Reinforcement Learning and Optimal Control

by

Dimitri P. Bertsekas

Massachusetts Institute of Technology

DRAFT TEXTBOOK

This is a draft of a textbook that is scheduled to be finalized in 2019, and to be published by Athena Scientific. It represents "work in progress," and it will be periodically updated. It more than likely contains errors (hopefully not serious ones). Furthermore, its references to the literature are incomplete. Your comments and suggestions to the author at dimitrib@mit.edu are welcome. The date of last revision is given below.

February 13, 2019

WWW site for book information and orders

http://www.athenasc.com

Athena Scientific, Belmont, Massachusetts

Athena Scientific
Post Office Box 805
Nashua, NH 03060
U.S.A.

Email: info@athenasc.com
WWW: http://www.athenasc.com

Publisher's Cataloging-in-Publication Data

Bertsekas, Dimitri P.
Reinforcement Learning and Optimal Control
Includes Bibliography and Index
1. Mathematical Optimization. 2. Dynamic Programming. I. Title.
QA402.5 .B465 2019 519.703 00-91281

ISBN-10: 1-886529-39-6, ISBN-13: 978-1-886529-39-7

ABOUT THE AUTHOR

Dimitri Bertsekas studied Mechanical and Electrical Engineering at the National Technical University of Athens, Greece, and obtained his Ph.D. in system science from the Massachusetts Institute of Technology. He has held faculty positions with the Engineering-Economic Systems Department, Stanford University, and the Electrical Engineering Department of the University of Illinois, Urbana. Since 1979 he has been teaching at the Electrical Engineering and Computer Science Department of the Massachusetts Institute of Technology (M.I.T.), where he is currently the McAfee Professor of Engineering.

His teaching and research have spanned several fields, including deterministic optimization, dynamic programming and stochastic control, large-scale and distributed computation, and data communication networks. He has authored or coauthored numerous research papers and seventeen books, several of which are currently used as textbooks in MIT classes, including "Dynamic Programming and Optimal Control," "Data Networks," "Introduction to Probability," and "Nonlinear Programming."

Professor Bertsekas was awarded the INFORMS 1997 Prize for Research Excellence in the Interface Between Operations Research and Computer Science for his book "Neuro-Dynamic Programming" (co-authored with John Tsitsiklis), the 2001 AACC John R. Ragazzini Education Award, the 2009 INFORMS Expository Writing Award, the 2014 AACC Richard Bellman Heritage Award, the 2014 INFORMS Khachiyan Prize for Life-Time Accomplishments in Optimization, the 2015 MOS/SIAM George B. Dantzig Prize, and the 2018 INFORMS John von Neumann Theory Prize. In 2001, he was elected to the United States National Academy of Engineering for "pioneering contributions to fundamental research, practice and education of optimization/control theory, and especially its application to data communication networks."

ATHENA SCIENTIFIC

OPTIMIZATION AND COMPUTATION SERIES

1. Abstract Dynamic Programming, 2nd Edition, by Dimitri P. Bertsekas, 2018, ISBN 978-1-886529-46-5, 360 pages
2. Dynamic Programming and Optimal Control, Two-Volume Set, by Dimitri P. Bertsekas, 2017, ISBN 1-886529-08-6, 1270 pages
3. Nonlinear Programming, 3rd Edition, by Dimitri P. Bertsekas, 2016, ISBN 1-886529-05-1, 880 pages
4. Convex Optimization Algorithms, by Dimitri P. Bertsekas, 2015, ISBN 978-1-886529-28-1, 576 pages
5. Convex Optimization Theory, by Dimitri P. Bertsekas, 2009, ISBN 978-1-886529-31-1, 256 pages
6. Introduction to Probability, 2nd Edition, by Dimitri P. Bertsekas and John N. Tsitsiklis, 2008, ISBN 978-1-886529-23-6, 544 pages
7. Convex Analysis and Optimization, by Dimitri P. Bertsekas, Angelia Nedić, and Asuman E. Ozdaglar, 2003, ISBN 1-886529-45-0, 560 pages
8. Network Optimization: Continuous and Discrete Models, by Dimitri P. Bertsekas, 1998, ISBN 1-886529-02-7, 608 pages
9. Network Flows and Monotropic Optimization, by R. Tyrrell Rockafellar, 1998, ISBN 1-886529-06-X, 634 pages
10. Introduction to Linear Optimization, by Dimitris Bertsimas and John N. Tsitsiklis, 1997, ISBN 1-886529-19-1, 608 pages
11. Parallel and Distributed Computation: Numerical Methods, by Dimitri P. Bertsekas and John N. Tsitsiklis, 1997, ISBN 1-886529-01-9, 718 pages
12. Neuro-Dynamic Programming, by Dimitri P. Bertsekas and John N. Tsitsiklis, 1996, ISBN 1-886529-10-8, 512 pages
13. Constrained Optimization and Lagrange Multiplier Methods, by Dimitri P. Bertsekas, 1996, ISBN 1-886529-04-3, 410 pages
14. Stochastic Optimal Control: The Discrete-Time Case, by Dimitri P. Bertsekas and Steven E. Shreve, 1996, ISBN 1-886529-03-5, 330 pages

Contents

1. Exact Dynamic Programming

1.1. Deterministic Dynamic Programming . . . . p. 2
 1.1.1. Deterministic Problems . . . . p. 2
 1.1.2. The Dynamic Programming Algorithm . . . . p. 7
 1.1.3. Approximation in Value Space . . . . p. 12
1.2. Stochastic Dynamic Programming . . . . p. 14
1.3. Examples, Variations, and Simplifications . . . . p. 17
 1.3.1. Deterministic Shortest Path Problems . . . . p. 19
 1.3.2. Discrete Deterministic Optimization . . . . p. 21
 1.3.3. Problems with a Terminal State . . . . p. 24
 1.3.4. Forecasts . . . . p. 26
 1.3.5. Problems with Uncontrollable State Components . . . p. 28
 1.3.6. Partial State Information and Belief States . . . . . . p. 33
 1.3.7. Linear Quadratic Optimal Control . . . . p. 37
1.4. Reinforcement Learning and Optimal Control - Some . . . . . .
 Terminology . . . . p. 40
1.5. Notes and Sources . . . . p. 42

2. Approximation in Value Space

2.1. General Issues of Approximation in Value Space . . . . . . p. 5
 2.1.1. Methods for Computing Approximations in . . . .
  Value Space . . . . p. 6
 2.1.2. Off-Line and On-Line Methods . . . . p. 7
 2.1.3. Model-Based Simplification of the Lookahead . . . . . .
  Minimization . . . . p. 8
 2.1.4. Model-Free Q-Factor Approximation in Value Space . . p. 9
 2.1.5. Approximation in Policy Space on Top of . . . .
  Approximation in Value Space . . . . p. 12
 2.1.6. When is Approximation in Value Space Effective? . . . p. 13
2.2. Multistep Lookahead . . . . p. 14
 2.2.1. Multistep Lookahead and Rolling Horizon . . . . . . p. 16

2.2.2. Multistep Lookahead and Deterministic Problems . . . p. 17
2.3. Problem Approximation . . . . p. 19
2.3.1. Enforced Decomposition . . . . p. 19
2.3.2. Probabilistic Approximation - Certainty Equivalent Control . . . . p. 26
2.4. Rollout . . . . p. 32
2.4.1. On-Line Rollout for Deterministic Finite-State Problems . . . . p. 33
2.4.2. Stochastic Rollout and Monte Carlo Tree Search . . . p. 42
2.5. On-Line Rollout for Deterministic Infinite-Spaces Problems - Optimization Heuristics . . . . p. 53
2.5.1. Model Predictive Control . . . . p. 53
2.5.2. Target Tubes and the Constrained Controllability Condition . . . . p. 60
2.5.3. Variants of Model Predictive Control . . . . p. 63
2.6. Notes and Sources . . . . p. 66

3. Parametric Approximation

3.1. Approximation Architectures . . . . p. 2
3.1.1. Linear and Nonlinear Feature-Based Architectures . . . p. 2
3.1.2. Training of Linear and Nonlinear Architectures . . . . p. 9
3.1.3. Incremental Gradient and Newton Methods . . . . . . p. 10
3.2. Neural Networks . . . . p. 23
3.2.1. Training of Neural Networks . . . . p. 27
3.2.2. Multilayer and Deep Neural Networks . . . . p. 30
3.3. Sequential Dynamic Programming Approximation . . . . . . p. 34
3.4. Q-factor Parametric Approximation . . . . p. 36
3.5. Notes and Sources . . . . p. 39

4. Infinite Horizon Reinforcement Learning

4.1. An Overview of Infinite Horizon Problems . . . . p. 3
4.2. Stochastic Shortest Path Problems . . . . p. 6
4.3. Discounted Problems . . . . p. 16
4.4. Exact and Approximate Value Iteration . . . . p. 21
4.5. Policy Iteration . . . . p. 25
4.5.1. Exact Policy Iteration . . . . p. 26
4.5.2. Optimistic and Multistep Lookahead Policy Iteration . p. 30
4.5.3. Policy Iteration for Q-factors . . . . p. 32
4.6. Approximation in Value Space - Performance Bounds . . . . p. 34
4.6.1. Limited Lookahead Performance Bounds . . . . p. 36
4.6.2. Rollout . . . . p. 39
4.6.3. Approximate Policy Iteration . . . . p. 43
4.7. Simulation-Based Policy Iteration with Parametric Approximation . . . . p. 46

4.7.1. Self-Learning and Actor-Critic Systems . . . . p. 46
4.7.2. A Model-Based Variant . . . . p. 47
4.7.3. A Model-Free Variant . . . . p. 50
4.7.4. Implementation Issues of Parametric Policy Iteration . p. 52
4.8. Q-Learning . . . . p. 55
4.9. Additional Methods - Temporal Differences . . . . p. 58
4.10. Exact and Approximate Linear Programming . . . . p. 69
4.11. Approximation in Policy Space . . . . p. 71
4.11.1. Training by Cost Optimization - Policy Gradient and . . Random Search Methods . . . . p. 73
4.11.2. Expert Supervised Training . . . . p. 80
4.12. Notes and Sources . . . . p. 81
4.13. Appendix: Mathematical Analysis . . . . p. 84
4.13.1. Proofs for Stochastic Shortest Path Problems . . . . p. 85
4.13.2. Proofs for Discounted Problems . . . . p. 90
4.13.3. Convergence of Exact and Optimistic . . . . Policy Iteration . . . . p. 91
4.13.4. Performance Bounds for One-Step Lookahead, Rollout, . . and Approximate Policy Iteration . . . . p. 93

5. Aggregation

5.1. Aggregation Frameworks . . . . p.
5.2. Classical and Biased Forms of the Aggregate Problem . . . . . p.
5.3. Bellman's Equation for the Aggregate Problem . . . . p.
5.4. Algorithms for the Aggregate Problem . . . . p.
5.5. Some Examples . . . . p.
5.6. Spatiotemporal Aggregation for Deterministic Problems . . . . p.
5.7. Notes and Sources . . . . p.

References . . . . p.

Index . . . . p.

Preface

In this book we consider large and challenging multistage decision problems, which can be solved in principle by dynamic programming (DP for short), but their exact solution is computationally intractable. We discuss solution methods that rely on approximations to produce suboptimal policies with adequate performance. These methods are collectively referred to as *reinforcement learning*, and also by alternative names such as *approximate dynamic programming*, and *neuro-dynamic programming*.

Our subject has benefited greatly from the interplay of ideas from optimal control and from artificial intelligence. One of the aims of the book is to explore the common boundary between these two fields and to form a bridge that is accessible by workers with background in either field.

Our primary focus will be on *approximation in value space*. Here, the control at each state is obtained by optimization of the cost over a limited horizon, plus an approximation of the optimal future cost, starting from the end of this horizon. The latter cost, which we generally denote by $\tilde{J}$, is a function of the state where we may be at the end of the horizon. It may be computed by a variety of methods, possibly involving simulation and/or some given or separately derived heuristic/suboptimal policy. The use of simulation often allows for implementations that do not require a mathematical model, a major idea that has allowed the use of DP beyond its classical boundaries.

We focus selectively on four types of methods for obtaining $\tilde{J}$:

(a) *Problem approximation*: Here $\tilde{J}$ is the optimal cost function of a related simpler problem, which is solved by exact DP. Certainty equivalent control and enforced decomposition schemes are discussed in some detail.

(b) *Rollout and model predictive control*: Here $\tilde{J}$ is the cost function of some known heuristic policy. The needed cost values to implement a rollout policy are often calculated by simulation. While this method applies to stochastic problems, the reliance on simulation favors de-

terministic problems, including challenging combinatorial problems for which heuristics may be readily implemented. Rollout may also be combined with adaptive simulation and Monte Carlo tree search, which have proved very effective in the context of games such as backgammon, chess, Go, and others.

Model predictive control was originally developed for continuous-space optimal control problems that involve some goal state, e.g., the origin in a classical control context. It can be viewed as a specialized rollout method that is based on a suboptimal optimization for reaching a goal state.

(c) *Parametric cost approximation*: Here $\tilde{J}$ is chosen from within a parametric class of functions, including neural networks, with the parameters "optimized" or "trained" by using state-cost sample pairs and some type of incremental least squares/regression algorithm. Approximate policy iteration and its variants are covered in some detail, including several actor-critic schemes. These involve policy evaluation with temporal difference-based training methods, and policy improvement that may rely on approximation in policy space.

(d) *Aggregation*: Here the cost function $\tilde{J}$ is the optimal cost function of some approximation to the original problem, called aggregate problem, which has fewer states. The aggregate problem can be formulated in a variety of ways, and may be solved by using exact DP techniques. Its optimal cost function is then used as $\tilde{J}$ in a limited horizon optimization scheme. Aggregation may also be used to provide local improvements to parametric approximation schemes that involve neural networks or linear feature-based architectures.

We have adopted a gradual expository approach, which proceeds along four directions:

(1) *From exact DP to approximate DP*: We first discuss exact DP algorithms, explain why they may be difficult to implement, and then use them as the basis for approximations.

(2) *From finite horizon to infinite horizon problems*: We first discuss finite horizon exact and approximate DP methodologies, which are intuitive and mathematically simple in Chapters 1-3. We then progress to infinite horizon problems in Chapters 4 and 5.

(3) *From deterministic to stochastic models*: We often discuss separately deterministic and stochastic problems. The reason is that deterministic problems are simpler and offer special advantages for some of our methods.

(4) *From model-based to model-free approaches*: Reinforcement learning methods offer a major potential benefit over classical DP approaches,

which were practiced exclusively up to the early 90s: they can be implemented by using a simulator/computer model rather than a mathematical model. In our presentation, we first discuss model-based methods, and then we identify those methods that can be appropriately modified to work with a simulator.

After the first chapter, each new class of methods is introduced as a more sophisticated or generalized version of a simpler method introduced earlier. Moreover, we illustrate some of the methods by means of examples, which should be helpful in providing insight into their use, but may also be skipped selectively and without loss of continuity. Detailed solutions to some of the simpler examples are given, and may illustrate some of the implementation details.

The mathematical style of this book is somewhat different from the one of the author's DP books [Ber12], [Ber17a], [Ber18a], and the 1996 neuro-dynamic programming (NDP) research monograph, written jointly with John Tsitsiklis [BeT96]. While we provide a rigorous, albeit short, mathematical account of the theory of finite and infinite horizon DP, and some fundamental approximation methods, we rely more on intuitive explanations and less on proof-based insights. Moreover, our mathematical requirements are quite modest: calculus, elementary probability, and a minimal use of matrix-vector algebra.

Several of the methods that we present are often successful in practice, but have less than solid performance properties. This is a reflection of the state of the art in the field: there are no methods that are guaranteed to work for all or even most problems, but there are enough methods to try on a given problem with a reasonable chance of success in the end. For this process to work, however, it is important to have proper intuition into the inner workings of each type of method, as well as an understanding of its analytical and computational properties. To quote a statement from the preface of the NDP monograph [BeT96]: "It is primarily through an understanding of the mathematical structure of the NDP methodology that we will be able to identify promising or solid algorithms from the bewildering array of speculative proposals and claims that can be found in the literature."

Another statement from a recent NY Times article [Str18], in connection with DeepMind's remarkable AlphaZero chess program, is also worth quoting: "What is frustrating about machine learning, however, is that the algorithms can't articulate what they're thinking. We don't know why they work, so we don't know if they can be trusted. AlphaZero gives every appearance of having discovered some important principles about chess, but it can't share that understanding with us. Not yet, at least. As human beings, we want more than answers. We want insight. This is going to be a source of tension in our interactions with computers from now on." To this we may add that human insight can only develop within some struc-

ture of human thought, and it appears that mathematical reasoning with algorithmic models is the most suitable structure for this purpose.

Dimitri P. Bertsekas

January 2019

Reinforcement Learning and Optimal Control

by

Dimitri P. Bertsekas

Massachusetts Institute of Technology

Chapter 1
Exact Dynamic Programming

DRAFT

This is Chapter 1 of the draft textbook "Reinforcement Learning and Optimal Control." The chapter represents "work in progress," and it will be periodically updated. It more than likely contains errors (hopefully not serious ones). Furthermore, its references to the literature are incomplete. Your comments and suggestions to the author at dimitrib@mit.edu are welcome. The date of last revision is given below.

The date of last revision is given below. (A "revision" is any version of the chapter that involves the addition or the deletion of at least one paragraph or mathematically significant equation.)

January 19, 2019

1

Exact Dynamic Programming

Contents

1.1. Deterministic Dynamic Programming . . . . p. 2
1.1.1. Deterministic Problems . . . . p. 2
1.1.2. The Dynamic Programming Algorithm . . . . . . p. 7
1.1.3. Approximation in Value Space . . . . p. 12
1.2. Stochastic Dynamic Programming . . . . p. 14
1.3. Examples, Variations, and Simplifications . . . . p. 17
1.3.1. Deterministic Shortest Path Problems . . . . . . p. 19
1.3.2. Discrete Deterministic Optimization . . . . p. 21
1.3.3. Problems with a Terminal State . . . . p. 24
1.3.4. Forecasts . . . . p. 26
1.3.5. Problems with Uncontrollable State Components . p. 28
1.3.6. Partial State Information and Belief States . . . . p. 33
1.3.7. Linear Quadratic Optimal Control . . . . p. 37
1.4. Reinforcement Learning and Optimal Control - Some . . . .
Terminology . . . . p. 40
1.5. Notes and Sources . . . . p. 42

In this chapter, we provide some background on exact dynamic programming (DP for short), with a view towards the suboptimal solution methods that are the main subject of this book. These methods are known by several essentially equivalent names: *reinforcement learning*, *approximate dynamic programming*, and *neuro-dynamic programming*. In this book, we will use primarily the most popular name: reinforcement learning (RL for short).

We first consider finite horizon problems, which involve a finite sequence of successive decisions, and are thus conceptually and analytically simpler. We defer the discussion of the more intricate infinite horizon problems to Chapter 4 and later chapters. We also discuss separately deterministic and stochastic problems (Sections 1.1 and 1.2, respectively). The reason is that deterministic problems are simpler and lend themselves better as an entry point to the optimal control methodology. Moreover, they have some favorable characteristics, which allow the application of a broader variety of methods. For example, simulation-based methods are greatly simplified and sometimes better understood in the context of deterministic optimal control.

Finally, in Section 1.3 we provide various examples of DP formulations, illustrating some of the concepts of Sections 1.1 and 1.2. The reader with substantial background in DP may wish to just scan Section 1.3 and skip to the next chapter, where we start the development of the approximate DP methodology.

1.1 DETERMINISTIC DYNAMIC PROGRAMMING

All DP problems involve a discrete-time dynamic system that generates a sequence of states under the influence of control. In finite horizon problems the system evolves over a finite number N of time steps (also called stages). The state and control at time k are denoted by x_k and u_k, respectively. In deterministic systems, x_{k+1} is generated nonrandomly, i.e., it is determined solely by x_k and u_k.

1.1.1 Deterministic Problems

A deterministic DP problem involves a discrete-time dynamic system of the form

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

where

k is the time index,

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

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

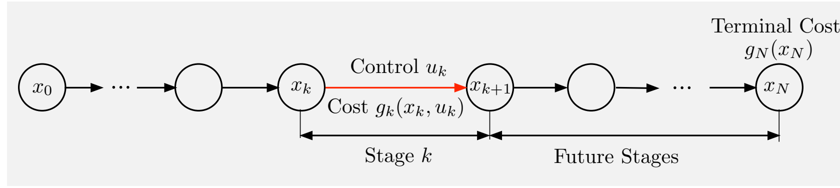

Figure 1.1.1 Illustration of a deterministic N-stage optimal control problem. Starting from state x_k, the next state under control u_k is generated nonrandomly, according to

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

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

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

N is the horizon or number of times control is applied,

The set of all possible x_k is called the *state space* at time k. It can be any set and can depend on k; *this generality is one of the great strengths of the DP methodology*. Similarly, the set of all possible u_k is called the *control space* at time k. Again it can be any set and can depend on k.

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

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

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

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

as a function of x_0. Figure 1.1.1 illustrates the main elements of the problem.

We will next illustrate deterministic problems with some examples.

† We use throughout "min" (in place of "inf") to indicate minimal value over a feasible set of controls, even when we are not sure that the minimum is attained by some feasible control.

