming Discrete Simulations* (Research and Development Books, Lawrence, KS, 1995)

[235] B.T. Polyak, *Introduction to Optimization* (Optimization Software, New York, 1987)

[236] P. Pontrandolfo, A. Gosavi, O.G. Okogbaa, T.K. Das, Global supply chain management: a reinforcement learning approach. Int. J. Prod. Res. **40**(6), 1299–1317 (2002)

[237] W. Powell, *Approximate Dynamic Programming: Solving the Curses of Dimensionality* (Wiley-Interscience, Hoboken, 2007)

[238] W. Powell, I. Ryzhov, *Optimal Learning* (Wiley, New York, 2012)

[239] W.H. Press, S.A. Tuekolsky, W.T. Vetterling, B.P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing* (Cambridge University Press, Cambridge, 1992)

[240] A.A. Prudius, S. Andradóttir, Balanced explorative and exploitative search with estimation for simulation optimization. INFORMS J. Comput. **21**, 193–208 (2009)

[241] M. Purohit, A. Gosavi, Stochastic policy search for semi-Markov variance-penalized control, in *Proceedings of the 2011 Winter Simulation Conference*, Phoenix (IEEE, 2011)

[242] M.L. Puterman, *Markov Decision Processes* (Wiley Interscience, New York, 1994)

[243] J.A. Ramirez-Hernandez, E. Fernandez, A case study in scheduling re-entrant manufacturing lines: optimal and simulation-based approaches, in *Proceedings of 44th IEEE Conference on Decision and Control*, Seville (IEEE, 2005), pp. 2158–2163

[244] K.K. Ravulapati, J. Rao, T.K. Das, A reinforcement learning appraoch to stochastic business games. IIE Trans. **36**, 373–385 (2004)

[245] S.A. Reveliotis, Uncertainty management in optimal disassembly planning through learning-based strategies. IIE Trans. **39**(6), 645–658 (2007)

[246] Y. Rinott, On two-stage selection procedures and related probability-inequalities. Commun. Stat. Theory Methods **A7**, 799–811 (1978)

[247] H. Robbins, S. Monro, A stochastic approximation method. Ann. Math. Stat. **22**, 400–407 (1951)

[248] H.E. Romeijn, R.L. Smith, Simulated annealing and adaptive search in Global optimization. Probab. Eng. Inf. Sci **8**, 571–590 (1994)

[249] H.E. Romeijn, R.L. Smith, Simulated annealing for constrained global optimization. J. Global Optim. **5**, 101–126 (1994)

[250] S.M. Ross, *Introduction to Stochastic Dynamic Programming* (Academic, New York, 1983)

[251] S.M. Ross, *Introduction to Probability Models* (Academic, San Diego, 1997)

[252] S.M. Ross, *A First Course in Probability*, 6th edn. (Academic, San Diego, 2002)

[253] R.Y. Rubinstein, A. Shapiro, *Sensitivity Analysis and Stochastic Optimization by the Score Function Method* (Wiley, New York, 1983)

[254] W. Rudin, *Real Analysis* (McGraw Hill, New York, 1964)

[255] G. Rudolph, Convergence of canonical genetic algorithms. IEEE Trans. Neural Netw. **5**, 96–101 (1994)

[256] G. Rudolph, Convergence of evolutionary algorithms in general search spaces, in *Proceedings of the Third IEEE Conference on Evolutionary Computation* (IEEE, Piscataway, 1994), pp. 50–54

[257] D.E. Rumelhart, G.E. Hinton, R.J. Williams, Learning internal representations by error propagation, in *Parallel Distributed Processing: Explorations in the Micro-structure of Cognition*, ed. by D.E. Rumelhart, J.L. McClelland (MIT, Cambridge, MA, 1986)

[258] G.A. Rummery, M. Niranjan, On-line Q-learning using connectionist systems. Technical report CUED/F-INFENG/TR 166. Engineering Department, Cambridge University, 1994

[259] I. Sabuncuoglu, S. Touhami, Simulation metamodeling with neural networks: an experimental investigation. Int. J. Prod. Res. **40**, 2483–2505 (2002)

[260] A.L. Samuel, Some studies in machine learning using the game of checkers, in *Computers and Thought*, ed. by E.A. Feigenbaum, J. Feldman (McGraw-Hill, New York 1959)