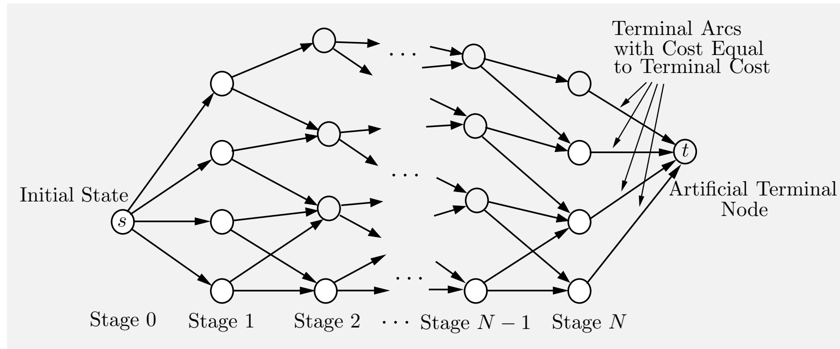

Figure 1.1.2 Transition graph for a deterministic finite-state system. Nodes correspond to states x_k. Arcs correspond to state-control pairs (x_k, u_k). An arc (x_k, u_k) has start and end nodes x_k and $x_{k+1} = f_k(x_k, u_k)$, respectively. We view the cost $g_k(x_k, u_k)$ of the transition as the length of this arc. The problem is equivalent to finding a shortest path from initial node s to terminal node t.

Discrete Optimal Control Problems

There are many situations where the state and control are naturally discrete and take a finite number of values. Such problems are often conveniently specified in terms of an acyclic graph specifying for each state x_k the possible transitions to next states x_{k+1}. The nodes of the graph correspond to states x_k and the arcs of the graph correspond to state-control pairs (x_k, u_k). Each arc with start node x_k corresponds to a choice of a single control $u_k \in U_k(x_k)$ and has as end node the next state $f_k(x_k, u_k)$. The cost of an arc (x_k, u_k) is defined as $g_k(x_k, u_k)$; see Fig. 1.1.2. To handle the final stage, an artificial terminal node t is added. Each state x_N at stage N is connected to the terminal node t with an arc having cost $g_N(x_N)$.

Note that control sequences correspond to paths originating at the initial state (node s at stage 0) and terminating at one of the nodes corresponding to the final stage N. If we view the cost of an arc as its length, we see that *a deterministic finite-state finite-horizon problem is equivalent to finding a minimum-length (or shortest) path from the initial node s of the graph to the terminal node t.* Here, by a path we mean a sequence of arcs such that given two successive arcs in the sequence the end node of the first arc is the same as the start node of the second. By the length of a path we mean the sum of the lengths of its arcs.†

† It turns out also that any shortest path problem (with a possibly nonacyclic graph) can be reformulated as a finite-state deterministic optimal control problem, as we will see in Section 1.3.1. See also [Ber17], Section 2.1, and [Ber98] for an extensive discussion of shortest path methods, which connects with our discussion here.

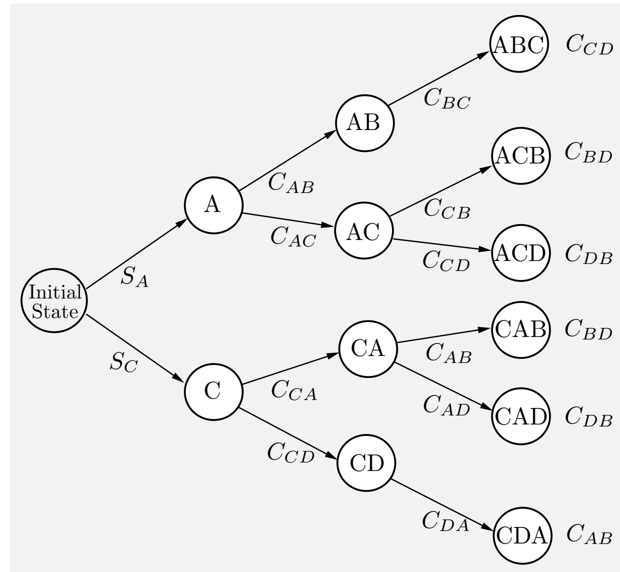

Figure 1.1.3 The transition graph of the deterministic scheduling problem of Example 1.1.1. Each arc of the graph corresponds to a decision leading from some state (the start node of the arc) to some other state (the end node of the arc). The corresponding cost is shown next to the arc. The cost of the last operation is shown as a terminal cost next to the terminal nodes of the graph.

Generally, combinatorial optimization problems can be formulated as deterministic finite-state finite-horizon optimal control problem. The following scheduling example illustrates the idea.

Example 1.1.1 (A Deterministic Scheduling Problem)

Suppose that to produce a certain product, four operations must be performed on a certain machine. The operations are denoted by A, B, C, and D. We assume that operation B can be performed only after operation A has been performed, and operation D can be performed only after operation C has been performed. (Thus the sequence CDAB is allowable but the sequence CDBA is not.) The setup cost C_{mn} for passing from any operation m to any other operation n is given. There is also an initial startup cost S_A or S_C for starting with operation A or C, respectively (cf. Fig. 1.1.3). The cost of a sequence is the sum of the setup costs associated with it; for example, the operation sequence ACDB has cost

$$S_A + C_{AC} + C_{CD} + C_{DB}.$$

We can view this problem as a sequence of three decisions, namely the choice of the first three operations to be performed (the last operation is

determined from the preceding three). It is appropriate to consider as state the set of operations already performed, the initial state being an artificial state corresponding to the beginning of the decision process. The possible state transitions corresponding to the possible states and decisions for this problem are shown in Fig. 1.1.3. Here the problem is deterministic, i.e., at a given state, each choice of control leads to a uniquely determined state. For example, at state AC the decision to perform operation D leads to state ACD with certainty, and has cost C_{CD}. Thus the problem can be conveniently represented in terms of the transition graph of Fig. 1.1.3. The optimal solution corresponds to the path that starts at the initial state and ends at some state at the terminal time and has minimum sum of arc costs plus the terminal cost.

Continuous-Spaces Optimal Control Problems

Many classical problems in control theory involve a state that belongs to a Euclidean space, i.e., the space of n-dimensional vectors of real variables, where n is some positive integer. The following is representative of the class of *linear-quadratic problems*, where the system equation is linear, the cost function is quadratic, and there are no control constraints. In our example, the states and controls are one-dimensional, but there are multidimensional extensions, which are very popular (see [Ber17], Section 3.1).

Example 1.1.2 (A Linear-Quadratic Problem)

A certain material is passed through a sequence of N ovens (see Fig. 1.1.4). Denote

x_0: initial temperature of the material,

x_k, $k = 1, \dots, N$: temperature of the material at the exit of oven k,

u_{k-1}, $k = 1, \dots, N$: heat energy applied to the material in oven k. In practice there will be some constraints on u_k, such as nonnegativity. However, for analytical tractability one may also consider the case where u_k is unconstrained, and check later if the solution satisfies some natural restrictions in the problem at hand.

We assume a system equation of the form

$$x_{k+1} = (1 - a)x_k + au_k, \qquad k = 0, 1, \dots, N - 1,$$

where a is a known scalar from the interval $(0, 1)$. The objective is to get the final temperature x_N close to a given target T, while expending relatively little energy. We express this with a cost function of the form

$$r(x_N - T)^2 + \sum_{k=0}^{N-1} u_k^2,$$

where $r > 0$ is a given scalar.

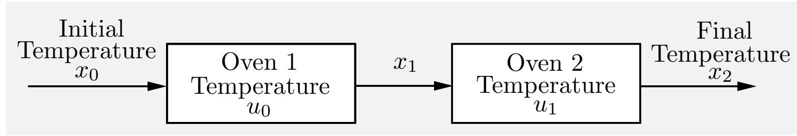

Figure 1.1.4 The linear-quadratic problem of Example 1.1.2 for $N = 2$. The temperature of the material evolves according to the system equation $x_{k+1} = (1-a)x_k + au_k$, where a is some scalar with $0 < a < 1$.

Linear-quadratic problems with no constraints on the state or the control admit a nice analytical solution, as we will see later in Section 1.3.6. In another frequently arising optimal control problem there are linear constraints on the state and/or the control. In the preceding example it would have been natural to require that $a_k \le x_k \le b_k$ and/or $c_k \le u_k \le d_k$, where a_k, b_k, c_k, d_k are given scalars. Then the problem would be solvable not only by DP but also by quadratic programming methods. Generally deterministic optimal control problems with continuous state and control spaces (in addition to DP) admit a solution by nonlinear programming methods, such as gradient, conjugate gradient, and Newton's method, which can be suitably adapted to their special structure.

1.1.2 The Dynamic Programming Algorithm

The DP algorithm rests on a simple idea, the *principle of optimality*, which roughly states the following; see Fig. 1.1.5.

Principle of Optimality

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

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

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

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

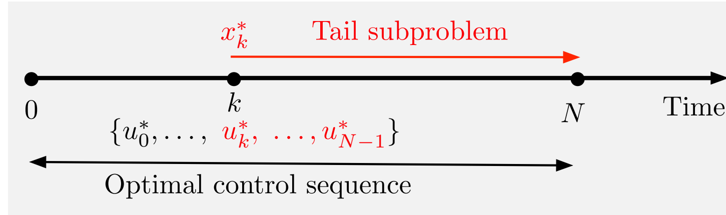

Figure 1.1.5 Illustration of the principle of optimality. The tail $\{u_k^*, \ldots, u_{N-1}^*\}$ of an optimal sequence $\{u_0^*, \ldots, u_{N-1}^*\}$ is optimal for the tail subproblem that starts at the state x_k^* of the optimal trajectory $\{x_1^*, \ldots, x_N^*\}$.

to an optimal sequence for the subproblem once we reach x_k^* (since the preceding choices $u_0^*, \ldots, u_{k-1}^*$ of controls do not restrict our future choices). For an auto travel analogy, suppose that the fastest route from Los Angeles to Boston passes through Chicago. The principle of optimality translates to the obvious fact that the Chicago to Boston portion of the route is also the fastest route for a trip that starts from Chicago and ends in Boston.

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

The DP algorithm is based on this idea: it proceeds sequentially, by *solving all the tail subproblems of a given time length, using the solution of the tail subproblems of shorter time length*. We illustrate the algorithm with the scheduling problem of Example 1.1.1. The calculations are simple but tedious, and may be skipped without loss of continuity. However, they may be worth going over by a reader that has no prior experience in the use of DP.

Example 1.1.1 (Scheduling Problem - Continued)

Let us consider the scheduling Example 1.1.1, and let us apply the principle of optimality to calculate the optimal schedule. We have to schedule optimally the four operations A, B, C, and D. The numerical values of the transition and setup costs are shown in Fig. 1.1.6 next to the corresponding arcs.

According to the principle of optimality, the "tail" portion of an optimal schedule must be optimal. For example, suppose that the optimal schedule is CABD. Then, having scheduled first C and then A, it must be optimal to complete the schedule with BD rather than with DB. With this in mind, we solve all possible tail subproblems of length two, then all tail subproblems of length three, and finally the original problem that has length four (the subproblems of length one are of course trivial because there is only one operation that is as yet unscheduled). As we will see shortly, the tail subproblems of

length $k + 1$ are easily solved once we have solved the tail subproblems of length k, and this is the essence of the DP technique.

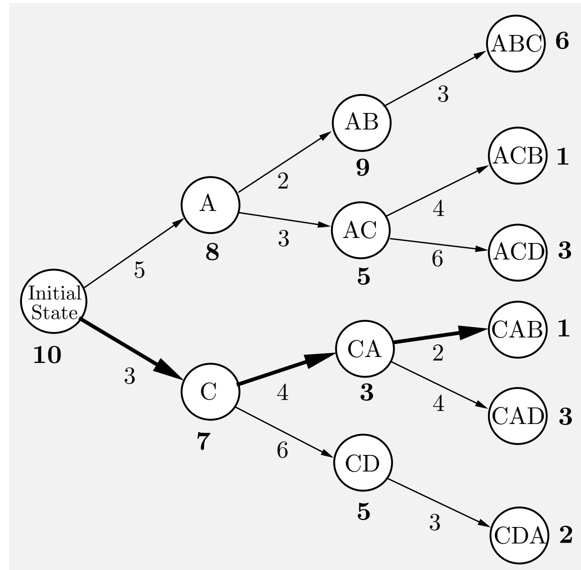

Figure 1.1.6 Transition graph of the deterministic scheduling problem, with the cost of each decision shown next to the corresponding arc. Next to each node/state we show the cost to optimally complete the schedule starting from that state. This is the optimal cost of the corresponding tail subproblem (cf. the principle of optimality). The optimal cost for the original problem is equal to 10, as shown next to the initial state. The optimal schedule corresponds to the thick-line arcs.

Tail Subproblems of Length 2: These subproblems are the ones that involve two unscheduled operations and correspond to the states AB, AC, CA, and CD (see Fig. 1.1.6).

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

State AC: Here the possibilities are to (a) schedule operation B and then D, which has cost 5, or (b) schedule operation D and then B, which has cost 9. The first possibility is optimal, and the corresponding cost of the tail subproblem is 5, as shown next to node AC in Fig. 1.1.6.

State CA: Here the possibilities are to (a) schedule operation B and then D, which has cost 3, or (b) schedule operation D and then B, which has cost 7. The first possibility is optimal, and the corresponding cost of

the tail subproblem is 3, as shown next to node CA in Fig. 1.1.6.

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

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

State A: Here the possibilities are to (a) schedule next operation B (cost 2) and then solve optimally the corresponding subproblem of length 2 (cost 9, as computed earlier), a total cost of 11, or (b) schedule next operation C (cost 3) and then solve optimally the corresponding subproblem of length 2 (cost 5, as computed earlier), a total cost of 8. The second possibility is optimal, and the corresponding cost of the tail subproblem is 8, as shown next to node A in Fig. 1.1.6.

State C: Here the possibilities are to (a) schedule next operation A (cost 4) and then solve optimally the corresponding subproblem of length 2 (cost 3, as computed earlier), a total cost of 7, or (b) schedule next operation D (cost 6) and then solve optimally the corresponding subproblem of length 2 (cost 5, as computed earlier), a total cost of 11. The first possibility is optimal, and the corresponding cost of the tail subproblem is 7, as shown next to node A in Fig. 1.1.6.

Original Problem of Length 4: The possibilities here are (a) start with operation A (cost 5) and then solve optimally the corresponding subproblem of length 3 (cost 8, as computed earlier), a total cost of 13, or (b) start with operation C (cost 3) and then solve optimally the corresponding subproblem of length 3 (cost 7, as computed earlier), a total cost of 10. The second possibility is optimal, and the corresponding optimal cost is 10, as shown next to the initial state node in Fig. 1.1.6.

Note that having computed the optimal cost of the original problem through the solution of all the tail subproblems, we can construct the optimal schedule: we begin at the initial node and proceed forward, each time choosing the optimal operation; i.e., the one that starts the optimal schedule for the corresponding tail subproblem. In this way, by inspection of the graph and the computational results of Fig. 1.1.6, we determine that CABD is the optimal schedule.

Finding an Optimal Control Sequence by DP

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

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

sequentially, starting from J_N^*, and proceeding backwards to J_{N-1}^*, J_{N-2}^*, etc.

DP Algorithm for Deterministic Finite Horizon Problems

Start with

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

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

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

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

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

where

$$J(x_k; u_k, \ldots, u_{N-1}) = g_N(x_N) + \sum_{m=k}^{N-1} g_m(x_m, u_m), \tag{1.6}$$

i.e., $J_k^*(x_k)$ is the optimal cost for an $(N-k)$-stage tail subproblem that starts at state x_k and time k, and ends at time N.†

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

$$\begin{aligned} J_k^*(x_k) &= \min_{\substack{u_m \in U_m(x_m) \\ m=k,\ldots,N-1}} \left[g_N(x_N) + \sum_{m=k}^{N-1} g_m(x_m, u_m) \right] \\ &= \min_{u_k \in U_k(x_k)} \Bigg[g_k(x_k, u_k) \\ &\qquad + \min_{\substack{u_m \in U_m(x_m) \\ m=k+1,\ldots,N-1}} \left[g_N(x_N) + \sum_{m=k+1}^{N-1} g_m(x_m, u_m) \right] \Bigg] \\ &= \min_{u_k \in U_k(x_k)} \Big[g_k(x_k, u_k) + J_{k+1}^*\big(f_k(x_k, u_k)\big) \Big], \end{aligned}$$

† Based on this fact, we call $J_k^*(x_k)$ the *optimal cost-to-go* at state x_k and time k, and refer to J_k^* as the *optimal cost-to-go function* or *optimal cost function* at time k. In maximization problems the DP algorithm (1.4) is written with maximization in place of minimization, and then J_k^* is referred to as the *optimal value function* at time k.

where for the last equality we use the induction hypothesis.†

Note that the algorithm solves every tail subproblem, i.e., the problem of minimization of the cost accumulated additively starting from an intermediate state up to the end of the horizon. Once the functions $J_0^*, \ldots, J_N^*$ have been obtained, we can use the following algorithm to construct an optimal control sequence $\{u_0^*, \ldots, u_{N-1}^*\}$ and corresponding state trajectory $\{x_1^*, \ldots, x_N^*\}$ for the given initial state x_0.

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

Set

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

and

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

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

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

and

$$x_{k+1}^* = f_k(x_k^*, u_k^*). \qquad (1.8)$$

The same algorithm can be used to find an optimal control sequence for any tail subproblem. Figure 1.1.6 traces the calculations of the DP algorithm for the scheduling Example 1.1.1. The numbers next to the nodes, give the corresponding cost-to-go values, and the thick-line arcs give the construction of the optimal control sequence using the preceding algorithm.

1.1.3 Approximation in Value Space

The preceding forward optimal control sequence construction is possible only after we have computed $J_k^*(x_k)$ by DP for all x_k and k. Unfortunately, in practice this is often prohibitively time-consuming, because of the number of possible x_k and k can be very large. However, a similar forward algorithmic process can be used if the optimal cost-to-go functions J_k^* are replaced by some approximations $\tilde{J}_k$. This is the basis for *approximation in value space*, which will be central in our future discussions. It

† A subtle mathematical point here is that, through the minimization operation, the cost-to-go functions J_k^* may take the value $-\infty$ for some x_k. Still the preceding induction argument is valid even if this is so.

constructs a suboptimal solution $\{\tilde{u}_0, \dots, \tilde{u}_{N-1}\}$ in place of the optimal $\{u_0^*, \dots, u_{N-1}^*\}$, based on using $\tilde{J}_k$ in place of J_k^* in the DP procedure (1.7).

Approximation in Value Space - Use of $\tilde{J}_k$ in Place of J_k^*

Start with

$$\tilde{u}_0 \in \arg \min_{u_0 \in U_0(x_0)} \Big[g_0(x_0, u_0) + \tilde{J}_1\big(f_0(x_0, u_0)\big) \Big],$$

and set

$$\tilde{x}_1 = f_0(x_0, \tilde{u}_0).$$

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

$$\tilde{u}_k \in \arg \min_{u_k \in U_k(\tilde{x}_k)} \Big[g_k(\tilde{x}_k, u_k) + \tilde{J}_{k+1}\big(f_k(\tilde{x}_k, u_k)\big) \Big], \tag{1.9}$$

and

$$\tilde{x}_{k+1} = f_k(\tilde{x}_k, \tilde{u}_k). \tag{1.10}$$

The construction of suitable approximate cost-to-go functions $\tilde{J}_k$ is a major focal point of the RL methodology. There are several possible methods, depending on the context, and they will be taken up starting with the next chapter.

Q-Factors and Q-Learning

The expression

$$\tilde{Q}_k(x_k, u_k) = g_k(x_k, u_k) + \tilde{J}_{k+1}\big(f_k(x_k, u_k)\big),$$

which appears in the right-hand side of Eq. (1.9) is known as the (approximate) *Q-factor of* (x_k, u_k).† In particular, the computation of the approximately optimal control (1.9) can be done through the Q-factor minimization

$$\tilde{u}_k \in \arg \min_{u_k \in U_k(\tilde{x}_k)} \tilde{Q}_k(\tilde{x}_k, u_k).$$

† The term "Q-learning" and some of the associated algorithmic ideas were introduced in the thesis by Watkins [Wat89] (after the symbol "Q" that he used to represent Q-factors). The term "Q-factor" was used in the book [BeT96], and is maintained here. Watkins [Wat89] used the term "action value" (at a given state), and the terms "state-action value" and "Q-value" are also common in the literature.

This suggests the possibility of using Q-factors in place of cost functions in approximation in value space schemes. Methods of this type use as starting point an alternative (and equivalent) form of the DP algorithm, which instead of the optimal cost-to-go functions J_k^*, generates the *optimal Q-factors*, defined for all pairs (x_k, u_k) and k by

$$Q_k^*(x_k, u_k) = g_k(x_k, u_k) + J_{k+1}^*\big(f_k(x_k, u_k)\big). \tag{1.11}$$

Thus the optimal Q-factors are simply the expressions that are minimized in the right-hand side of the DP equation (1.4). Note that this equation implies that the optimal cost function J_k^* can be recovered from the optimal Q-factor Q_k^* by means of

$$J_k^*(x_k) = \min_{u_k \in U_k(x_k)} Q_k^*(x_k, u_k).$$

Moreover, using the above relation, the DP algorithm can be written in an essentially equivalent form that involves Q-factors only

$$Q_k^*(x_k, u_k) = g_k(x_k, u_k) + \min_{u_{k+1} \in U_{k+1}(f_k(x_k, u_k))} Q_{k+1}^*\big(f_k(x_k, u_k), u_{k+1}\big).$$

We will discuss later exact and approximate forms of related algorithms in the context of a class of RL methods known as *Q-learning*.

1.2 STOCHASTIC DYNAMIC PROGRAMMING

The stochastic finite horizon optimal control problem differs from the deterministic version primarily in the nature of the discrete-time dynamic system that governs the evolution of the state x_k. This system includes a random "disturbance" w_k, which is characterized by a probability distribution $P_k(\cdot \mid x_k, u_k)$ that may depend explicitly on x_k and u_k, but not on values of prior disturbances $w_{k-1}, \ldots, w_0$. The system has the form

$$x_{k+1} = f_k(x_k, u_k, w_k), \qquad k = 0, 1, \ldots, N-1,$$

where as before x_k is an element of some state space S_k, the control u_k is an element of some control space. The cost per stage is denoted $g_k(x_k, u_k, w_k)$ and also depends on the random disturbance w_k; see Fig. 1.2.1. The control u_k is constrained to take values in a given subset $U(x_k)$, which depends on the current state x_k.

An important difference is that we optimize not over control sequences $\{u_0, \ldots, u_{N-1}\}$, but rather over *policies* (also called *closed-loop control laws*, or *feedback policies*) that consist of a sequence of functions

$$\pi = \{\mu_0, \ldots, \mu_{N-1}\},$$

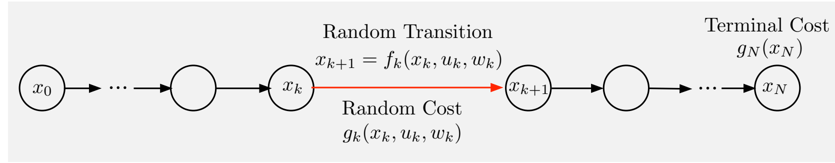

Figure 1.2.1 Illustration of an N-stage stochastic optimal control problem. Starting from state x_k, the next state under control u_k is generated randomly, according to

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

where w_k is the random disturbance, and a random stage cost $g_k(x_k, u_k, w_k)$ is incurred.

where μ_k maps states x_k into controls $u_k = \mu_k(x_k)$, and satisfies the control constraints, i.e., is such that $\mu_k(x_k) \in U_k(x_k)$ for all $x_k \in S_k$. Such policies will be called *admissible*. Policies are more general objects than control sequences, and in the presence of stochastic uncertainty, they can result in improved cost, since they allow choices of controls u_k that incorporate knowledge of the state x_k. Without this knowledge, the controller cannot adapt appropriately to unexpected values of the state, and as a result the cost can be adversely affected. This is a fundamental distinction between deterministic and stochastic optimal control problems.

Another important distinction between deterministic and stochastic problems is that in the latter, the evaluation of various quantities such as cost function values involves forming expected values, and this may necessitate the use of Monte Carlo simulation. In fact several of the methods that we will discuss for stochastic problems will involve the use of simulation.

Given an initial state x_0 and a policy $\pi = \{\mu_0, \ldots, \mu_{N-1}\}$, the future states x_k and disturbances w_k are random variables with distributions defined through the system equation

$$x_{k+1} = f_k\big(x_k, \mu_k(x_k), w_k\big), \qquad k = 0, 1, \ldots, N-1.$$

Thus, for given functions g_k, $k = 0, 1, \ldots, N$, the expected cost of π starting at x_0 is

$$J_\pi(x_0) = E\left\{ g_N(x_N) + \sum_{k=0}^{N-1} g_k\big(x_k, \mu_k(x_k), w_k\big) \right\},$$

where the expected value operation $E\{\cdot\}$ is over all the random variables w_k and x_k. An optimal policy π^* is one that minimizes this cost; i.e.,

$$J_{\pi^*}(x_0) = \min_{\pi \in \Pi} J_\pi(x_0),$$

where Π is the set of all admissible policies.

The optimal cost depends on x_0 and is denoted by $J^*(x_0)$; i.e.,

$$J^*(x_0) = \min_{\pi \in \Pi} J_\pi(x_0).$$

It is useful to view J^* as a function that assigns to each initial state x_0 the optimal cost $J^*(x_0)$, and call it the *optimal cost function* or *optimal value function*, particularly in problems of maximizing reward.

Finite Horizon Stochastic Dynamic Programming

The DP algorithm for the stochastic finite horizon optimal control problem has a similar form to its deterministic version, and shares several of its major characteristics:

(a) Using tail subproblems to break down the minimization over multiple stages to single stage minimizations.

(b) Generating backwards for all k and x_k the values $J_k^*(x_k)$, which give the optimal cost-to-go starting at stage k at state x_k.

(c) Obtaining an optimal policy by minimization in the DP equations.

(d) A structure that is suitable for approximation in value space, whereby we replace J_k^* by approximations $\tilde{J}_k$, and obtain a suboptimal policy by the corresponding minimization.

DP Algorithm for Stochastic Finite Horizon Problems

Start with

$$J_N^*(x_N) = g_N(x_N), \tag{1.12}$$

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

$$J_k^*(x_k) = \min_{u_k \in U_k(x_k)} E\Big\{ g_k(x_k, u_k, w_k) + J_{k+1}^*\big(f_k(x_k, u_k, w_k)\big)\Big\}. \tag{1.13}$$

If $u_k^* = \mu_k^*(x_k)$ minimizes the right side of this equation for each x_k and k, the policy $\pi^* = \{\mu_0^*, \ldots, \mu_{N-1}^*\}$ is optimal.

The key fact is that for every initial state x_0, the optimal cost $J^*(x_0)$ is equal to the function $J_0^*(x_0)$, obtained at the last step of the above DP algorithm. This can be proved by induction similar to the deterministic case; we will omit the proof (see the discussion of Section 1.3 in the textbook [Ber17]).†

As in deterministic problems, the DP algorithm can be very time-consuming, in fact more so since it involves the expected value operation

† There are some technical/mathematical difficulties here, having to do with

in Eq. (1.13). This motivates suboptimal control techniques, such as approximation in value space whereby we replace J_k^* with easier obtainable approximations $\tilde{J}_k$. We will discuss this approach at length in subsequent chapters.

Q-Factors for Stochastic Problems

We can define optimal Q-factors for stochastic problem, similar to the case of deterministic problems [cf. Eq. (1.11)], as the expressions that are minimized in the right-hand side of the stochastic DP equation (1.13). They are given by

$$Q_k^*(x_k,u_k) = E\Big\{g_k(x_k,u_k,w_k) + J_{k+1}^*\big(f_k(x_k,u_k,w_k)\big)\Big\}.$$

The optimal cost-to-go functions J_k^* can be recovered from the optimal Q-factors Q_k^* by means of

$$J_k^*(x_k) = \min_{u_k\in U_k(x_k)} Q_k^*(x_k,u_k),$$

and the DP algorithm can be written in terms of Q-factors as

$$Q_k^*(x_k,u_k) = E\Big\{g_k(x_k,u_k,w_k) + \min_{u_{k+1}\in U_{k+1}(f_k(x_k,u_k,w_k))} Q_{k+1}^*\big(f_k(x_k,u_k,w_k),u_{k+1}\big)\Big\}.$$

Note that the expected value in the right side of this equation can be approximated more easily by sampling and simulation than the right side of the DP algorithm (1.13). This will prove to be a critical mathematical point later when we discuss simulation-based algorithms for Q-factors.

1.3 EXAMPLES, VARIATIONS, AND SIMPLIFICATIONS

In this section we provide some examples to illustrate problem formulation techniques, solution methods, and adaptations of the basic DP algorithm to various contexts. As a guide for formulating optimal control problems in

---

the expected value operation in Eq. (1.13) being well-defined and finite. These difficulties are of no concern in practice, and disappear completely when the disturbance spaces w_k can take only a finite number of values, in which case all expected values consist of sums of finitely many real number terms. For a mathematical treatment, see the relevant discussion in Chapter 1 of [Ber17] and the book [BeS78].

a manner that is suitable for DP solution, the following two-stage process is suggested:

(a) Identify the controls/decisions u_k and the times k at which these controls are applied. Usually this step is fairly straightforward. However, in some cases there may be some choices to make. For example in deterministic problems, where the objective is to select an optimal sequence of controls $\{u_0, \ldots, u_{N-1}\}$, one may lump multiple controls to be chosen together, e.g., view the pair (u_0, u_1) as a single choice. This is usually not possible in stochastic problems, where distinct decisions are differentiated by the information/feedback available when making them.

(b) Select the states x_k. The basic guideline here is that x_k should encompass all the information that is known to the controller at time k and can be used with advantage in choosing u_k. In effect, at time k the state x_k should separate the past from the future, in the sense that anything that has happened in the past (states, controls, and disturbances from stages prior to stage k) is irrelevant to the choices of future controls as long we know x_k. Sometimes this is described by saying that the state should have a "Markov property" to express the similarity with states of Markov chains, where (by definition) the conditional probability distribution of future states depends on the past history of the chain only through the present state.

Note that there may be multiple possibilities for selecting the states, because information may be packaged in several different ways that are equally useful from the point of view of control. It is thus worth considering alternative ways to choose the states; for example try to use states that minimize the dimensionality of the state space. For a trivial example that illustrates the point, if a quantity x_k qualifies as state, then (x_{k-1}, x_k) also qualifies as state, since (x_{k-1}, x_k) contains all the information contained within x_k that can be useful to the controller when selecting u_k. However, using (x_{k-1}, x_k) in place of x_k, gains nothing in terms of optimal cost while complicating the DP algorithm which would be defined over a larger space. The concept of a *sufficient statistic*, which refers to a quantity that summarizes all the essential content of the information available to the controller, may be useful in reducing the size of the state space (see the discussion in Section 3.1.1, and in [Ber17], Section 4.3). Section 1.3.6 provides an example, and Section 3.1.1 contains further discussion.

Generally minimizing the dimension of the state makes sense but there are exceptions. A case in point is problems involving *partial* or *imperfect* state information, where we collect measurements to use for control of some quantity of interest y_k that evolves over time (for example, y_k may be the position/velocity vector of a moving vehicle). If I_k is the collection of all measurements up to time k, it is correct to use I_k as state. However,

a better alternative may be to use as state the conditional probability distribution $P_k(y_k \mid I_k)$, called *belief state*, which may subsume all the information that is useful for the purposes of choosing a control. On the other hand, the belief state $P_k(y_k \mid I_k)$ is an infinite-dimensional object, whereas I_k may be finite dimensional, so the best choice may be problem-dependent; see [Ber17] for further discussion of partial state information problems.

We refer to DP textbooks for extensive additional discussions of modeling and problem formulation techniques. The subsequent chapters do not rely substantially on the material of this section, so the reader may selectively skip forward to the next chapter and return to this material later as needed.

1.3.1 Deterministic Shortest Path Problems

Let $\{1, 2, \ldots, N, t\}$ be the set of nodes of a graph, and let a_{ij} be the cost of moving from node i to node j [also referred to as the *length* of the arc (i, j) that joins i and j]. Node t is a special node, which we call the *destination*. By a path we mean a sequence of arcs such that the end node of each arc in the sequence is the start node of the next arc. The length of a path from a given node to another node is the sum of the lengths of the arcs on the path. We want to find a shortest (i.e., minimum length) path from each node i to node t.

We make an assumption relating to cycles, i.e., paths of the form $(i, j_1), (j_1, j_2), \ldots, (j_k, i)$ that start and end at the same node. In particular, we exclude the possibility that a cycle has negative total length. Otherwise, it would be possible to decrease the length of some paths to arbitrarily small values simply by adding more and more negative-length cycles. We thus assume that *all cycles have nonnegative length*. With this assumption, it is clear that an optimal path need not take more than N moves, so we may limit the number of moves to N. We formulate the problem as one where *we require exactly N moves but allow degenerate moves from a node i to itself with cost $a_{ii} = 0$*. We also assume that *for every node i there exists at least one path from i to t*.

We can formulate this problem as a deterministic DP problem with N stages, where the states at any stage $0, \ldots, N-1$ are the nodes $\{1, \ldots, N\}$, the destination t is the unique state at stage N, and the controls correspond to the arcs (i, j), including the self arcs (i, i). Thus at each state i we select a control (i, j) and move to state j at cost a_{ij}.

We can write the DP algorithm for our problem, with the optimal cost-to-go functions J_k^* having the meaning

$$J_k^*(i) = \text{optimal cost of getting from } i \text{ to } t \text{ in } N - k \text{ moves},$$

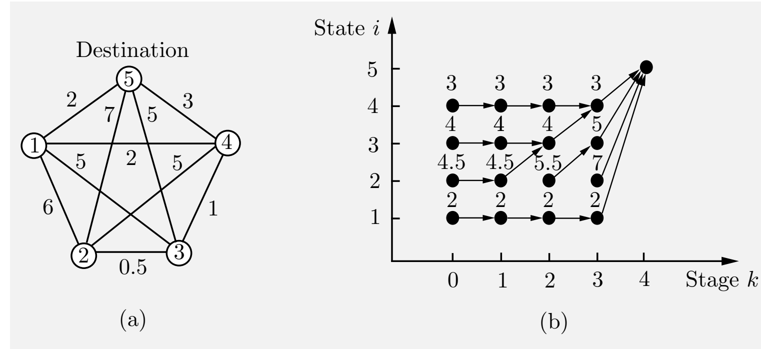

Figure 1.3.1 (a) Shortest path problem data. The destination is node 5. Arc lengths are equal in both directions and are shown along the line segments connecting nodes. (b) Costs-to-go generated by the DP algorithm. The number along stage k and state i is $J_k^*(i)$. Arrows indicate the optimal moves at each stage and node. The optimal paths are $1 \to 5$, $2 \to 3 \to 4 \to 5$, $3 \to 4 \to 5$, $4 \to 5$.

for $i = 1, \ldots, N$ and $k = 0, \ldots, N-1$. The cost of the optimal path from i to t is $J_0^*(i)$. The DP algorithm takes the intuitively clear form

$$\text{optimal cost from } i \text{ to } t \text{ in } N-k \text{ moves}$$
$$= \min_{\text{All arcs } (i,j)} \big[a_{ij} + (\text{optimal cost from } j \text{ to } t \text{ in } N-k-1 \text{ moves})\big],$$

or

$$J_k^*(i) = \min_{\text{All arcs } (i,j)} \big[a_{ij} + J_{k+1}^*(j)\big], \qquad k = 0, 1, \ldots, N-2,$$

with

$$J_{N-1}^*(i) = a_{it}, \qquad i = 1, 2, \ldots, N.$$

This algorithm is also known as the *Bellman-Ford algorithm* for shortest paths.

The optimal policy when at node i after k moves is to move to a node j^* that minimizes $a_{ij} + J_{k+1}^*(j)$ over all j such that (i, j) is an arc. If the optimal path obtained from the algorithm contains degenerate moves from a node to itself, this simply means that the path involves in reality less than N moves.

Note that if for some $k > 0$, we have

$$J_k^*(i) = J_{k+1}^*(i), \qquad \text{for all } i,$$

then subsequent DP iterations will not change the values of the cost-to-go $[J_{k-m}^*(i) = J_k^*(i)$ for all $m > 0$ and $i]$, so the algorithm can be terminated with $J_k^*(i)$ being the shortest distance from i to t, for all i.

To demonstrate the algorithm, consider the problem shown in Fig. 1.3.1(a) where the costs a_{ij} with $i \neq j$ are shown along the connecting line segments (we assume that $a_{ij} = a_{ji}$). Figure 1.3.1(b) shows the optimal cost-to-go $J_k^*(i)$ at each i and k together with the optimal paths.

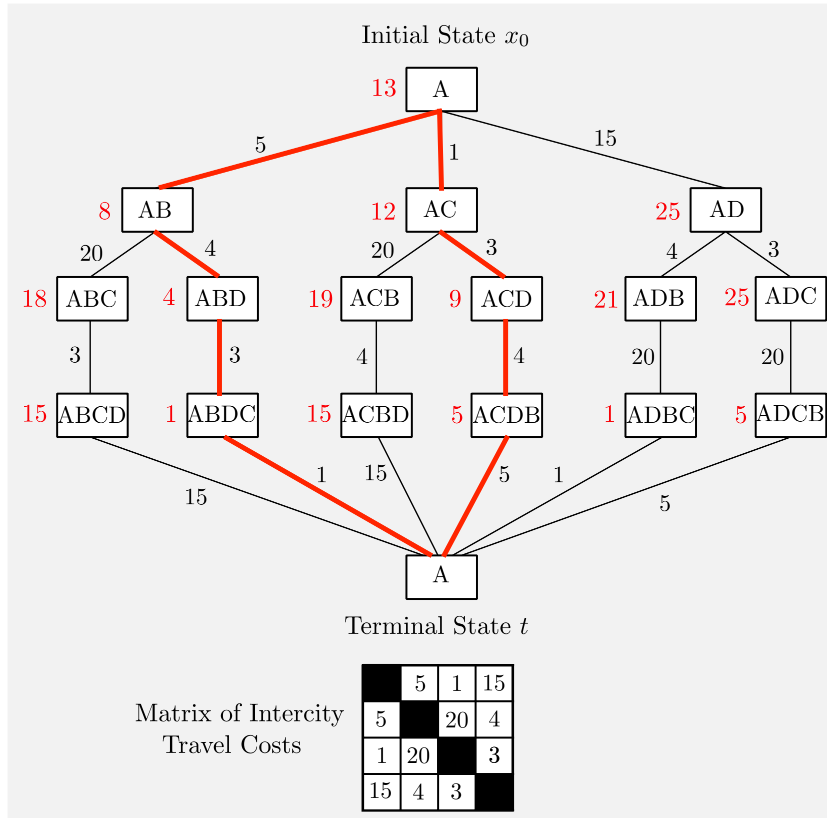

Figure 1.3.2 Example of a DP formulation of the traveling salesman problem. The travel times between the four cities A, B, C, and D are shown in the matrix at the bottom. We form a graph whose nodes are the k-city sequences and correspond to the states of the kth stage. The transition costs/travel times are shown next to the arcs. The optimal costs-to-go are generated by DP starting from the terminal state and going backwards towards the initial state, and are shown next to the nodes. There are two optimal sequences here (ABDCA and ACDBA), and they are marked with thick lines. Both optimal sequences can be obtained by forward minimization [cf. Eq. (1.7)], starting from the initial state x_0.

1.3.2 Discrete Deterministic Optimization

Discrete optimization problems can be formulated as DP problems by breaking down each feasible solution into a sequence of decisions/controls; as illustrated by the scheduling Example 1.1.1. This formulation will often lead to an intractable DP computation because of an exponential explosion of the number of states. However, it brings to bear approximate DP methods, such as rollout and others that we will discuss in future chapters. We illustrate the reformulation by means of an example and then we generalize.

Example 1.3.1 (The Traveling Salesman Problem)

An important model for scheduling a sequence of operations is the classical traveling salesman problem. Here we are given N cities and the travel time between each pair of cities. We wish to find a minimum time travel that visits each of the cities exactly once and returns to the start city. To convert this problem to a DP problem, we form a graph whose nodes are the sequences of k distinct cities, where $k = 1, \ldots, N$. The k-city sequences correspond to the states of the kth stage. The initial state x_0 consists of some city, taken as the start (city A in the example of Fig. 1.3.2). A k-city node/state leads to a $(k+1)$-city node/state by adding a new city at a cost equal to the travel time between the last two of the $k+1$ cities; see Fig. 1.3.2. Each sequence of N cities is connected to an artificial terminal node t with an arc of cost equal to the travel time from the last city of the sequence to the starting city, thus completing the transformation to a DP problem.

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

Let us now extend the ideas of the preceding example to the general discrete optimization problem:

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

where U is a finite set of feasible solutions and $G(u)$ is a cost function. We assume that each solution u has N components; i.e., it has the form $u = (u_1, \ldots, u_N)$, where N is a positive integer. We can then view the problem as a sequential decision problem, where the components $u_1, \ldots, u_N$ are selected one-at-a-time. A k-tuple $(u_1, \ldots, u_k)$ consisting of the first k components of a solution is called an *k-solution*. We associate k-solutions with the kth stage of the finite horizon DP problem shown in Fig. 1.3.3. In particular, for $k = 1, \ldots, N$, we view as the states of the kth stage all the k-tuples $(u_1, \ldots, u_k)$. The initial state is an artificial state denoted s. From this state we may move to any state (u_1), with u_1 belonging to the set

$$U_1 = \big\{ \tilde{u}_1 \mid \text{there exists a solution of the form } (\tilde{u}_1, \tilde{u}_2, \ldots, \tilde{u}_N) \in U \big\}.$$

Thus U_1 is the set of choices of u_1 that are consistent with feasibility.