[261] S. Sanchez, Better than a petaflop: the power of efficient experimental design, in *Proceedings of the 2008 Winter Simulation Conference*, Miami (IEEE, 2008), pp. 73–84

[262] S. Sarkar, S. Chavali, Modeling parameter space behavior of vision systems using Bayesian networks. Comput. Vis. Image Underst. **79**, 185–223 (2000)

[263] B. Schmeiser, Approximation to the inverse cumulative normal function for use on hand calculators. Appl. Stat. **28**, 175–176 (1979)

[264] B. Schmeiser, Batch-size effects in the analysis of simulation output. Oper. Res. **30**, 556–568 (1982)

[265] F.A.V. Schouten, S.G. Vanneste, Maintenance optimization with buffer capacity. Eur. J. Oper. Res. **82**, 323–338 (1992)

[266] L. Schrage, A more portable random number generator. Assoc. Comput. Mach. Trans. Math. Softw. **5**, 132–138 (1979)

[267] N. Schutze, G.H.Schmitz. Neuro-dynamic programming as a new framework for decision support for deficit irrigation systems, in *International Congress on Modelling and Simulation*, Christchurch, 2007, pp. 2271–2277

[268] A. Schwartz, A reinforcement learning method for maximizing undiscounted rewards, in *Proceeding of the Tenth Annual Conference on Machine Learning*, Amherst, 1993, pp. 298–305

[269] E. Seneta, *Non-negative Matrices and Markov Chains* (Springer, New York, 1981)

[270] L.I. Sennott, *Stochastic Dynamic Programming and the Control of Queueing Systems.* (Wiley, New York, 1999)

[271] L.S. Shapley, Stochastic games. Proc. Natl. Acad. Sci. **39**, 1095–1100 (1953)

[272] L. Shi, S. Men, Optimal buffer allocation in production lines. IIE Trans. **35**, 1–10 (2003)

[273] L. Shi, S. Olafsson, Nested partitions method for Global optimization. Oper. Res. **48**(3), 390–407 (2000)

[274] L. Shi, S. Olafsson, Nested partitions method for stochastic optimization. Methodol. Comput. Appl. Probab. **2**(3), 271–291 (2000)

[275] L. Shi, S. Olafsson, *Nested Partitions Method, Theory and Applications* (Springer, New York, 2008)

[276] G. Simon, P. Volgyesi, M. Maroti, A. Ledeczi, Simulation-based optimization of communication protocols for large-scale wireless networks, in *Aerospace Conference Proceedings*, Big Sky (IEEE, 2003), pp. 1339–1346

[277] S. Singh, T. Jaakkola, M. Littman, C. Szepesvari, Convergence results for single-step on-policy reinforcement-learning algorithms. Mach. Learn. **39**, 287–308 (2000)

[278] S. Singh, V. Tadic, A. Doucet, A policy-gradient method for semi-Markov decision processes with application to call admission control. Eur. J. Oper. Res. **178**(3), 808–818 (2007)

[279] R.L. Smith, Efficient Monte Carlo procedures for generating points uniformly distributed over bounded regions. Oper. Res. **32**, 1296–1308 (1984)

[280] J.C. Spall, Multivariate stochastic approximation using a simultaneous perturbation gradient approximation. IEEE Trans. Autom. Control **37**, 332–341 (1992)

[281] J.C. Spall, *Introduction to Stochastic Search and Optimization: Estimation, Simulation, and Control* (Wiley, New York, 2003)

[282] S. Spieckermann, K. Gutenschwager, H. Heinzel, S. Voss, Simulation-based optimization in the automotive industry—a case study on body shop design. Simulation **75**(5), 276–286 (2000)

[283] A.L. Strehl, M.L. Littman, A theoretical analysis of model-based interval estimation, in *Proceedings of the 22th International Conference on Machine Learning*, Bonn, 2005, pp. 856–863

[284] Z. Sui, A. Gosavi, L. Lin, A reinforcement learning approach for inventory replenishment in vendor-managed inventory systems with consignment inventory. Eng. Manag. J. **22**(4), 44–53 (2010)

[285] R.S. Sutton, Learning to predict by the method of temporal differences. Mach. Learn. **3**, 9–44 (1988)

[286] R.S. Sutton, Integrated architectures for learning, planning, and reacting based on approximating dynamic programming, in *Proceedings of the 7th International Workshop on Machine Learning*, Austin (Morgan Kaufmann, San Mateo, 1990), pp. 216–224

[287] R.S. Sutton, Generalization in reinforcement learning: successful examples using sparse coarse coding, in *Advances in Neural Information Processing Systems 8* (MIT, Cambridge, MA 1996)

[288] R. Sutton, A.G. Barto, *Reinforcement Learning: An Introduction* (MIT, Cambridge, MA, 1998)

[289] C. Szepesvári, Algorithms for reinforcement learning, in *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 10 (Morgan Claypool Publishers, San Rafael, 2010), pp. 1–103

[290] C. Szepesvari, M. Littman, A unified analysis of value-function-based reinforcement-learning algorithms. Neural Comput. **11**(8), 2017–2059 (1998)

[291] H. Szu, R. Hartley, Fast simulated annealing. Phys. Lett. A **122**, 157–162 (1987)

[292] P. Tadepalli, D. Ok, Model-based average reward reinforcement learning algorithms. Artif. Intell. **100**, 177–224 (1998)

[293] H.A. Taha, *Operations Research: An Introduction* (Prentice Hall, Upper Saddle River, 1997)

[294] K. Talluri, G. van Ryzin, *The Theory and Practice of Revenue Management* (Kluwer Academic, Boston, 2004)

[295] H. Taylor, S. Karlin, *An Introduction to Stochastic Modeling* (Academic, New York, 1984)

[296] G. Tesauro, Practical issues in temporal difference learning. Mach. Learn. **8**(3) (1992)

[297] T. Tezcan, A. Gosavi, Optimal buffer allocation in production lines using an automata search, in *Proceedings of the 2001 Institute of Industrial Engineering Research Conference*, Dallas, 2001

[298] M.A.L. Thathachar, P.S. Sastry, Learning optimal discriminant functions through a cooperative game of automata. IEEE Trans. Syst. Man Cybern. **17**, 73–85 (1987)

[299] S. Thrun, W. Burgard, D. Fox, *Probabilistic Robotics* (MIT, Cambridge, MA, 2005)

[300] J. Tsitsiklis, Asynchronous stochastic approximation and Q-learning. Mach. Learn. **16**, 185–202 (1994)

[301] A. Turgeon, Optimal operation of multi-reservoir power systems with stochastic inflows. Water Resour. Res. **16**(2), 275–283 (1980)

[302] J.M. Twomey, A.E. Smith, Bias and variance of validation models for function approximation neural networks under conditions of sparse data. IEEE Trans. Syst. Man Cybern. **28**(3), 417–430 (1998)

[303] N.J. van Eck, M. van Wezel, Application of reinforcement learning to the game of Othello. Comput. Oper. Res. **35**, 1999–2017 (2008)

[304] P. van Laarhoven, E. Aarts, *Simulated Annealing: Theory and Applications* (Kluwer Academic, Dordrecht, 1987)

[305] J.A.E.E. van Nunen, A set of successive approximation methods for discounted Markovian decision problems. Z. Oper. Res. **20**, 203–208 (1976)

[306] G. van Ryzin, G. Vulcano, Simulation-based optimization of virtual nesting controls of network revenue management. Oper. Res. **56**(4), 865–880 (2008)

[307] H. van Seijen, S. Whiteson, H. van Hasselt, M. Wiering, Exploiting best-match equations for efficient reinforcement learning. J. Mach. Learn. Res. **12**, 2045–2094 (2011)

[308] J. von Neumann, O. Morgenstern, *The Theory of Games and Economic Behavior* (Princeton University Press, Princeton, 1944)

[309] S. Voss, D.L. Woodruff, *Introduction to Computational Optimization Models for Production Planning in a Supply Chain*, 2nd edn. (Springer, New York, 2006)

[310] B.J. Wagner, Recent advances in simulation-optimization groundwater management modeling. Rev. Geophys. **33**(S1), 1021–1028 (1995)

[311] I.-J. Wang, J.C. Spall, Stochastic optimization with inequality constraints using simultaneous perturbations and penalty functions, in *Proceedings of the 42nd IEEE Conference on Decision and Control*, Maui, 2003