More generally, from a state $(u_1, \ldots, u_k)$, we may move to any state of the form $(u_1, \ldots, u_k, u_{k+1})$, with u_{k+1} belonging to the set

$$\begin{aligned} U_{k+1}(u_1, \ldots, u_k) = \big\{ \tilde{u}_{k+1} \mid &\text{ there exists a solution of the form} \\ &(u_1, \ldots, u_k, \tilde{u}_{k+1}, \ldots, \tilde{u}_N) \in U \big\}. \end{aligned}$$

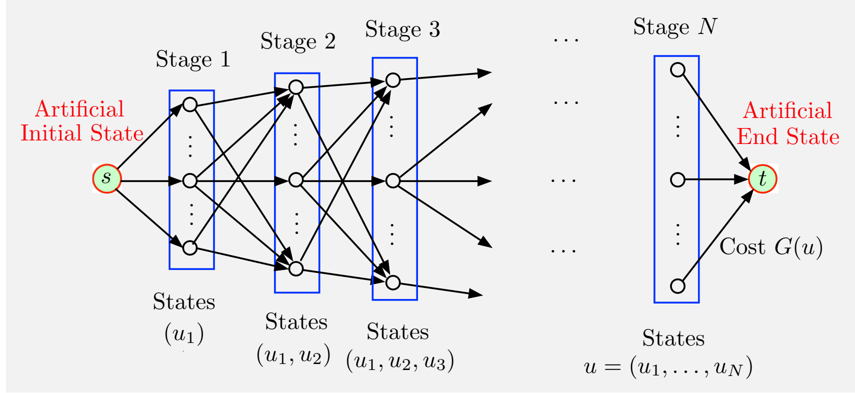

Figure 1.3.3. Formulation of a discrete optimization problem as a DP problem with $N+1$ stages. There is a cost $G(u)$ only at the terminal stage on the arc connecting an N-solution $u = (u_1, \ldots, u_N)$ to the artificial terminal state. Alternative formulations may use fewer states by taking advantage of the problem's structure.

At state $(u_1, \ldots, u_k)$ we must choose u_{k+1} from the set $U_{k+1}(u_1, \ldots, u_k)$. These are the choices of u_{k+1} that are consistent with the preceding choices $u_1, \ldots, u_k$, and are also consistent with feasibility. The terminal states correspond to the N-solutions $u = (u_1, \ldots, u_N)$, and the only nonzero cost is the terminal cost $G(u)$. This terminal cost is incurred upon transition from u to an artificial end state; see Fig. 1.3.3.

Let $J_k^*(u_1, \ldots, u_k)$ denote the optimal cost starting from the k-solution $(u_1, \ldots, u_k)$, i.e., the optimal cost of the problem over solutions whose first k components are constrained to be equal to u_i, $i = 1, \ldots, k$, respectively. The DP algorithm is described by the equation

$$J_k^*(u_1, \ldots, u_k) = \min_{u_{k+1} \in U_{k+1}(u_1, \ldots, u_k)} J_{k+1}^*(u_1, \ldots, u_k, u_{k+1}), \tag{1.14}$$

with the terminal condition

$$J_N^*(u_1, \ldots, u_N) = G(u_1, \ldots, u_N).$$

The algorithm (1.14) executes backwards in time: starting with the known function $J_N^* = G$, we compute J_{N-1}^*, then J_{N-2}^*, and so on up to computing J_1^*. An optimal solution $(u_1^*, \ldots, u_N^*)$ is then constructed by going forward through the algorithm

$$u_{k+1}^* \in \arg \min_{u_{k+1} \in U_{k+1}(u_1^*, \ldots, u_k^*)} J_{k+1}^*(u_1^*, \ldots, u_k^*, u_{k+1}), \quad k = 0, \ldots, N-1, \tag{1.15}$$

first compute u_1^*, then u_2^*, and so on up to u_N^*; cf. Eq. (1.7).

Of course here the number of states typically grows exponentially with N, but we can use the DP minimization (1.15) as a starting point for the use of approximation methods. For example we may try to use approximation in value space, whereby we replace J^*_{k+1} with some suboptimal $\tilde{J}_{k+1}$ in Eq. (1.15). One possibility is to use as

$$\tilde{J}_{k+1}(u_1^*, \ldots, u_k^*, u_{k+1}),$$

the cost generated by a heuristic method that solves the problem suboptimally with the values of the first $k+1$ decision components fixed at $u_1^*, \ldots, u_k^*, u_{k+1}$. This is called a *rollout algorithm*, and it is a very simple and effective approach for approximate combinatorial optimization. It will be discussed later in this book, in Chapter 2 for finite horizon stochastic problems, and in Chapter 4 for infinite horizon problems, where it will be related to the method of policy iteration.

Finally, let us mention that shortest path and discrete optimization problems with a sequential character can be addressed by a variety of approximate shortest path methods. These include the so called *label correcting, A^*, and branch and bound methods* for which extensive accounts can be found in the literature [the author's DP textbook [Ber17] (Chapter 2) contains a substantial account, which connects with the material of this section].

1.3.3 Problems with a Terminal State

Many DP problems of interest involve a *terminal state*, i.e., a state t that is cost-free and absorbing in the sense that

$$g_k(t, u_k, w_k) = 0, \quad f_k(t, u_k, w_k) = t, \qquad \text{for all } u_k \in U_k(t),\ k = 0, 1, \ldots.$$

Thus the control process essentially terminates upon reaching t, even if this happens before the end of the horizon. One may reach t by choice if a special stopping decision is available, or by means of a transition from another state.

Generally, when it is known that an optimal policy will reach the terminal state within at most some given number of stages N, the DP problem can be formulated as an N-stage horizon problem.† The reason is that even if the terminal state t is reached at a time $k < N$, we can extend our stay at t for an additional $N - k$ stages at no additional cost. An example is the deterministic shortest path problem that we discussed in Section 1.3.1.

† When an upper bound on the number of stages to termination is not known, the problem must be formulated as an infinite horizon problem, as will be discussed in a subsequent chapter.

Discrete deterministic optimization problems generally have a close connection to shortest path problems as we have seen in Section 1.3.2. In the problem discussed in that section, the terminal state is reached after exactly N stages (cf. Fig. 1.3.3), but in other problems it is possible that termination can happen earlier. The following well known puzzle is an example.

Example 1.3.2 (The Four Queens Problem)

Four queens must be placed on a 4×4 portion of a chessboard so that no queen can attack another. In other words, the placement must be such that every row, column, or diagonal of the 4×4 board contains at most one queen. Equivalently, we can view the problem as a sequence of problems; first, placing a queen in one of the first two squares in the top row, then placing another queen in the second row so that it is not attacked by the first, and similarly placing the third and fourth queens. (It is sufficient to consider only the first two squares of the top row, since the other two squares lead to symmetric positions; this is an example of a situation where we have a choice between several possible state spaces, but we select the one that is smallest.)

We can associate positions with nodes of an acyclic graph where the root node s corresponds to the position with no queens and the terminal nodes correspond to the positions where no additional queens can be placed without some queen attacking another. Let us connect each terminal position with an artificial terminal node t by means of an arc. Let us also assign to all arcs cost zero except for the artificial arcs connecting terminal positions with less than four queens with the artificial node t. These latter arcs are assigned a cost of 1 (see Fig. 1.3.4) to express the fact that they correspond to dead-end positions that cannot lead to a solution. Then, the four queens problem reduces to finding a minimal cost path from node s to node t, with an optimal sequence of queen placements corresponding to cost 0.

Note that once the states/nodes of the graph are enumerated, the problem is essentially solved. In this 4×4 problem the states are few and can be easily enumerated. However, we can think of similar problems with much larger state spaces. For example consider the problem of placing N queens on an $N \times N$ board without any queen attacking another. Even for moderate values of N, the state space for this problem can be extremely large (for $N = 8$ the number of possible placements with exactly one queen in each row is $8^8 = 16,777,216$). It can be shown that there exist solutions to this problem for all $N \geq 4$ (for $N = 2$ and $N = 3$, clearly there is no solution).

There are also several variants of the N queens problem. For example finding the minimal number of queens that can be placed on an $N \times N$ board so that they either occupy or attack every square; this is known as the *queen domination problem*. The minimal number can be found in principle by DP, and it is known for some N (for example the minimal number is 5 for $N = 8$), but not for all N (see e.g., the paper by Fernau [Fe10]).

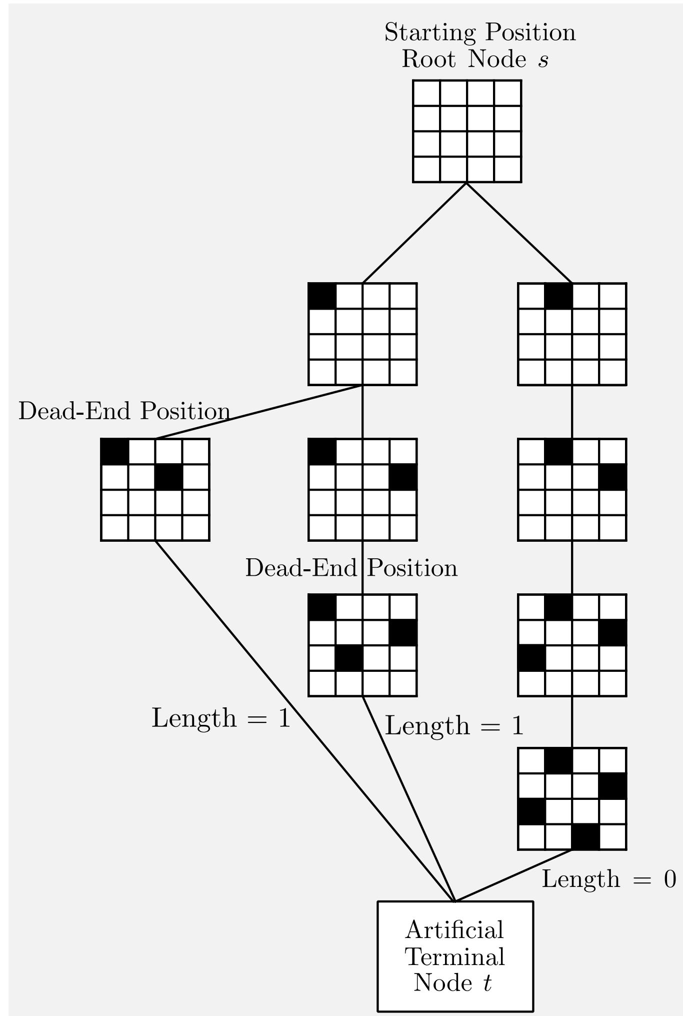

Figure 1.3.4 Discrete optimization formulation of the four queens problem. Symmetric positions resulting from placing a queen in one of the rightmost squares in the top row have been ignored. Squares containing a queen have been darkened. All arcs have length zero except for those connecting dead-end positions to the artificial terminal node.

1.3.4 Forecasts

Consider a situation where at time k the controller has access to a forecast y_k that results in a reassessment of the probability distribution of w_k and possibly of future disturbances. For example, y_k may be an exact prediction of w_k or an exact prediction that the probability distribution of w_k is a

specific one out of a finite collection of distributions. Forecasts of interest in practice are, for example, probabilistic predictions on the state of the weather, the interest rate for money, and the demand for inventory.

Generally, forecasts can be handled by introducing additional states corresponding to the information that the forecasts provide. We will illustrate the process with a simple example.

Assume that at the beginning of each stage k, the controller receives an accurate prediction that the next disturbance w_k will be selected according to a particular probability distribution out of a given collection of distributions $\{P_1, \ldots, P_m\}$; i.e., if the forecast is i, then w_k is selected according to P_i. The a priori probability that the forecast will be i is denoted by p_i and is given.

The forecasting process can be represented by means of the equation

$$y_{k+1} = \xi_k,$$

where y_{k+1} can take the values $1, \ldots, m$, corresponding to the m possible forecasts, and ξ_k is a random variable taking the value i with probability p_i. The interpretation here is that when ξ_k takes the value i, then w_{k+1} will occur according to the distribution P_i.

By combining the system equation with the forecast equation $y_{k+1} = \xi_k$, we obtain an augmented system given by

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} f_k(x_k, u_k, w_k) \\ \xi_k \end{pmatrix}.$$

The new state is

$$\tilde{x}_k = (x_k, y_k).$$

The new disturbance is

$$\tilde{w}_k = (w_k, \xi_k),$$

and its probability distribution is determined by the distributions P_i and the probabilities p_i, and depends explicitly on $\tilde{x}_k$ (via y_k) but not on the prior disturbances.

Thus, by suitable reformulation of the cost, the problem can be cast as a stochastic DP problem. Note that the control applied depends on both the current state and the current forecast. The DP algorithm takes the form

$$J_N^*(x_N, y_N) = g_N(x_N),$$

$$\begin{aligned} J_k^*(x_k, y_k) = \min_{u_k \in U_k(x_k)} \mathop{E}_{w_k} \Big\{ & g_k(x_k, u_k, w_k) \\ & + \sum_{i=1}^{m} p_i J_{k+1}^*\big(f_k(x_k, u_k, w_k), i\big) \;\big|\; y_k \Big\}, \end{aligned} \tag{1.16}$$

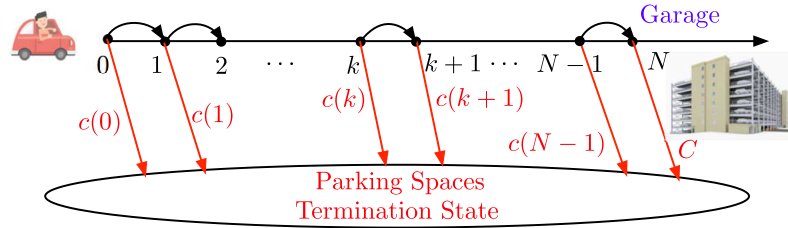

Figure 1.3.5 Cost structure of the parking problem. The driver may park at space $k = 0, 1, \ldots, N-1$ at cost $c(k)$, if the space is free, or continue to the next space $k+1$ at no cost. At space N (the garage) the driver must park at cost C.

where y_k may take the values $1, \ldots, m$, and the expectation over w_k is taken with respect to the distribution P_{y_k}.

It should be clear that the preceding formulation admits several extensions. One example is the case where forecasts can be influenced by the control action (e.g., pay extra for a more accurate forecast) and involve several future disturbances. However, the price for these extensions is increased complexity of the corresponding DP algorithm.

1.3.5 Problems with Uncontrollable State Components

In many problems of interest the natural state of the problem consists of several components, some of which cannot be affected by the choice of control. In such cases the DP algorithm can be simplified considerably, and be executed over the controllable components of the state. Before describing how this can be done in generality, let us consider an example.

Example 1.3.3 (Parking)

A driver is looking for inexpensive parking on the way to his destination. The parking area contains N spaces, and a garage at the end. The driver starts at space 0 and traverses the parking spaces sequentially, i.e., from space k he goes next to space $k+1$, etc. Each parking space k costs $c(k)$ and is free with probability $p(k)$ independently of whether other parking spaces are free or not. If the driver reaches the last parking space and does not park there, he must park at the garage, which costs C. The driver can observe whether a parking space is free only when he reaches it, and then, if it is free, he makes a decision to park in that space or not to park and check the next space. The problem is to find the minimum expected cost parking policy.

We formulate the problem as a DP problem with N stages, corresponding to the parking spaces, and an artificial terminal state t that corresponds to having parked; see Fig. 1.3.5. At each stage $k = 0, \ldots, N-1$, in addition to t, we have two states (k, F) and $(k, \overline{F})$, corresponding to space k being free or taken, respectively. The decision/control is to park or continue at state (k, F) [there is no choice at states $(k, \overline{F})$ and the garage].

Let us now derive the form of DP algorithm, denoting

$J_k^*(F)$: The optimal cost-to-go upon arrival at a space k that is free.

$J_k^*(\overline{F})$: The optimal cost-to-go upon arrival at a space k that is taken.

$J_N^*(t) = C$: The cost-to-go upon arrival at the garage.

$J_k^*(t) = 0$: The terminal cost-to-go.

The DP algorithm for $k = 0, \ldots, N-1$ takes the form

$$J_k^*(F) = \begin{cases} \min\Big[c(k),\, p(k+1)J_{k+1}^*(F) + \big(1 - p(k+1)\big)J_{k+1}^*(\overline{F})\Big] & \text{if } k < N-1, \\ \min\Big[c(N-1),\, C\Big] & \text{if } k = N-1, \end{cases}$$

$$J_k^*(\overline{F}) = \begin{cases} p(k+1)J_{k+1}^*(F) + \big(1 - p(k+1)\big)J_{k+1}^*(\overline{F}) & \text{if } k < N-1, \\ C & \text{if } k = N-1, \end{cases}$$

(we omit here the obvious equations for the terminal state t and the garage state N).

While this algorithm is easily executed, it can be written in a simpler and equivalent form, which takes advantage of the fact that the second component (F or $\overline{F}$) of the state is uncontrollable. This can be done by introducing the scalars

$$\hat{J}_k = p(k)J_k^*(F) + \big(1 - p(k)\big)J_k^*(\overline{F}), \qquad k = 0, \ldots, N-1,$$

which can be viewed as the optimal expected cost-to-go upon arriving at space k but *before verifying its free or taken status*.

Indeed, from the preceding DP algorithm, we have

$$\hat{J}_{N-1} = p(N-1)\min\Big[c(N-1),\, C\Big] + \big(1 - p(N-1)\big)C,$$

$$\hat{J}_k = p(k)\min\Big[c(k),\, \hat{J}_{k+1}\Big] + \big(1 - p(k)\big)\hat{J}_{k+1}, \qquad k = 0, \ldots, N-2.$$

From this algorithm we can also obtain the optimal parking policy, which is to park at space $k = 0, \ldots, N-1$ if it is free and $c(k) \leq \hat{J}_{k+1}$.

Figure 1.3.6 provides a plot for $\hat{J}_k$ for the case where

$$p(k) \equiv 0.05, \quad c(k) = N - k, \quad C = 100, \quad N = 200. \tag{1.17}$$

The optimal policy is to travel to space 165 and then to park at the first available space. The reader may verify that this type of policy, characterized by a single threshold distance, is optimal assuming that $c(k)$ is monotonically decreasing with k.

We will now formalize the procedure illustrated in the preceding example. Let the state of the system be a composite (x_k, y_k) of two components x_k and y_k. The evolution of the main component, x_k, is affected by the control u_k according to the equation

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

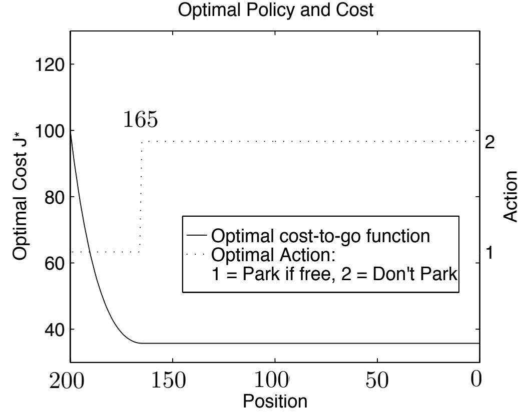

Figure 1.3.6 Optimal cost-to-go and optimal policy for the parking problem with the data in Eq. (1.17). The optimal policy is to travel from space 0 to space 165 and then to park at the first available space.

where the probability distribution $P_k(w_k \mid x_k, y_k, u_k)$ is given. The evolution of the other component, y_k, is governed by a given conditional distribution $P_k(y_k \mid x_k)$ and cannot be affected by the control, except indirectly through x_k. One is tempted to view y_k as a disturbance, but there is a difference: y_k is observed by the controller before applying u_k, while w_k occurs after u_k is applied, and indeed w_k may probabilistically depend on u_k.

We will formulate a DP algorithm that is executed over the controllable component of the state, with the dependence on the uncontrollable component being "averaged out" similar to the preceding example. In particular, let $J_k^*(x_k, y_k)$ denote the optimal cost-to-go at stage k and state (x_k, y_k), and define

$$\hat{J}_k(x_k) = \mathop{E}_{y_k}\big\{ J_k^*(x_k, y_k) \mid x_k \big\}.$$

We will derive a DP algorithm that generates $\hat{J}_k(x_k)$.

Indeed, we have

$$\begin{aligned}
\hat{J}_k(x_k) &= E_{y_k}\big\{ J_k^*(x_k, y_k) \mid x_k \big\} \\
&= E_{y_k}\Big\{ \min_{u_k \in U_k(x_k, y_k)} E_{w_k, x_{k+1}, y_{k+1}} \big\{ g_k(x_k, y_k, u_k, w_k) \\
&\qquad\qquad + J_{k+1}^*(x_{k+1}, y_{k+1}) \bigm| x_k, y_k, u_k \big\} \Bigm| x_k \Big\} \\
&= E_{y_k}\Big\{ \min_{u_k \in U_k(x_k, y_k)} E_{w_k, x_{k+1}} \Big\{ g_k(x_k, y_k, u_k, w_k) \\
&\qquad\qquad + E_{y_{k+1}} \big\{ J_{k+1}^*(x_{k+1}, y_{k+1}) \bigm| x_{k+1} \big\} \Bigm| x_k, y_k, u_k \Big\} \Bigm| x_k \Big\},
\end{aligned}$$

and finally

$$\hat{J}_k(x_k) = \underset{y_k}{E}\left\{ \min_{u_k \in U_k(x_k,y_k)} \underset{w_k}{E}\Big\{ g_k(x_k,y_k,u_k,w_k) + \hat{J}_{k+1}\big(f_k(x_k,y_k,u_k,w_k)\big)\Big\} \;\Big|\; x_k \right\}. \tag{1.18}$$

The advantage of this equivalent DP algorithm is that it is executed over a significantly reduced state space. For example, if x_k takes n possible values and y_k takes m possible values, then DP is executed over n states instead of nm states. Note, however, that the minimization in the right-hand side of the preceding equation yields an optimal control law as a function of the full state (x_k, y_k).

As an example, consider the augmented state resulting from the incorporation of forecasts, as described earlier in Section 1.3.4. Then, the forecast y_k represents an uncontrolled state component, so that the DP algorithm can be simplified as in Eq. (1.18). In particular, using the notation of Section 1.3.4, by defining

$$\hat{J}_k(x_k) = \sum_{i=1}^{m} p_i J_k^*(x_k, i), \qquad k = 0, 1, \ldots, N-1,$$

and

$$\hat{J}_N(x_N) = g_N(x_N),$$

we have, using Eq. (1.16),

$$\hat{J}_k(x_k) = \sum_{i=1}^{m} p_i \min_{u_k \in U_k(x_k)} \underset{w_k}{E}\Big\{ g_k(x_k,u_k,w_k) + \hat{J}_{k+1}\big(f_k(x_k,u_k,w_k)\big) \;\big|\; y_k = i \Big\},$$

which is executed over the space of x_k rather than x_k *and* y_k. This is a simpler algorithm than the one of Eq. (1.16).

Uncontrollable state components often occur in arrival systems, such as queueing, where action must be taken in response to a random event (such as a customer arrival) that cannot be influenced by the choice of control. Then the state of the arrival system must be augmented to include the random event, but the DP algorithm can be executed over a smaller space, as per Eq. (1.18). Here is another example of similar type.

Example 1.3.4 (Tetris)

Tetris is a popular video game played on a two-dimensional grid. Each square in the grid can be full or empty, making up a "wall of bricks" with "holes" and a "jagged top" (see Fig. 1.3.7). The squares fill up as blocks of different

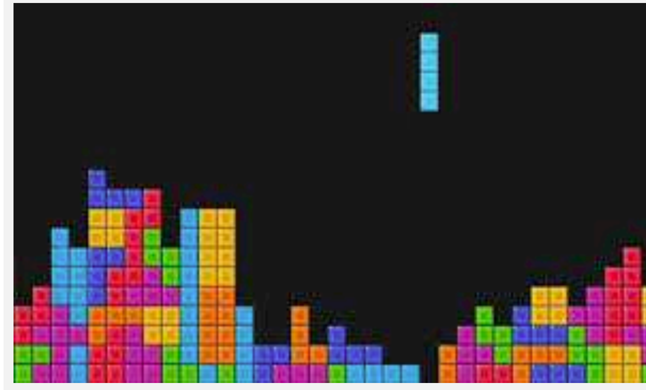

Figure 1.3.7 Illustration of a tetris board.

shapes fall from the top of the grid and are added to the top of the wall. As a given block falls, the player can move horizontally and rotate the block in all possible ways, subject to the constraints imposed by the sides of the grid and the top of the wall. The falling blocks are generated independently according to some probability distribution, defined over a finite set of standard shapes. The game starts with an empty grid and ends when a square in the top row becomes full and the top of the wall reaches the top of the grid. When a row of full squares is created, this row is removed, the bricks lying above this row move one row downward, and the player scores a point. The player's objective is to maximize the score attained (total number of rows removed) within N steps or up to termination of the game, whichever occurs first.

We can model the problem of finding an optimal tetris playing strategy as a stochastic DP problem. The control, denoted by u, is the horizontal positioning and rotation applied to the falling block. The state consists of two components:

(1) The board position, i.e., a binary description of the full/empty status of each square, denoted by x.

(2) The shape of the current falling block, denoted by y.

There is also an additional termination state which is cost-free. Once the state reaches the termination state, it stays there with no change in cost.

The shape y is generated according to a probability distribution $p(y)$, independently of the control, so it can be viewed as an uncontrollable state component. The DP algorithm (1.18) is executed over the space of x and has the intuitive form

$$\hat{J}_k(x) = \sum_y p(y) \max_u \Big[g(x,y,u) + \hat{J}_{k+1}\big(f(x,y,u)\big) \Big], \qquad \text{for all } x,$$

where

$g(x,y,u)$ is the number of points scored (rows removed),

$f(x, y, u)$ is the board position (or termination state),

when the state is (x, y) and control u is applied, respectively. Note, however, that despite the simplification in the DP algorithm achieved by eliminating the uncontrollable portion of the state, the number of states x is enormous, and the problem can only be addressed by suboptimal methods, which will be discussed later in this book.

1.3.6 Partial State Information and Belief States

We have assumed so far that the controller has access to the exact value of the current state x_k, so a policy consists of a sequence of functions $\mu_k(x_k)$, $k = 0, \ldots, N-1$. However, in many practical settings this assumption is unrealistic, because some components of the state may be inaccessible for measurement, the sensors used for measuring them may be inaccurate, or the cost of obtaining accurate measurements may be prohibitive.

Often in such situations the controller has access to only some of the components of the current state, and the corresponding measurements may also be corrupted by stochastic uncertainty. For example in three-dimensional motion problems, the state may consist of the six-tuple of position and velocity components, but the measurements may consist of noise-corrupted radar measurements of the three position components. This gives rise to problems of *partial* or *imperfect* state information, which have received a lot of attention in the optimization and artificial intelligence literature (see e.g., [Ber17], Ch. 4). Even though there are DP algorithms for partial information problems, these algorithms are far more computationally intensive than their perfect information counterparts. For this reason, in the absence of an analytical solution, partial information problems are typically solved suboptimally in practice.

On the other hand it turns out that conceptually, partial state information problems are no different than the perfect state information problems we have been addressing so far. In fact by various reformulations, we can reduce a partial state information problem to one with perfect state information (see [Ber17], Ch. 4). The most common approach is to replace the state x_k with a *belief state*, which is the probability distribution of x_k given all the observations that have been obtained by the controller up to time k (see Fig. 1.3.8). This probability distribution can in principle be computed, and it can serve as "state" in an appropriate DP algorithm. We illustrate this process with a simple example.

Example 1.3.5 (Treasure Hunting)

In a classical problem of search, one has to decide at each of N periods whether to search a site that may contain a treasure. If a treasure is present, the search reveals it with probability β, in which case the treasure is removed from the site. Here the state x_k has two values: either a treasure is present in

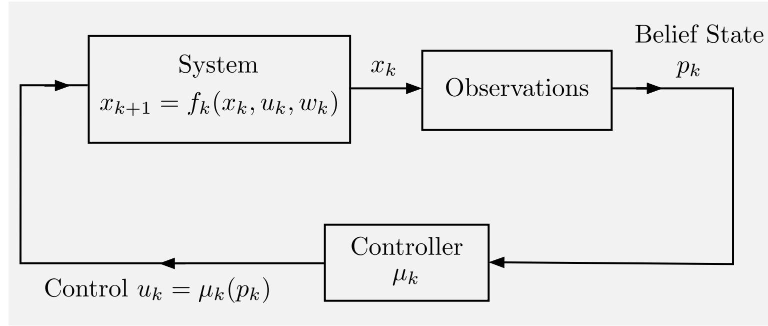

Figure 1.3.8 Schematic illustration of a control system with imperfect state observations. The belief state p_k is the conditional probability distribution of x_k given all the observations up to time k.

the site or it is not. The control u_k takes two values: search and not search. If the site is searched, we obtain an observation, which takes one of two values: treasure found or not found. If the site is not searched, no information is obtained.

Denote

p_k : probability a treasure is present at the beginning of period k.

This is the belief state at time k and it evolves according to the equation

$$p_{k+1} = \begin{cases} p_k & \text{if the site is not searched at time } k, \\ 0 & \text{if the site is searched and a treasure is found,} \\ \frac{p_k(1-\beta)}{p_k(1-\beta)+1-p_k} & \text{if the site is searched but no treasure is found.} \end{cases} \tag{1.19}$$

The third relation above follows by application of Bayes' rule (p_{k+1} is equal to the kth period probability of a treasure being present *and* the search being unsuccessful, divided by the probability of an unsuccessful search). The second relation holds because the treasure is removed after a successful search.

Let us view p_k as the state of a "belief system" given by Eq. (1.19), and write a DP algorithm, assuming that the treasure's worth is V, that each search costs C, and that once we decide not to search at a particular time, then we cannot search at future times. The algorithm takes the form

$$J_k^*(p_k) = \max \left[J_{k+1}^*(p_k), \right.$$
$$\left. - C + p_k\beta V + (1 - p_k\beta)\hat{J}_{k+1}\left(\frac{p_k(1-\beta)}{p_k(1-\beta) + 1 - p_k}\right) \right], \tag{1.20}$$

with $\hat{J}_N(p_N) = 0$.

This DP algorithm can be used to obtain an analytical solution. In particular, it is straightforward to show by induction that the functions $\hat{J}_k$

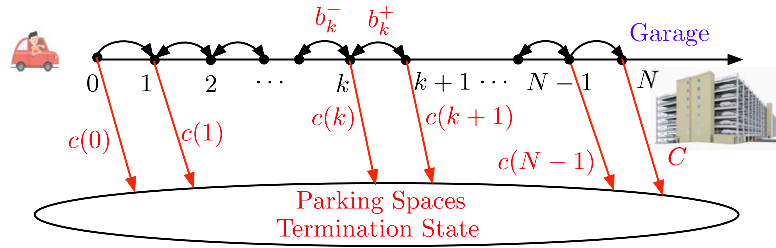

Figure 1.3.9 Cost structure and transitions of the bidirectional parking problem. The driver may park at space $k = 0, 1, \ldots, N-1$ at cost $c(k)$, if the space is free, can move to $k-1$ at cost β_k^- or can move to $k+1$ at cost β_k^+. At space N (the garage) the driver must park at cost C.

satisfy $\hat{J}_k(p_k) \geq 0$ if $p_k \in [0,1]$ and

$$\hat{J}_k(p_k) = 0 \qquad \text{if} \qquad p_k \leq \frac{C}{\beta V}.$$

From this it follows that it is optimal to search at period k if and only if

$$\frac{C}{\beta V} \leq p_k.$$

Thus, it is optimal to search if and only if the expected reward from the next search, $p_k \beta V$, is greater or equal to the cost C of the search - a myopic policy that focuses on just the next stage.

Of course the preceding example is extremely simple, involving a state x_k that takes just two values. As a result, the belief state p_k takes values within the interval $[0,1]$. Still there are infinitely many values in this interval, and if a computational solution were necessary, the belief state would have to be discretized and the DP algorithm (1.20) would have to be adapted to the discretization.

In problems where the state x_k can take a finite but large number of values, say n, the belief states comprise an n-dimensional simplex, so discretization becomes problematic. As a result, alternative suboptimal solution methods are often used in partial state information problems. Some of these methods will be described in future chapters.

The following is a simple example of a partial state information problem whose belief state has large enough size to make an exact DP solution impossible.

Example 1.3.6 (Bidirectional Parking)

Let us consider a more complex version of the parking problem of Example 1.3.3. As in that example, a driver is looking for inexpensive parking on the

way to his destination, along a line of N parking spaces with a garage at the end. The difference is that the driver can move in either direction, rather than just forward towards the garage. In particular, at space i, the driver can park at cost $c(i)$ if i is free, can move to $i-1$ at a cost β_i^- or can move to $i+1$ at a cost β_i^+. Moreover, the driver records the free/taken status of the spaces previously visited and may return to any of these spaces; see Fig. 1.3.9.

Let us assume that the probability $p(i)$ of a space i being free changes over time, i.e., a space found free (or taken) at a given visit may get taken (or become free, respectively) by the time of the next visit. The initial probabilities $p(i)$, before visiting any spaces, are known, and the mechanism by which these probabilities change over time is also known to the driver. As an example, we may assume that at each time period, $p(i)$ increases by a certain known factor with some probability ξ and decreases by another known factor with the complementary probability $1-\xi$.

Here the belief state is the vector of current probabilities

$$\big(p(1), \ldots, p(N)\big),$$

and it is updated at each time based on the new observation: the free/taken status of the space visited at that time. Thus the belief state can be computed exactly by the driver, given the parking status observations of the spaces visited thus far. While it is possible to state an exact DP algorithm that is defined over the set of belief states, and we will do so later, the algorithm is impossible to execute in practice.† Thus the problem can only be solved with approximations.

Example 1.3.7 (Bidirectional Parking - Sufficient Statistics)

It is instructive to consider a simple special case of the preceding bidirectional parking example. This is the case where the spaces do not change status, during the parking process. Thus a space that was earlier observed to be free, remains free for the driver to return at a later time, while taken spaces remain taken. The state is now simplified: it consists of the driver's current position, and the free/taken status of all the spaces visited thus far. This is still a very large state space, but it turns out that there is a much simpler sufficient statistic (cf. the discussion in the introduction to this section): this is the free/taken status of the driver's current position, together with the location of *the best space found to be free earlier*; "best" empty space here means that it has minimal parking cost plus travel cost to reach it from the current position.

---

† The problem as stated is an infinite horizon problem because there is nothing to prevent the driver from moving forever in the parking lot without ever parking. We can convert the problem to a similarly difficult finite horizon problem by restricting the number of moves to a given upper limit $\overline{N} > N$, and requiring that if the driver is at distance of k spaces from the garage at time $\overline{N} - k$, then driving in the direction away from the garage is not an option.

The number of stages of the DP algorithm is N [counting a decision at stage k to park at an earlier seen empty space $i \leq k$ as a single stage with cost equal to the parking cost $c(i)$ plus the travel cost from k to i]. The DP algorithm may be executed over the space of earlier seen empty spaces i_k at stage k, which consists of the integer values $0, 1, \ldots, k$, plus an extra state denoted -1, which corresponds to all spaces seen so far being taken.

To state this DP algorithm, let $t(k,i)$ denote the cost of traveling from space k to space $i \leq k$ plus the cost $c(i)$ of parking at i, with $t(k,-1) = \infty$. Let also $f_k(i,F)$ and $f_k(i,\overline{F})$ denote the two possible values of the next location of best space found to be free after space $k+1$ is visited, given that the best space was i after space k was visited,

$$f_k(i,F) = \begin{cases} i & \text{if } t(k+1,i) < c(k+1), \\ k+1 & \text{if } t(k+1,i) \geq c(k+1), \end{cases}$$

$$f_k(i,\overline{F}) = i,$$

corresponding to space $k+1$ being free or taken (F or $\overline{F}$, respectively). The DP algorithm has the form

$$\hat{J}_{N-1}(i_{N-1}) = \min\big[t(N-1,i_{N-1}),\, C\big],$$

and for $k = 0, \ldots, N-2$,

$$\hat{J}_k(i_k) = p(k)\min\Big[t(k,i_k),\, E\big\{\hat{J}_{k+1}(i_{k+1})\big\}\Big] + \big(1-p(k)\big)E\big\{\hat{J}_{k+1}(i_{k+1})\big\},$$

where we denote by $E\big\{\hat{J}_{k+1}(i_{k+1})\big\}$ the expected value of the cost-to-go at the next state i_{k+1}:

$$\begin{aligned} E\big\{\hat{J}_{k+1}(i_{k+1})\big\} = {} & p(k+1)\hat{J}_{k+1}\big(f_k(i_k,F)\big) \\ & + \big(1-p(k+1)\big)\hat{J}_{k+1}\big(f_k(i_k,\overline{F})\big). \end{aligned}$$

This algorithm should be compared with its one-directional counterpart of Example 1.3.3, which has the form

$$\hat{J}_{N-1} = \min\big[c(N-1),\, C\big],$$

$$\hat{J}_k = p(k)\min\big[c(k),\, \hat{J}_{k+1}\big] + \big(1-p(k)\big)\hat{J}_{k+1}, \qquad k = 0, \ldots, N-2.$$

1.3.7 Linear Quadratic Optimal Control

In a few exceptional special cases the DP algorithm yields an analytical solution, which can be used among other purposes, as a starting point for approximate DP schemes. Prominent among such cases are various linear quadratic optimal control problems, which involve a linear (possibly multidimensional) system, a quadratic cost function, and no constraints

on the control. Let us illustrate this with the deterministic scalar linear quadratic Example 1.1.2. We will apply the DP algorithm for the case of just two stages ($N = 2$), and illustrate the method for obtaining a nice analytical solution.

As defined in Example 1.1.2, the terminal cost is

$$g_2(x_2) = r(x_2 - T)^2.$$

Thus the DP algorithm starts with

$$J_2^*(x_2) = g_2(x_2) = r(x_2 - T)^2,$$

[cf. Eq. (1.3)].

For the next-to-last stage, we have [cf. Eq. (1.4)]

$$J_1^*(x_1) = \min_{u_1}\big[u_1^2 + J_2^*(x_2)\big] = \min_{u_1}\Big[u_1^2 + J_2^*\big((1-a)x_1 + au_1\big)\Big].$$

Substituting the previous form of J_2^*, we obtain

$$J_1^*(x_1) = \min_{u_1}\Big[u_1^2 + r\big((1-a)x_1 + au_1 - T\big)^2\Big]. \tag{1.21}$$

This minimization will be done by setting to zero the derivative with respect to u_1. This yields

$$0 = 2u_1 + 2ra\big((1-a)x_1 + au_1 - T\big),$$

and by collecting terms and solving for u_1, we obtain the optimal temperature for the last oven as a function of x_1:

$$\mu_1^*(x_1) = \frac{ra\big(T - (1-a)x_1\big)}{1 + ra^2}. \tag{1.22}$$

By substituting the optimal u_1 in the expression (1.21) for J_1^*, we obtain

$$\begin{aligned}
J_1^*(x_1) &= \frac{r^2a^2\big((1-a)x_1 - T\big)^2}{(1+ra^2)^2} + r\left((1-a)x_1 + \frac{ra^2\big(T-(1-a)x_1\big)}{1+ra^2} - T\right)^2 \\
&= \frac{r^2a^2\big((1-a)x_1 - T\big)^2}{(1+ra^2)^2} + r\left(\frac{ra^2}{1+ra^2} - 1\right)^2\big((1-a)x_1 - T\big)^2 \\
&= \frac{r\big((1-a)x_1 - T\big)^2}{1+ra^2}.
\end{aligned}$$

We now go back one stage. We have [cf. Eq. (1.4)]

$$J_0^*(x_0) = \min_{u_0}\big[u_0^2 + J_1^*(x_1)\big] = \min_{u_0}\Big[u_0^2 + J_1^*\big((1-a)x_0 + au_0\big)\Big],$$

and by substituting the expression already obtained for J_1^*, we have

$$J_0^*(x_0) = \min_{u_0} \left[u_0^2 + \frac{r\big((1-a)^2 x_0 + (1-a)a u_0 - T\big)^2}{1 + ra^2} \right].$$

We minimize with respect to u_0 by setting the corresponding derivative to zero. We obtain

$$0 = 2u_0 + \frac{2r(1-a)a\big((1-a)^2 x_0 + (1-a)a u_0 - T\big)}{1 + ra^2}.$$

This yields, after some calculation, the optimal temperature of the first oven:

$$\mu_0^*(x_0) = \frac{r(1-a)a\big(T - (1-a)^2 x_0\big)}{1 + ra^2\big(1 + (1-a)^2\big)}. \tag{1.23}$$

The optimal cost is obtained by substituting this expression in the formula for J_0^*. This leads to a straightforward but lengthy calculation, which in the end yields the rather simple formula

$$J_0^*(x_0) = \frac{r\big((1-a)^2 x_0 - T\big)^2}{1 + ra^2\big(1 + (1-a)^2\big)}.$$

This completes the solution of the problem.

Note that the algorithm has simultaneously yielded an optimal policy $\{\mu_0^*, \mu_1^*\}$ via Eqs. (1.23) and (1.22): a rule that tells us the optimal oven temperatures $u_0 = \mu_0^*(x_0)$ and $u_1 = \mu_1^*(x_1)$ for every possible value of the states x_0 and x_1, respectively. Thus the DP algorithm solves all the tail subproblems and provides a feedback policy.

A noteworthy feature in this example is the facility with which we obtained an analytical solution. A little thought while tracing the steps of the algorithm will convince the reader that what simplifies the solution is the quadratic nature of the cost and the linearity of the system equation. Indeed, it can be shown in generality that when the system is linear and the cost is quadratic, the optimal policy and cost-to-go function are given by closed-form expressions, regardless of the number of stages N (see [Ber17], Section 3.1).

Stochastic Linear Quadratic Problems - Certainty Equivalence

Let us now introduce a zero-mean stochastic additive disturbance in the linear system equation. Remarkably, it turns out that the optimal policy remains unaffected. To see this, assume that the material's temperature evolves according to

$$x_{k+1} = (1-a)x_k + a u_k + w_k, \qquad k = 0, 1,$$

where w_0 and w_1 are independent random variables with given distribution, zero mean

$$E\{w_0\} = E\{w_1\} = 0,$$

and finite variance. Then the equation for J_1^* [cf. Eq. (1.4)] becomes

$$\begin{aligned} J_1^*(x_1) &= \min_{u_1} \mathop{E}_{w_1} \Big\{ u_1^2 + r\big((1-a)x_1 + au_1 + w_1 - T\big)^2 \Big\} \\ &= \min_{u_1} \Big[u_1^2 + r\big((1-a)x_1 + au_1 - T\big)^2 \\ &\qquad + 2rE\{w_1\}\big((1-a)x_1 + au_1 - T\big) + rE\{w_1^2\} \Big]. \end{aligned}$$

Since $E\{w_1\} = 0$, we obtain

$$J_1^*(x_1) = \min_{u_1} \Big[u_1^2 + r\big((1-a)x_1 + au_1 - T\big)^2 \Big] + rE\{w_1^2\}.$$

Comparing this equation with Eq. (1.21), we see that the presence of w_1 has resulted in an additional inconsequential constant term, $rE\{w_1^2\}$. Therefore, the optimal policy for the last stage remains unaffected by the presence of w_1, while $J_1^*(x_1)$ is increased by $rE\{w_1^2\}$. It can be seen that a similar situation also holds for the first stage. In particular, the optimal cost is given by the same expression as before except for an additive constant that depends on $E\{w_0^2\}$ and $E\{w_1^2\}$.