[312] C.J. Watkins, Learning from delayed rewards, PhD thesis, Kings College, Cambridge, May 1989

[313] P.J. Werbös, Beyond regression: new tools for prediction and analysis of behavioral sciences, PhD thesis, Harvard University, Cambridge, MA, May 1974

[314] P.J. Werbös, Building and understanding adaptive systems: a statistical/numerical approach to factory automation and brain research. IEEE Trans. Syst. Man Cybern. **17**, 7–20 (1987)

[315] P.J. Werbös, Consistency of HDP applied to a simple reinforcement learning problem. Neural Netw. **3**, 179–189 (1990)

[316] P.J. Werbös, Approximate dynamic programming for real-time control and neural modeling, in *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches* (Van Nostrand, New York, 1992), pp. 493–525

[317] P.J. Werbös, A menu of designs for reinforcement learning over time, in *Neural Networks for Control* (MIT, Cambridge, MA, 1990), pp. 67–95

[318] P.J. Werbös, Approximate dynamic programming for real-time control and neural modeling, in *Handbook of Intelligent Control*, ed. by D.A. White, D.A. Sofge (Van Nostrand Reinhold, New York, 1992)

[319] R.M. Wheeler, K.S. Narenda, Decentralized learning in finite Markov chains. IEEE Trans. Autom. Control **31**(6), 373–376 (1986)

[320] D.J. White, Dynamic programming, Markov chains, and the method of successive approximations. J. Math. Anal. Appl. **6**, 373–376 (1963)

[321] S. Whiteson, *Adaptive Representations for Reinforcement Learning.* Volume 291 of Studies in Computational Intelligence (Springer, Berlin, 2010)

[322] S. Whiteson, P. Stone, Evolutionary function approximation for reinforcement learning. J. Mach. Learn. Res. **7**, 877–917 (2006)

[323] B. Widrow, M.E. Hoff, Adaptive Switching Circuits, in *Institute of Radio Engineers, Western Electronic Show and Convention, Convention Record*, part 4, 1960, pp. 96–104, (New York)

[324] M.A. Wiering, R.P. Salustowicz, J. Schmidhuber, Model-based reinforcement learning for evolving soccer strategies, in *Computational Intelligence in Games* (Springer, Heidelberg, 2001)

[325] R.R. Wilcox, A table for Rinott's selection procedure. J. Qual. Technol. **16**, 97–100 (1984)

[326] R.J. Williams, On the use of backpropagation in associative reinforcement learning, in *Proceedings of the Second International Conference on Neural Networks*, vol. I, San Diego, CA (IEEE, New York, 1988)

[327] R. Williams, Simple statistical gradient following algorithms for connectionist reinforcement learning. Mach. Learn. **8**, 229–256 (1992)

[328] I.H. Witten, An adaptive optimal controller for discrete time Markov environments. Inf. Control **34**, 286–295 (1977)

[329] D. Yan, H. Mukai, Stochastic discrete optimization. SIAM J. Control Optim. **30**, 594–612 (1992)

[330] W.-L. Yeow, C.-K. Tham, W.-C. Wong, Energy efficient multiple target tracking in wireless sensor networks. IEEE Trans. Veh. Technol. **56**(2), 918–928 (2007)

[331] W. Yoshida, S. Ishii, Model-based reinforcement learning: a computational model and an fMRI study. Neurocomputing **63**, 253–269 (2005)

[332] H. Yu, D.P. Bertsekas, On boundedness of Q-learning iterates for stochastic shortest path problems. Math. Oper. Res. **38**, 209–227 (2013)

[333] Z.B. Zabinsky, *Stochastic Adaptive Search for Global Optimization* (Springer, New York, 2003)

[334] Z. Zabinsky, Random search algorithms for simulation optimization, in *Handbook of Simulation Optimization* (forthcoming), ed. by M. Fu (Springer, New York, 2014)

[335] Z.B. Zabinsky, D. Bulger, C. Khompatraporn, Stopping and restarting strategy for stochastic sequential search in global optimization. J. Global Optim. **46**, 273–286 (2010)

[336] F. Zhu, S. Ukkusuri, Accounting for dynamic speed limit control in a stochastic traffic environment: a reinforcement learning approach. Transp. Res. Part C **41**, 30–47 (2014)

Index