Generally, if the optimal policy is unaffected when the disturbances are replaced by their means, we say that *certainty equivalence* holds. This occurs in several types of problems involving a linear system and a quadratic cost; see [Ber17], Sections 3.1 and 4.2. For other problems, certainty equivalence can be used as a basis for problem approximation, e.g., assume that certainty equivalence holds (i.e., replace stochastic quantities by some typical values, such as their expected values) and apply exact DP to the resulting deterministic optimal control problem (see Section 2.3.2).

1.4 REINFORCEMENT LEARNING AND OPTIMAL CONTROL - SOME TERMINOLOGY

There has been intense interest in DP-related approximations in view of their promise to deal with the *curse of dimensionality* (the explosion of the computation as the number of states increases is dealt with the use of approximate cost functions) and the *curse of modeling* (a simulator/computer model may be used in place of a mathematical model of the problem). The current state of the subject owes much to an enormously beneficial cross-fertilization of ideas from optimal control (with its traditional emphasis on sequential decision making and formal optimization methodologies), and from artificial intelligence (and its traditional emphasis on learning through

observation and experience, heuristic evaluation functions in game-playing programs, and the use of feature-based and other representations).

The boundaries between these two fields are now diminished thanks to a deeper understanding of the foundational issues, and the associated methods and core applications. Unfortunately, however, there have been substantial differences in language and emphasis in RL-based discussions (where artificial intelligence-related terminology is used) and DP-based discussions (where optimal control-related terminology is used). This includes the typical use of maximization/value function/reward in the former field and the use of minimization/cost function/cost per stage in the latter field, and goes much further.

The terminology used in this book is standard in DP and optimal control, and in an effort to forestall confusion of readers that are accustomed to either the RL or the optimal control terminology, we provide a list of selected terms commonly used in RL, and their optimal control counterparts.

(a) **Agent** = Decision maker or controller.

(b) **Action** = Decision or control.

(c) **Environment** = System.

(d) **Reward of a stage** = (Opposite of) Cost of a stage.

(e) **State value** = (Opposite of) Cost starting from a state.

(f) **Value (or reward, or state-value) function** = (Opposite of) Cost function.

(g) **Maximizing the value function** = Minimizing the cost function.

(h) **Action (or state-action) value** = Q-factor (or Q-value) of a state-control pair.

(i) **Planning** = Solving a DP problem with a known mathematical model.

(j) **Learning** = Solving a DP problem in model-free fashion.

(k) **Self-learning** (or self-play in the context of games) = Solving a DP problem using policy iteration.

(l) **Deep reinforcement learning** = Approximate DP using value and/or policy approximation with deep neural networks.

(m) **Prediction** = Policy evaluation.

(n) **Generalized policy iteration** = Optimistic policy iteration.

(o) **State abstraction** = Aggregation.

(p) **Learning a model** = System identification.

(q) **Episodic task or episode** = Finite-step system trajectory.

(r) **Continuing task** = Infinite-step system trajectory.

(s) **Backup** = Applying the DP operator at some state.

(t) **Sweep** = Applying the DP operator at all states.

(u) **Greedy policy with respect to a cost function** J = Minimizing policy in the DP expression defined by J.

(v) **Afterstate** = Post-decision state.

Some of the preceding terms will be introduced in future chapters. The reader may then wish to return to this section as an aid in connecting with the relevant RL literature.

Notation

Unfortunately the confusion arising from different terminology has been exacerbated by the use of different notations. The present textbook follows Bellman's notation [Bel67], and more generally the "standard" notation of the Bellman/Pontryagin optimal control era. This notation is consistent with the author's other DP books.

A summary of our most prominently used symbols is as follows:

(a) x: state.

(b) u: control.

(c) J: cost function.

(d) g: cost per stage.

(e) f: system function.

(f) i: discrete state.

(g) $p_{ij}(u)$: transition probability from state i to state j under control u.

The x-u-J notation is standard in optimal control textbooks (e.g., the classical books by Athans and Falb [AtF66], and Bryson and Ho [BrH75], as well as the more recent book by Liberzon [Lib11]). The notations f and g are also used most commonly in the literature of the early optimal control period as well as later. The discrete system notations i and $p_{ij}(u)$ are very common in the discrete-state Markov decision problem and operations research literature, where discrete state problems have been treated extensively.

1.5 NOTES AND SOURCES

Our discussion of exact DP in this chapter has been brief since our focus in this book will be on approximate DP and RL. The author's DP textbooks

[Ber12], [Ber17] provide an extensive discussion of exact DP and its applications to discrete and continuous spaces problems. The mathematical aspects of exact DP are discussed in the monograph by Bertsekas and Shreve [BeS78], particularly the probabilistic/measure-theoretic issues associated with stochastic optimal control. The author's abstract DP monograph [Ber18a] aims at a unified development of the core theory and algorithms of total cost sequential decision problems, and addresses simultaneously stochastic, minimax, game, risk-sensitive, and other DP problems, through the use of abstract DP operators. The book by Puterman [Put94] provides a detailed treatment of finite-state Markovian decision problems, and particularly the infinite horizon case.

The approximate DP and RL literature has expanded tremendously since the connections between DP and RL became apparent in the late 80s and early 90s. We will restrict ourselves to mentioning textbooks, research monographs, and broad surveys, which supplement our discussions and collectively provide a guide to the literature. The author wishes to apologize in advance for the many omissions of important research references.

Two books were written on our subject in the 1990s, setting the tone for subsequent developments in the field. One in 1996 by Bertsekas and Tsitsiklis [BeT96], which reflects a decision, control, and optimization viewpoint, and another in 1998 by Sutton and Barto, which reflects an artificial intelligence viewpoint (a 2nd edition, [SuB18], was published in 2018). We refer to the former book and also to the author's DP textbooks [Ber12], [Ber17] for a broader discussion of some of the topics of the present book, including algorithmic convergence issues and more general DP models.

More recent books are the 2003 book by Gosavi (a much expanded 2nd edition [Gos15] appeared in 2015), which emphasizes simulation-based optimization and RL algorithms, Cao [Cao07], which emphasizes a sensitivity approach to simulation-based methods, Chang, Fu, Hu, and Marcus [CFH07], which emphasizes finite-horizon/limited lookahead schemes and adaptive sampling, Busoniu et. al. [BBD10], which focuses on function approximation methods for continuous space systems and includes a discussion of random search methods, Powell [Pow11], which emphasizes resource allocation and operations research applications, Vrabie, Vamvoudakis, and Lewis [VVL13], which discusses neural network-based methods, on-line adaptive control, and continuous-time optimal control applications, and Liu et al. [LWW17], which explores the interface between reinforcement learning and optimal control. The book by Haykin [Hay08] discusses approximate DP in the broader context of neural network-related subjects. The book by Borkar [Bor08] is an advanced monograph that addresses rigorously many of the convergence issues of iterative stochastic algorithms in approximate DP, mainly using the so called ODE approach. The book by Meyn [Mey07] is broader in its coverage, but touches upon some of the approximate DP algorithms that we discuss.

Influential early surveys were written, from an artificial intelligence

viewpoint, by Barto, Bradtke, and Singh [BBS95] (which dealt with the methodologies of real-time DP and its antecedent, real-time heuristic search [Kor90], and the use of asynchronous DP ideas [Ber82], [Ber83], [BeT89] within their context), and by Kaelbling, Littman, and Moore [KLM96] (which focused on general principles of reinforcement learning).

Several survey papers in the volumes by Si, Barto, Powell, and Wunsch [SBP04], and Lewis and Liu [LeL12], and the special issue by Lewis, Liu, and Lendaris [LLL08] describe approximation methodology that we will not be covering in this book: linear programming-based approaches (De Farias [DeF04]), large-scale resource allocation methods (Powell and Van Roy [PoV04]), and deterministic optimal control approaches (Ferrari and Stengel [FeS04], and Si, Yang, and Liu [SYL04]). The volume by White and Sofge [WhS92] contains several surveys that describe early work in the field.

More recent surveys and short monographs are Borkar [Bor09] (a methodological point of view that explores connections with other Monte Carlo schemes), Lewis and Vrabie [LeV09] (a control theory point of view), Werbos [Web09] (which reviews potential connections between brain intelligence, neural networks, and DP), Szepesvari [Sze10] (which provides a detailed description of approximation in value space from a RL point of view), Deisenroth, Neumann, and Peters [DNP11], and Grondman et al. [GBL12] (which focus on policy gradient methods), Browne et al. [BPW12] (which focuses on Monte Carlo Tree Search), Mausam and Kolobov [MaK12] (which deals with Markovian decision problems from an artificial intelligence viewpoint), Schmidhuber [Sch15], Arulkumaran et al. [ADB17], and Li [Li17] (which deal with reinforcement learning schemes that are based on the use of deep neural networks), the author's [Ber05a] (which focuses on rollout algorithms and model predictive control), [Ber11a] (which focuses on approximate policy iteration), and [Ber18b] (which focuses on aggregation methods), and Recht [Rec18] (which focuses on continuous spaces optimal control). The blogs and video lectures by A. Rahimi and B. Recht provide interesting views on the current state of the art of machine learning, reinforcement learning, and their relation to optimization and optimal control.

Reinforcement Learning and Optimal Control

by

Dimitri P. Bertsekas

Massachusetts Institute of Technology

Chapter 2
Approximation in Value Space

DRAFT

This is Chapter 2 of the draft textbook "Reinforcement Learning and Optimal Control." The chapter represents "work in progress," and it will be periodically updated. It more than likely contains errors (hopefully not serious ones). Furthermore, its references to the literature are incomplete. Your comments and suggestions to the author at dimitrib@mit.edu are welcome. The date of last revision is given below.

The date of last revision is given below. (A "revision" is any version of the chapter that involves the addition or the deletion of at least one paragraph or mathematically significant equation.)

January 22, 2019

2

Approximation in Value Space

Contents

2.1. General Issues of Approximation in Value Space . . . . p. 5
2.1.1. Methods for Computing Approximations in Value Space . . . . . . p. 6
2.1.2. Off-Line and On-Line Methods . . . . p. 7
2.1.3. Model-Based Simplification of the Lookahead Minimization . . . . p. 8
2.1.4. Model-Free Q-Factor Approximation in Value Space p. 9
2.1.5. Approximation in Policy Space on Top of Approximation in Value Space . . . . p. 12
2.1.6. When is Approximation in Value Space Effective? . p. 13
2.2. Multistep Lookahead . . . . p. 14
2.2.1. Multistep Lookahead and Rolling Horizon . . . . p. 16
2.2.2. Multistep Lookahead and Deterministic Problems . p. 17
2.3. Problem Approximation . . . . p. 19
2.3.1. Enforced Decomposition . . . . p. 19
2.3.2. Probabilistic Approximation - Certainty Equivalent Control . . . . p. 26
2.4. Rollout . . . . p. 32
2.4.1. On-Line Rollout for Deterministic Finite-State Problems . . . . p. 33
2.4.2. Stochastic Rollout and Monte Carlo Tree Search . p. 42
2.5. On-Line Rollout for Deterministic Infinite-Spaces Problems - Optimization Heuristics . . . . p. 53
2.5.1. Model Predictive Control . . . . p. 53
2.5.2. Target Tubes and the Constrained Controllability Condition . . . . p. 60
2.5.3. Variants of Model Predictive Control . . . . . . p. 63
2.6. Notes and Sources . . . . p. 66

As we noted in Chapter 1, the exact solution of optimal control problems by DP is often impossible. To a great extent, the reason lies in what Bellman has called the "curse of dimensionality." This refers to a rapid increase of the required computation and memory storage as the problem's size increases. Moreover, there are many circumstances where the structure of the given problem is known well in advance, but some of the problem data, such as various system parameters, may be unknown until shortly before control is needed, thus seriously constraining the amount of time available for the DP computation. These difficulties motivate suboptimal control schemes that strike a reasonable balance between convenient implementation and adequate performance.

Approximation in Value Space

There are two general approaches for DP-based suboptimal control. The first is *approximation in value space*, where we approximate the optimal cost-to-go functions J_k^* with some other functions $\tilde{J}_k$. We then replace J_k^* in the DP equation with $\tilde{J}_k$. In particular, at state x_k, we use the control obtained from the minimization

$$\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} E\Big\{g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}\big(f_k(x_k, u_k, w_k)\big)\Big\}. \quad (2.1)$$

This defines a suboptimal policy $\{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$. There are several possibilities for selecting or computing the functions $\tilde{J}_k$, which are discussed in this chapter, and also in subsequent chapters.

Note that the expected value expression appearing in the right-hand side of Eq. (2.1) can be viewed as an approximate Q-factor

$$\tilde{Q}_k(x_k, u_k) = E\Big\{g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}\big(f_k(x_k, u_k, w_k)\big)\Big\},$$

and the minimization in Eq. (2.1) can be written as

$$\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k),$$

(cf. Section 1.2). This also suggests a variant of approximation in value space, which is based on using Q-factor approximations that may be obtained directly, i.e., without the intermediate step of obtaining the cost function approximations $\tilde{J}_k$. In what follows in this chapter, we will focus on cost function approximation, but we will occasionally digress to discuss direct Q-factor approximation.

Approximation in value space based on the minimization (2.1) is commonly referred to as *one-step lookahead*, because the future costs are approximated by $\tilde{J}_{k+1}$, after a single step. An important variation is *multistep lookahead*, whereby we minimize over $\ell > 1$ stages with the future costs

approximated by a function $\tilde{J}_{k+\ell}$. For example, in two-step lookahead the function $\tilde{J}_{k+1}$ is given by

$$\tilde{J}_{k+1}(x_{k+1}) = \min_{u_{k+1}\in U_{k+1}(x_{k+1})} E\Big\{g_{k+1}(x_{k+1},u_{k+1},w_{k+1}) + \tilde{J}_{k+2}\big(f_{k+1}(x_{k+1},u_{k+1},w_{k+1})\big)\Big\},$$

where $\tilde{J}_{k+2}$ is some approximation of the optimal cost-to-go function J^*_{k+2}.

Actually, as the preceding two-step lookahead case illustrates, one may view ℓ-step lookahead as the special case of one-step lookahead where the lookahead function is the optimal cost function of an $(\ell-1)$-stage DP problem with a terminal cost at the end of the $\ell-1$ stages, which is equal to $\tilde{J}_{k+\ell}$. However, it is often important to discuss ℓ-step lookahead separately, in order to address special implementation issues that do not arise in the context of one-step lookahead.

In our initial discussion of approximation in value space of Section 2.1, we will focus on one-step lookahead. There are straightforward extensions of the main ideas to the multistep context, which we will discuss in Section 2.2.

Approximation in Policy Space

The major alternative to approximation in value space is *approximation in policy space*, whereby we select the policy by using optimization over a suitably restricted class of policies, usually a parametric family of some form. An important advantage of this approach is that the computation of controls during on-line operation of the system is often much easier compared with the minimization (2.1). However, this advantage can also be gained by combining approximation in value space with policy approximation in a two-stage scheme:

(a) Obtain the approximately optimal cost-to-go functions $\tilde{J}_k$, thereby defining a corresponding suboptimal policy $\tilde{\mu}_k$, $k=0,\ldots,N-1$, via the one-step lookahead minimization (2.1).

(b) Approximate $\tilde{\mu}_k$, $k=0,\ldots,N-1$, using a training set consisting of a large number q of sample pairs (x_k^s,u_k^s), $s=1,\ldots,q$, where $u_k^s=\tilde{\mu}_k(x_k^s)$. For example, introduce a parametric family of policies $\mu_k(x_k,r_k)$, $k=0,\ldots,N-1$, of some form, where r_k is a parameter, such as a family represented by a neural net, and then estimate the parameters r_k using some form of regression, e.g.,

$$r_k \in \arg\min_r \sum_{s=1}^{q} \big\|u_k^s - \mu_k(x_k^s,r)\big\|^2.$$

In this chapter we discuss primarily approximation in value space, although some of the ideas are also relevant to approximation in policy space.

We focus on finite horizon problems, postponing the discussion of infinite horizon problems for Chapter 4 and later. However, the finite horizon ideas are relevant to the infinite horizon setting, and many of the methods of the present chapter and Chapter 3 also apply with small modifications to infinite horizon problems.

Model-Based Versus Model-Free Implementation

An important attribute of the solution process is whether an analytical model or Monte Carlo simulation is used to compute expected values such as those arising in one-step and multistep lookahead expressions. We distinguish between two cases:

(a) In the *model-based* case, we assume that the conditional probability distribution of w_k, given (x_k, u_k), is available in essentially closed form. By this we mean that the value of $p_k(w_k \mid x_k, u_k)$ is available for any triplet (x_k, u_k, w_k). Moreover, the functions g_k and f_k are also available. In a model-based solution process, expected values, such as the one in the lookahead expression of Eq. (2.1), are obtained with algebraic calculations as opposed to Monte Carlo simulation.

(b) In the *model-free* case, the calculation of the expected value in the expression of Eq. (2.1), and other related expressions, is done with Monte Carlo simulation. There may be two reasons for this.

   (1) A mathematical model of the probabilities $p_k(w_k \mid x_k, u_k)$ is not available, but instead there is a computer program/simulator that for any given state x_k and control u_k, simulates sample probabilistic transitions to a successor state x_{k+1}, and generates the corresponding transition costs. In this case the expected value can be computed approximately by Monte Carlo simulation.†

   (2) The probabilities $p_k(w_k \mid x_k, u_k)$ are available for any triplet (x_k, u_k, w_k), but for reasons of computational efficiency we prefer to compute the expected value in the expression (2.1) by using sampling and Monte Carlo simulation. Thus the expected value is computed as in the case where there is no mathematical model, but instead there is a computer simulator.‡

---

† The term Monte Carlo simulation mostly refers to the use of a software simulator. However, a hardware or a combined hardware/software simulator may also be used in some practical situations to generate samples that are used for Monte Carlo averaging.

‡ The idea of using Monte Carlo simulation to compute complicated integrals or even sums of many numbers is used widely in various types of numerical computations. It encompasses efficient Monte Carlo techniques known as *Monte Carlo*

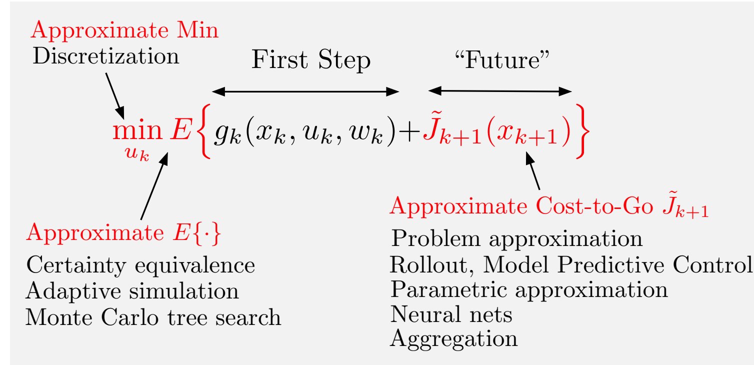

Figure 2.1.1 Schematic illustration of various options for approximation in value space with one-step lookahead. The lookahead function values $\tilde J_{k+1}(x_{k+1})$ approximate the optimal cost-to-go values $J^*_{k+1}(x_{k+1})$, and can be computed by a variety of methods. There may be additional approximations in the minimization over u_k and in the computation of the expected value over w_k; see Section 2.1.1.

In summary, *the use of sampling and Monte Carlo simulation is the defining attribute for a method to be model-based or model-free in the terminology of this book*. In particular, a model-free method is one that can be applied both when there is a mathematical model and when there is not. This view of a model-free approach aims to avoid ambiguities in cases where a model is available, but a model-free method is used anyway for reasons of convenience or computational efficiency.

Note that for deterministic problems there is no expected value to compute, so these problems typically come under the model-based category, even if values of the functions g_k and f_k become available through complicated computer calculations. Still however, Monte Carlo simulation may enter the solution process of a deterministic problem for a variety of reasons. For example the games of chess and Go are perfectly deterministic, but the AlphaGo and AlphaZero programs (Silver et al. [SHM16], [SHS17]) use randomized policies and rely heavily on Monte Carlo tree search techniques, which will be discussed later in Section 2.4. The same is true for some policy gradient methods, which will be discussed in Chapter 4.

2.1 GENERAL ISSUES OF APPROXIMATION IN VALUE SPACE

There are two major issues in a value space approximation scheme, and each of the two can be considered separately from the other:

(1) *Obtaining* $\tilde J_k$, i.e., the method to compute the lookahead functions $\tilde J_k$ that are involved in the lookahead minimization (2.1). There are

---

integration and *importance sampling*; see textbooks such as [Liu01], [AsG10], [RoC10] for detailed developments.

quite a few approaches here (see Fig. 2.1.1). Several of them are discussed in this chapter, and more will be discussed in subsequent chapters.

(2) *Control selection*, i.e., the method to perform the minimization (2.1) and implement the suboptimal policy $\tilde{\mu}_k$. Again there are several exact and approximate methods for control selection, some of which will be discussed in this chapter (see Fig. 2.1.1).

In this section we will provide a high level discussion of these issues.

2.1.1 Methods for Computing Approximations in Value Space

Regarding the computation of $\tilde{J}_k$, we will consider four types of methods:

(a) *Problem approximation (Section 2.3)*: Here the functions $\tilde{J}_k$ in Eq. (2.1) are obtained as the optimal or nearly optimal cost functions of a simplified optimization problem, which is more convenient for computation. Simplifications may include, exploiting decomposable structure, ignoring various types of uncertainties, and reducing the size of the state space. The latter form of simplification is known as *aggregation*, and is discussed separately in Chapter 5.

(b) *On-line approximate optimization (Section 2.4)*: These methods often involve the use of a suboptimal policy or heuristic, which is applied on-line when needed to approximate the true optimal cost-to-go values. The suboptimal policy may be obtained by any other method, e.g., problem approximation. *Rollout algorithms* and *model predictive control* are prime examples of these methods.

(c) *Parametric cost approximation (Chapter 3)*: Here the functions $\tilde{J}_k$ in Eq. (2.1) are obtained from a given parametric class of functions $\tilde{J}_k(x_k, r_k)$, where r_k is a parameter vector, selected by a suitable algorithm. The parametric class is typically obtained by using prominent characteristics of x_k called *features*, which can be obtained either through insight into the problem at hand, or by using training data and some form of neural network.

(d) *Aggregation (Chapter 5)*: This is a special but rather sophisticated form of problem approximation. A simple example is to select a set of representative states for each stage, restrict the DP algorithm to these states only, and approximate the costs-to-go of other states by interpolation between the optimal costs-to-go of the representative states. In another example of aggregation, the state space is divided into subsets, and each subset is viewed as a state of an "aggregate DP problem." The functions $\tilde{J}_k$ are then derived from the optimal cost functions of the aggregate problem. The state space partition can be arbitrary, but is often determined by using features (states with "similar" features are grouped together). Moreover, aggregation can be

combined in complementary fashion with the methods (a)-(c) above, and can use as a starting point an approximate cost-to-go function produced by any one of these methods; e.g., apply a parametric approximation method and enhance the resulting cost function through local corrections obtained by aggregation.

Additional variations of the above methods are obtained when used in combination with approximate minimization over u_k in Eq. (2.1), and also when the expected value over w_k is computed approximately via a certainty equivalence approximation (cf. Section 2.3.2) or adaptive simulation and Monte Carlo tree search (Sections 2.1.2 and 2.4.2).

2.1.2 Off-Line and On-Line Methods

In approximation in value space an important consideration is whether the cost-to-go functions $\tilde{J}_{k+1}$ and the suboptimal control functions $\tilde{\mu}_k$, $k = 0, \ldots, N-1$, of Eq. (2.1) are computed *off-line* (i.e., before the control process begins, and for all x_k and k), or *on-line* (i.e., after the control process begins, when needed, and for just the states x_k to be encountered).

Usually, for challenging problems, the controls $\tilde{\mu}_k(x_k)$ are computed on-line, since their storage may be difficult for a large state space. However, the on-line or off-line computation of $\tilde{J}_{k+1}$ is an important design choice. We thus distinguish between:

(i) *Off-line methods*, where the entire function $\tilde{J}_{k+1}$ in Eq. (2.1) is computed for every k, before the control process begins. The values $\tilde{J}_{k+1}(x_{k+1})$ are either stored in memory or can be obtained with a simple and fast computation, as needed in order to compute controls by one-step lookahead. The advantage of this is that most of the computation is done off-line, before the first control is applied at time 0. Once the control process starts, no extra computation is needed to obtain $\tilde{J}_{k+1}(x_{k+1})$ for implementing the corresponding suboptimal policy.

(ii) *On-line methods*, where most of the computation is performed just after the current state x_k becomes known, the values $\tilde{J}_k(x_{k+1})$ are computed only at the relevant next states x_{k+1}, and are used to compute the control to be applied via Eq. (2.1). These methods require the computation of a control only for the N states actually encountered in the control process. In contrast with the off-line approximation methods, these methods are well-suited for *on-line replanning*, whereby the problem data may change over time.

Examples of typically off-line schemes are neural network and other parametric approximations, as well as aggregation. Examples of typically on-line schemes are rollout and model predictive control. Schemes based on problem approximation may be either on-line or off-line depending on other

problem-related factors. Of course there are also problem-dependent hybrid methods, where significant computation is done off-line to expedite the on-line computation of needed values of $\tilde{J}_{k+1}$.

2.1.3 Model-Based Simplification of the Lookahead Minimization

We will now consider ways to facilitate the calculation of the suboptimal control $\tilde{\mu}_k(x_k)$ at state x_k via the minimization of the one-step lookahead expression

$$E\Big\{g_k(x_k,u_k,w_k)+\tilde{J}_{k+1}\big(f_k(x_k,u_k,w_k)\big)\Big\}, \tag{2.2}$$

once the cost-to-go approximating functions $\tilde{J}_{k+1}$ have been selected. In this section, we will assume that we have a mathematical model, i.e., that the functions g_k and f_k are available in essentially closed form, and that the conditional probability distribution of w_k, given (x_k,u_k), is also available. In particular, Monte Carlo simulation is not used to compute the expected value in Eq. (2.2). We will address the model-free case in the next section.

Important issues here are the computation of the expected value (if the problem is stochastic) and the minimization over $u_k\in U_k(x_k)$ in Eq. (2.2). Both of these operations may involve substantial work, which is of particular concern when the minimization is to be performed on-line.

One possibility to eliminate the expected value from the expression (2.2) is (assumed) *certainty equivalence*. Here we choose a typical value $\tilde{w}_k$ of w_k, and use the control $\tilde{\mu}_k(x_k)$ that solves the deterministic problem

$$\min_{u_k\in U_k(x_k)}\Big[g_k(x_k,u_k,\tilde{w}_k)+\tilde{J}_{k+1}\big(f_k(x_k,u_k,\tilde{w}_k)\big)\Big]. \tag{2.3}$$

The approach of turning a stochastic problem into a deterministic one by replacing uncertain quantities with single typical values highlights the possibility that $\tilde{J}_k$ may itself be obtained by using deterministic methods. We will discuss this approach and its variations in greater detail later in this chapter (see Section 2.3).

Let us now consider the issue of algorithmic minimization over $U_k(x_k)$ in Eqs. (2.2) and (2.3). If $U_k(x_k)$ is a finite set, the minimization can be done by brute force, through exhaustive computation and comparison of the relevant cost expressions. This of course can be very time consuming, particularly for multistep lookahead, but parallel computation can be used with great effect for this purpose [as well as for the calculation of the expected value in the expression (2.2)]. For some discrete control problems, *integer programming* techniques may also be used. Moreover, for deterministic problems with multistep lookahead, sophisticated exact or approximate *shortest path methods* may be considered; several methods of this type are available, such as label correcting methods, A^* methods, and their variants (see the author's books [Ber98] and [Ber17] for detailed accounts, which are consistent with the context of this chapter).

When the control constraint set is infinite, it may be replaced by a finite set through discretization. However, a more efficient alternative may be to use continuous space *nonlinear programming* techniques. This possibility can be attractive for deterministic problems, which lend themselves better to continuous space optimization; an example is the model predictive control context (see Section 2.5).

For stochastic problems and either one-step or multistep lookahead and continuous control spaces, the methodology of *stochastic programming*, which bears a close connection with linear and nonlinear programming methods, may be useful. We refer to the textbook [Ber17] for a discussion of its application to the approximate DP context, and references to the relevant literature. Still another possibility to simplify the one-step lookahead minimization (2.2) is based on Q-factor approximation, which is also suitable for model-free policy implementation, as we discuss next.

2.1.4 Model-Free Q-Factor Approximation in Value Space

One of the major aims of this book is to address stochastic *model-free* situations, i.e., methods where a mathematical model [the system functions f_k, the probability distribution of w_k, and the one-stage cost functions g_k] is not used because it is either hard to construct, or simply inconvenient. We assume instead that the system and cost structure can be simulated in software far more easily (think, for example, control of a queueing network with complicated but well-defined service disciplines at the queues).†

In this section, we will review some of the high-level ideas of passing from model-based to model-free policy implementations for stochastic problems. In particular, we assume that:

(a) There is a computer program/simulator that for any given state x_k and control u_k, simulates sample probabilistic transitions to a successor state x_{k+1}, and generates the corresponding transition costs.

(b) A cost function approximation $\tilde{J}_{k+1}$ is available. Approaches to obtain $\tilde{J}_{k+1}$ in model-free fashion will be discussed in the context of specific methods later. For example $\tilde{J}_{k+1}$ may be obtained by solving a simpler problem for which a model is available, or it may be separately obtained without a mathematical model, by using a simulator.

We want to use the functions $\tilde{J}_{k+1}$ and the simulator to compute or approximate the Q-factors

$$E\Big\{g_k(x_k,u_k,w_k)+\tilde{J}_{k+1}\big(f_k(x_k,u_k,w_k)\big)\Big\}, \tag{2.4}$$

for all $u_k \in U_k(x_k)$, and then find the minimal Q-factor and corresponding one-step lookahead control.

† Another possibility is to use the real system to provide the next state and transition cost, but we will not deal explicitly with this case in this book.

Given a state x_k, we may use the simulator to compute these Q-factors for all the pairs (x_k, u_k), $u_k \in U_k(x_k)$, and then select the minimizing control. However, in many cases this can be very time-consuming. To deal with this difficulty, we may introduce a parametric family/approximation architecture of Q-factor functions, $\tilde{Q}_k(x_k, u_k, r_k)$, where r_k is the parameter vector and use a least squares fit/regression to approximate the expected value that is minimized in Eq. (2.2). One possibility is to use a neural network parametric architecture; see Chapter 3, where we will discuss methods for selecting and training parametric architectures. The steps are as follows:

Summary of Q-Factor Approximation Based on Approximation in Value Space

Assume that the value of $\tilde{J}_{k+1}(x_{k+1})$ is available for any given x_{k+1}:

(a) Use the simulator to collect a large number of "representative" sample state-control-successor states-stage cost quadruplets

$$(x_k^s, u_k^s, x_{k+1}^s, g_k^s),$$

and corresponding sample Q-factors

$$\beta_k^s = g_k^s + \tilde{J}_{k+1}(x_{k+1}^s), \qquad s = 1, \ldots, q. \tag{2.5}$$

Here x_{k+1}^s is the simulator's output of the next state

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

that corresponds to some disturbance w_k^s. This disturbance also determines the one-stage-cost sample

$$g_k^s = g_k(x_k^s, u_k^s, w_k^s).$$

The simulator need not output w_k^s; only the sample next state x_{k+1}^s and sample cost g_k^s are needed (see Fig. 2.1.2).

(b) Determine the parameter vector $\bar{r}_k$ by the least-squares regression

$$\bar{r}_k \in \arg\min_{r_k} \sum_{s=1}^{q} \big(\tilde{Q}_k(x_k^s, u_k^s, r_k) - \beta_k^s\big)^2. \tag{2.6}$$

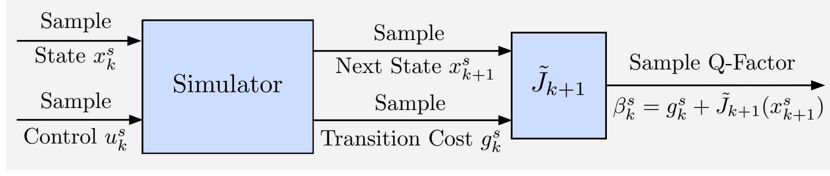

Figure 2.1.2 Schematic illustration of the simulator used for a model-free Q-factor approximation, assuming approximate cost functions $\tilde{J}_{k+1}$ are known. The input to the simulator are sample state-control pairs (x_k^s, u_k^s), and the outputs are a next state sample x_{k+1}^s and cost sample g_k^s. These correspond to a disturbance w_k^s according to

$$x_{k+1}^s = f_k(x_k^s, u_k^s, w_k^s), \qquad g_k^s = g_k(x_k^s, u_k^s, w_k^s).$$

The actual value of w_k^s need not be output by the simulator. The sample Q-factors β_k^s are generated according to Eq. (2.5), and are used in the least squares regression (2.6) to yield a parametric Q-factor approximation $\tilde{Q}_k$ and the policy implementation (2.7).

(c) Use the policy

$$\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k, \bar{r}_k). \tag{2.7}$$

Note some important points about the preceding procedure:

(1) It is model-free in the sense that it is based on Monte Carlo simulation. Moreover, it does not need the functions f_k and g_k, and the probability distribution of w_k to generate the policy $\tilde{\mu}_k$ through the least squares regression (2.6) and the Q-factor minimization (2.7). The simulator to collect the samples (2.5) and the cost function approximation $\tilde{J}_{k+1}$ suffice.

(2) Two approximations are potentially required: One to compute $\tilde{J}_{k+1}$, which is needed for the samples β_k^s [cf. Eq. (2.5)], and another to compute $\tilde{Q}_k$ through the regression (2.6). The approximation methods to obtain $\tilde{J}_{k+1}$ and $\tilde{Q}_k$ may be unrelated.

(3) The policy $\tilde{\mu}_k$ obtained through the minimization (2.7) is not the same as the one obtained through the minimization (2.2). There are two reasons for this. One is the approximation error introduced by the Q-factor architecture $\tilde{Q}_k$, and the other is the simulation error introduced by the finite-sample regression (2.6). We have to accept these sources of error as the price to pay for the convenience of not requiring a mathematical model for policy implementation.

Let us also mention a variant of the least squares minimization in Eq. (2.6), which is to use a *regularized minimization* where a quadratic regularization term is added to the least squares objective. This term is a multiple of the squared deviation $\|r - \hat{r}\|^2$ of r from some initial guess $\hat{r}$. Moreover, in some cases, a nonquadratic minimization may be used in place of Eq. (2.6) to determine $\bar{r}_k$, but in this book we will focus on least squares exclusively.

2.1.5 Approximation in Policy Space on Top of Approximation in Value Space

A common approach for approximation in policy space, is to introduce a parametric family of policies $\tilde{\mu}_k(x_k, r_k)$, where r_k is a parameter vector. The parametrization may involve a neural network as we will discuss in Chapter 3. Alternatively, the parametrization may involve problem-specific features, exploiting the special structure of the problem at hand.

A general scheme for parametric approximation in policy space is to obtain a large number of sample state-control pairs (x_k^s, u_k^s), $s = 1, \ldots, q$, such that for each s, u_k^s is a "good" control at state x_k^s. We can then choose the parameter r_k by solving the least squares/regression problem

$$\min_{r_k} \sum_{s=1}^{q} \big\| u_k^s - \tilde{\mu}_k(x_k^s, r_k) \big\|^2 \tag{2.8}$$

(possibly with added regularization). In particular, we may determine u_k^s using a human or a software "expert" that can choose "near-optimal" controls at given states, so $\tilde{\mu}_k$ is trained to match the behavior of the expert. Methods of this type are commonly referred to as *supervised learning* in artificial intelligence (see also the discussion in Section 4.11).

A special case of the above procedure, which connects with approximation in value space, is to generate the sample state-control pairs (x_k^s, u_k^s) through a one-step lookahead minimization of the form

$$u_k^s \in \arg \min_{u \in U_k(x_k)} E\Big\{ g_k(x_k^s, u, w_k) + \tilde{J}_{k+1}\big(f_k(x_k^s, u, w_k)\big) \Big\}, \tag{2.9}$$

where $\tilde{J}_{k+1}$ is a suitable (separately obtained) approximation in value space; cf. Eq. (2.2), or an approximate Q-factor based minimization

$$u_k^s \in \arg \min_{u_k \in U_k(x_k^s)} \tilde{Q}_k(x_k^s, u_k, \bar{r}_k), \tag{2.10}$$

[cf. Eq. (2.7)]. In this case, we collect the sample state-control pairs (x_k^s, u_k^s), $s = 1, \ldots, q$, by using approximation in value space through Eq. (2.9) or Eq. (2.10), and then apply approximation in policy space through

Eq. (2.8) (i.e., approximation in policy space is built on top of approximation in value space).

A major advantage of schemes based on the minimization (2.8) is that once the parametrized policy is obtained, the on-line implementation of the policy is fast and does not involve extensive calculations such as minimizations of the form (2.9) or (2.10). This advantage is generally shared by schemes that are based on approximation in policy space.

2.1.6 When is Approximation in Value Space Effective?

An important question is what constitutes good approximating functions $\tilde{J}_k$ in a one-step lookahead scheme. An answer that suggests itself is that $\tilde{J}_k$ should be "close" to the optimal cost-to-go function J_k^* for all k. This guarantees a certain degree of quality of the approximation scheme, but is neither necessary nor is it satisfied by all or even most good practical schemes.

For example if the approximating values $\tilde{J}_k(x_k)$ differ from the optimal values $J_k^*(x_k)$ uniformly by the same constant, the policy obtained by the approximation in value space scheme is optimal. This suggests that a better condition might be that relative values of $\tilde{J}_k$ and J_k^* should be "close" to each other, i.e.,

$$\tilde{J}_k(x_k) - \tilde{J}_{k+\ell}(x'_k) \approx J_k^*(x_k) - J_k^*(x'_k),$$

for all pairs of states x_k and x'_k. Still, however, this guideline neglects the role of the first stage cost (or the cost of the first ℓ stages in the case of ℓ-step lookahead).

A more accurate predictor of good quality of the suboptimal policy obtained is that the Q-factor approximation error $Q_k(x_k, u) - \tilde{Q}_k(x_k, u)$ changes gradually as u changes, where Q_k and $\tilde{Q}_k$ denote the exactly optimal Q-factor and its approximation, respectively. For a heuristic explanation, suppose that approximation in value space generates a control $\tilde{u}_k$ at a state x_k where another control u_k is optimal. Then we have

$$\tilde{Q}_k(x_k, u_k) - \tilde{Q}_k(x_k, \tilde{u}_k) \geq 0, \tag{2.11}$$

since $\tilde{u}_k$ minimizes $\tilde{Q}_k(x_k, \cdot)$, and

$$Q_k(x_k, \tilde{u}_k) - Q_k(x_k, u_k) \geq 0, \tag{2.12}$$

since u_k minimizes $Q_k(x_k, \cdot)$. If $\tilde{u}_k$ is far from optimal, the Q-factor difference in Eq. (2.12) will be large, and by adding Eq. (2.11), it follows that the expression

$$\big(Q_k(x_k, \tilde{u}_k) - \tilde{Q}_k(x_k, \tilde{u}_k)\big) - \big(Q_k(x_k, u_k) - \tilde{Q}_k(x_k, u_k)\big)$$

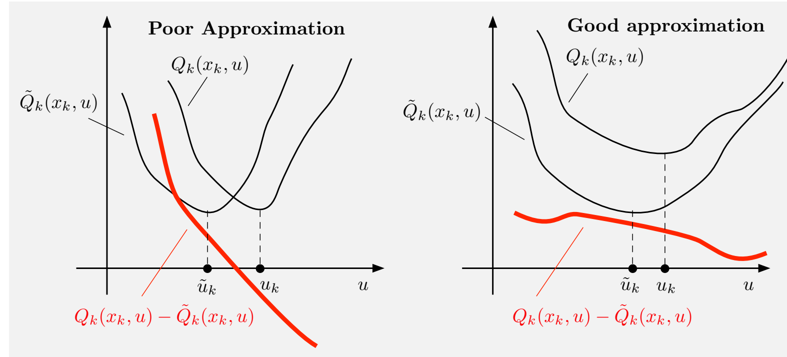

Figure 2.1.3 Schematic illustration of the "slope" of the Q-factor approximation error as a predictor of quality of an approximation in value space scheme. At a given state x_k, let u_k be optimal, so that it minimizes $Q_k(x_k,u)$ over $u \in U_k(x_k)$, and let $\tilde{u}_k$ be generated by approximation in value space, so that it minimizes $\tilde{Q}_k(x_k,u)$ over $u \in U_k(x_k)$. In the figure on the right the approximation error $Q_k(x_k,u) - \tilde{Q}_k(x_k,u)$ changes gradually (i.e., has small "slope"), and $\tilde{u}_k$ is a good choice, because $Q_k(x_k,\tilde{u}_k)$ is close the optimal $Q_k(x_k,u_k)$. In the figure on the left the approximation error $Q_k(x_k,u) - \tilde{Q}_k(x_k,u)$ changes fast (i.e., has large "slope"), and $\tilde{u}_k$ is a poor choice. In the extreme case where the Q-factors $Q_k(x_k,u)$ and $\tilde{Q}_k(x_k,u)$ differ by a constant, minimization of either one of them yields the same result.

will be even larger. This is not likely to happen if the approximation error $Q_k(x_k,u) - \tilde{Q}_k(x_k,u)$ changes gradually (i.e., has small "slope") for u in a neighborhood that includes u_k and $\tilde{u}_k$ (cf. Fig. 2.1.3). In many practical settings, as u changes, the corresponding changes in the approximate Q-factors $\tilde{Q}_k(x_k,u)$ tend to have "similar" form to the changes in the exact Q-factors $Q_k(x_k,u)$, thus providing some explanation for the observed success of approximation in value space in practice.

Of course, one would like to have quantitative tests to check the quality of either the approximate cost functions $\tilde{J}_k$ and Q-factors $\tilde{Q}_k$, or the suboptimal policies obtained. However, general tests of this type are not available, and it is often hard to assess how a particular suboptimal policy compares to the optimal, except on a heuristic, problem-dependent basis. Unfortunately, this is a recurring difficulty in approximate DP/RL.

2.2 MULTISTEP LOOKAHEAD

The approximation in value space scheme that we have discussed so far is known as one-step lookahead, since it involves solving a one-step minimization problem at each time k [cf. Eq. (2.1)]. A more ambitious, but also computationally more intensive scheme is *multistep lookahead*.

As an example, in *two-step lookahead* we apply at time k and state x_k, the control $\tilde{\mu}_k(x_k)$ attaining the minimum in Eq. (2.1), where now

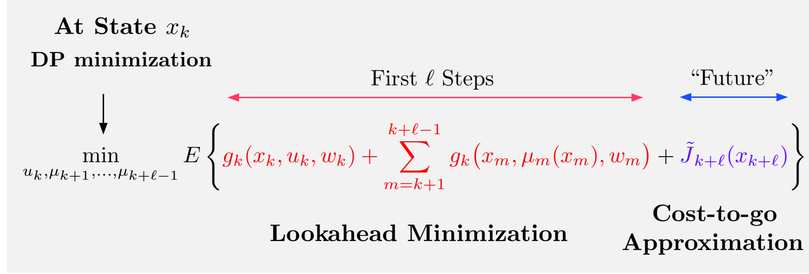

Figure 2.2.1 Illustration of approximation in value space with ℓ-step lookahead.

$\tilde J_{k+1}$ is obtained itself on the basis of a one-step lookahead approximation. In other words, for all possible states x_{k+1} that can be generated via the system equation starting from x_k,

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

we have

$$\tilde J_{k+1}(x_{k+1}) = \min_{u_{k+1}\in U_{k+1}(x_{k+1})} E\Big\{ g_{k+1}(x_{k+1}, u_{k+1}, w_{k+1}) + \tilde J_{k+2}\big(f_{k+1}(x_{k+1}, u_{k+1}, w_{k+1})\big)\Big\},$$

where $\tilde J_{k+2}$ is some approximation of the optimal cost-to-go function J^*_{k+2}.

Thus two-step lookahead amounts to *solving a two-stage version of the DP problem with x_k as the initial state and $\tilde J_{k+2}$ as the terminal cost function*. Given x_k, the solution of this DP problem yields a two-stage policy that consists of the single control u_k for the first lookahead stage, plus a control $\mu_{k+1}(x_{k+1})$ for each value of $x_{k+1} = f_k(x_k, u_k, w_k)$ that can occur at the second lookahead stage $k+1$. However, once this two-stage policy is computed, the controls $\mu_{k+1}(x_{k+1})$ are discarded, and only u_k is used as the control applied by the two-step lookahead policy at x_k. At the next stage, this process is repeated, i.e., we solve a two-stage DP problem with x_{k+1} as the initial state and $\tilde J_{k+3}$ as the terminal cost function.

Policies with lookahead of $\ell > 2$ stages are similarly defined: at state x_k, we solve an ℓ-stage version of the DP problem with x_k as the initial state and $\tilde J_{k+\ell}$ as the terminal cost function, and use the first control of the ℓ-stage policy thus obtained, while discarding the others; see Fig. 2.2.1. Of course, in the final stages where $k > N - \ell$, we should shorten the size of lookahead to $N - k$. Note that the simplifications in the one-step lookahead minimization discussed in Section 2.1.2 (assumed certainty equivalence, adaptive sampling, etc), and model-free policy implementation (Section 2.1.3) extend to multistep lookahead.

2.2.1 Multistep Lookahead and Rolling Horizon

There are several ways to compute the lookahead functions $\tilde{J}_{k+\ell}$ in ℓ-step lookahead, similar to the one-step lookahead case. However, there is also another possibility: with sufficiently long lookahead, we may capture enough of the character of the DP problem at hand so that a sophisticated choice of $\tilde{J}_{k+\ell}$ may not be needed.

In particular, we may set $\tilde{J}_{k+\ell}(x_{k+\ell}) \equiv 0$, or set

$$\tilde{J}_{k+\ell}(x_{k+\ell}) = g_N(x_{k+\ell}).$$

The idea is to use a sufficiently large number of lookahead steps ℓ to ensure a reasonably faithful approximation of the optimal Q-factors Q_k or cost-to-go functions $J^*_{k+\ell}$ within a constant.† This is also referred to as the *rolling horizon approach*, but essentially it is the same as multistep lookahead with a simplified cost-to-go approximation. Note that the idea of a rolling horizon is well-suited and applies with few modifications to infinite horizon problems as well.‡

Typically, *as the size ℓ of the lookahead is chosen larger, the need for a good choice of $\tilde{J}_{k+\ell}$ tends to diminish.* The reason is that the effective cost-to-go approximation in ℓ-step lookahead consists of two components:

(a) The cost of an $(\ell - 1)$ step problem involving the last $(\ell - 1)$ stages of the ℓ-step lookahead.

(b) The terminal cost approximation $\tilde{J}_{k+\ell}$.

Since the $(\ell - 1)$-step problem is treated by exact optimization, the overall approximation will be accurate if the contribution of the terminal cost approximation is relatively insignificant. This is likely to be true with large enough ℓ.

Thus, one is tempted to conjecture that if ℓ is increased, then the performance of the lookahead policy is improved. This, however, need not be true always, essentially because beyond the next ℓ stages, the policy may be "blind" to the presence of particularly "favorable" or "unfavorable" states. The following example is an illustration.

---

† See the discussion in Section 2.1.4. Generally, rolling horizon schemes tend to work well if the probability distribution of the state $k+\ell$ steps ahead is roughly independent of the current state and control, or is concentrated around "low cost" states.

‡ For infinite horizon problems the cost-to-go approximations $\tilde{J}_k$ will typically be the same at all stages k, i.e., $\tilde{J}_k \equiv \tilde{J}$ for some $\tilde{J}$. As a result, the limited lookahead approach produces a stationary policy. In the case of discounted problems with an infinite horizon (see Chapter 4), a simple approach is to use a rolling horizon that is long enough so that the tail cost is negligible and can be replaced by zero, but it is also possible to use a small number of lookahead stages ℓ, as long as we compensate with a terminal cost function approximation $\tilde{J}$.

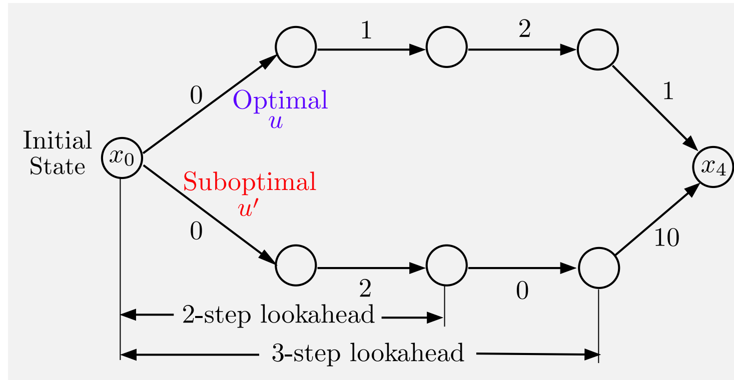

Figure 2.2.2 The 4-stage deterministic shortest problem of Example 2.2.1, illustrating how using a longer lookahead with a cost function approximation $\tilde{J}_k(x_k) \equiv 0$ may degrade the performance of the policy obtained.

Example 2.2.1

This is an oversimplified example that demonstrates a pitfall of the multistep lookahead and rolling horizon approaches with approximate cost-to-go functions

$$\tilde{J}_k(x_k) \equiv 0.$$

It may happen that with longer lookahead the quality of the suboptimal control obtained is degraded.

Consider the 4-stage deterministic shortest problem of Fig. 2.2.2. At the initial state there are two possible controls, denoted u and u'. At all other states there is only one control available, so a policy is specified by just the initial choice between controls u and u'. The costs of the four transitions on the upper and the lower path are shown next to the corresponding arcs $(0, 1, 2, 1$ for the upper path and $0, 2, 0, 10$ on the lower path). From the initial state, 2-step lookahead with terminal cost approximation $\tilde{J}_2 = 0$, compares $0 + 1$ with $0 + 2$ and prefers the optimal control u, while 3-step lookahead with terminal cost approximation $\tilde{J}_3 = 0$, compares $0 + 1 + 2$ with $0 + 2 + 0$ and prefers the suboptimal control u'. Thus using a longer lookahead yields worse performance. The problem here has to do with large cost changes at the "edge" of the lookahead (a cost of 0 just after the 2-step lookahead, followed by a cost of 10 just after the 3-step lookahead).

2.2.2 Multistep Lookahead and Deterministic Problems

Generally, the implementation of multistep lookahead can be prohibitively time-consuming for stochastic problems, because it requires at each step the solution of a stochastic DP problem with a horizon that is equal to the size of the lookahead. However, when the problem is deterministic, the lookahead problems are also deterministic, and can be solved by shortest

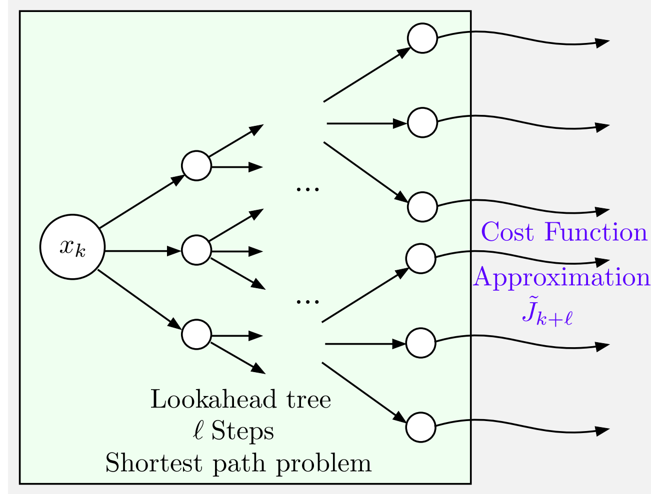

Figure 2.2.3 Multistep lookahead for a deterministic finite-state problem. The lookahead minimization is equivalent to a shortest path problem.

path methods for a finite spaces problem, or even for an infinite spaces problem after some form of discretization. This makes deterministic problems particularly good candidates for the use of long-step lookahead in conjunction with the rolling horizon approach that we discussed in the preceding section.

Similarly, for a continuous-spaces deterministic optimal control problem, the lookahead minimization may be conveniently solvable by nonlinear programming methods. This idea finds wide application in the context of model predictive control (see the discussion in Section 2.5).

Partially Deterministic Form of Multistep Lookahead

When the problem is stochastic, one may consider a hybrid, partially deterministic approach: at state x_k, *allow for a stochastic disturbance* w_k *at the current stage, but fix the future disturbances* $w_{k+1}, \ldots, w_{k+\ell-1}$*, up to the end of the lookahead horizon, to some typical values*. This allows us to bring to bear deterministic methods in the computation of approximate costs-to-go beyond the first stage.

In particular, with this approach, the needed values $\tilde{J}_{k+1}(x_{k+1})$ will be computed by solving an $\ell - 1$-step deterministic shortest path problem involving the typical values of the disturbances $w_{k+1}, \ldots, w_{k+\ell-1}$. Then the values $\tilde{J}_{k+1}(x_{k+1})$ will be used to compute the approximate Q-factors of pairs (x_k, u_k) using the formula

$$\tilde{Q}_k(x_k, u_k) = E\Big\{g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}\big(f_k(x_k, u_k, w_k)\big)\Big\},$$

which incorporates the first stage uncertainty. Finally, the control chosen by such a scheme at time k will be

$$\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k);$$

cf. Eq. (2.7).

The idea of fixing uncertain quantities to typical values for approximation purposes is generally referred to as (assumed) *certainty equivalence*, and will be discussed at length in Section 2.3.2. The idea of using multistep lookahead for deterministic problems will also be reexamined in the context of the rollout algorithm in Section 2.4.1.

2.3 PROBLEM APPROXIMATION

A key issue in implementing a limited lookahead policy is the choice of the cost-to-go approximation at the end of the lookahead. In this section, we discuss the problem approximation approach whereby we approximate the optimal cost function J_k^* with some function $\tilde{J}_k$ derived from a related but simpler problem (for example the optimal cost-to-go function of that problem). In the following subsections we consider two approaches:

(1) *Simplifying the structure of the problem through enforced decomposition*, e.g., replacing coupling constraints with simpler decoupled constraints or with Lagrange multiplier-related penalties.

(2) *Simplifying the probabilistic structure of the problem*, e.g., replacing stochastic disturbances with deterministic ones.

Another approach that can be viewed as problem approximation is *aggregation*, whereby the original problem is approximated with a related "aggregate" problem that has smaller dimension or fewer states. This problem is solved exactly to yield a cost-to-go approximation for the original problem. Aggregation is also related to the feature-based parametric approximation ideas of Chapter 3, and will be discussed in Chapter 5.

2.3.1 Enforced Decomposition

The simplification/approximation approach is often well-suited for problems involving a number of subsystems that are coupled through the system equation, or the cost function, or the control constraints, but the degree of coupling is "relatively weak." It is difficult to define precisely what constitutes "weak coupling," but in specific problem contexts, usually this type of structure is easily recognized. For such problems it is often sensible to introduce approximations by artificially decoupling the subsystems in some way, thereby creating either a simpler problem or a simpler cost calculation, where subsystems can be dealt with in isolation.

There are a number of different ways to effect enforced decomposition, and the best approach is often problem-dependent. Generally, for a deterministic problem, enforced decomposition can be applied both off-line and on-line to produce a suboptimal control sequence. For a stochastic problem, it can be applied with off-line computation of the approximate cost-to-go functions $\tilde{J}_k$ and on-line computation of the corresponding suboptimal policy. We will illustrate these two possibilities with various application contexts in what follows.

Optimization of One Subsystem at a Time

When a problem involves multiple subsystems, a potentially interesting approximation approach is to optimize one subsystem at a time. In this way the control computation at time k may become simpler.

As an example consider an N-stage deterministic problem, where the control u_k at state x_k consists of n components, $u_k = \{u_k^1, \ldots, u_k^n\}$, with u_k^i corresponding to the ith subsystem. Then to compute a cost-to-go approximation at a given state x_k, one may optimize over the control sequence of a single subsystem, while keeping the controls of the remaining subsystems at some nominal values. Thus, upon arriving at x_k, we first optimize over the control sequence $\{u_k^1, u_{k+1}^1, \ldots, u_{N-1}^1\}$ of the first subsystem, then optimize over the controls of the second subsystem, and so on, while keeping the controls of the other subsystem at the latest "optimal" values computed.

There are several possible variations; for example to make the order in which the subsystems are considered subject to optimization as well, or to repeat cycling through the subsystems multiple times, each time using the results of the latest computation as nominal values of subsystem controls. This is similar to a "coordinate descent" approach, used in other optimization contexts.

Additional variations are obtained when we use approximate minimization over u_k in Eq. (2.1), and also when the expected value over w_k is computed approximately via adaptive simulation or a certainty equivalence approximation (cf. Section 2.1.2).

Example 2.3.1 (Vehicle Routing)

Consider n vehicles that move along the arcs of a given graph. Each node of the graph has a known "value" and the first vehicle that will pass through the node will collect its value, while vehicles that pass subsequently through the node do not collect any value. This may serve as a model of a situation where there are various valuable tasks to be performed at the nodes of a transportation network, and each task can be performed at most once and by a single vehicle. We assume that each vehicle starts at a given node and after at most a given number of arc moves, it must return to some other

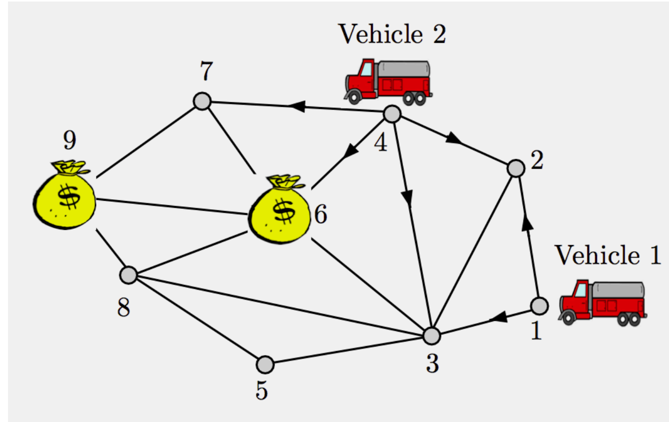

Figure 2.3.1 Schematic illustration of the vehicle routing problem and the one-vehicle-at-a-time approach. As an example, given the position pair $x_k = (1, 4)$ of the two vehicles and the current valuable tasks at positions 6 and 9, we consider moves to all possible positions pairs x_{k+1}:

$$(2, 2),\ (2, 3),\ (2, 6),\ (2, 7),\ (3, 2),\ (3, 3),\ (3, 6),\ (3, 7).$$

From each of these pairs, we first compute the best route of vehicle 1 assuming vehicle 2 does not move, and then the best route vehicle 2, taking into account the previously computed route of vehicle 1. We then select the pair x_{k+1} that results in optimal value, and move the vehicles to the corresponding positions.

given node. The problem is to find a route for each vehicle satisfying these constraints, so that the total value collected by the vehicles is maximized.

This is a difficult combinatorial problem that in principle can be approached by DP. In particular, we can view as state the n-tuple of current positions of the vehicles together with the list of nodes that have been visited by some vehicle in the past, and have thus "lost" their value. Unfortunately, the number of these states is enormous (it increases exponentially with the number of nodes and the number of vehicles). The version of the problem that involves a single vehicle, while still difficult in principle, can often be solved in reasonable time either exactly by DP or fairly accurately using a suitable heuristic. Thus a one-step lookahead policy suggests itself, with the value-to-go approximation obtained by solving single vehicle problems.

In particular, in a one-step lookahead scheme, at a given time k and from a given state x_k we consider all possible n-tuples of moves by the n vehicles. At the resulting state x_{k+1} corresponding to each n-tuple of vehicle moves, we approximate the optimal value-to-go with the value corresponding to a suboptimal set of paths. These paths are obtained as follows: we fix an order of the vehicles and we calculate a path for the first vehicle, starting at x_{k+1}, assuming the other vehicles do not move. (This is done either optimally by DP, or near optimally using some heuristic.) Then we calculate a path for the second vehicle, taking into account the value collected by the first vehicle,

and we similarly continue: for each vehicle, we calculate in the given order a path, taking into account the value collected by the preceding vehicles. We end up with a set of paths that have a certain total value associated with them. This is the value $\tilde{J}_{k+1}(x_{k+1})$ associated with the successor state x_{k+1}. We repeat with all successor states x_{k+1} corresponding to all the n-tuples of vehicle moves that are possible at x_k. We then use as suboptimal control at x_k the n-tuple of moves that yields the best value; see Fig. 2.3.1.

There are several enhancements and variations of the scheme just described. For example, we can consider multiple alternative orders for optimizing paths one-at-a-time, and choose the n-tuple of moves that corresponds to the best value obtained. Other variations may include travel costs between nodes of the graph, and constraints on how many tasks can be performed by each vehicle.

Constraint Decoupling by Constraint Relaxation

Let us now consider problems involving coupled subsystems where the coupling comes only through the control constraint. Typical cases involve the allocation of a limited resource to a set of subsystems whose system equations are completely decoupled from each other. We will illustrate with examples a few enforced decomposition approaches to deal with such situations. The first approach is *constraint relaxation*, whereby the constraint set is replaced by another constraint set that does not involve coupling.

Example 2.3.2 (Restless Multiarmed Bandit Problems)

An interesting DP model, generally referred to as the *multiarmed bandit problem*, involves n projects of which only one can be worked on at any time period. Each project i is characterized at time k by its state x_k^i. If project i is worked on at time k, one receives an expected reward $R^i(x_k^i)$, and the state x_k^i then evolves according to the equation

$$x_{k+1}^i = f^i(x_k^i, w_k^i), \qquad \text{if } i \text{ is worked on at time } k,$$

where w_k^i is a random disturbance with probability distribution depending on x_k^i but not on prior disturbances. If project i is not worked on, its state changes according to

$$x_{k+1}^i = \overline{f}^i(x_k^i, \overline{w}_k^i),$$

where $\overline{f}^i$ is a given function and $\overline{w}_k^i$ is a random disturbance with distribution depending on x_k^i but not on prior disturbances. Furthermore, a reward $\overline{R}^i(x_k^i)$ is earned, where $\overline{R}^i$ is a given function. The projects are coupled through the control constraint (only one of the projects may be worked on at any one

period). † A suboptimal enforced decomposition approach is to consider the n single project problems where a single project is worked on through the entire remaining horizon, and add the contributions of the n problems to form an optimal reward approximation.

In particular, suppose that the optimal reward function $J_k^*(x^1,\ldots,x^n)$ is approximated by a separable function of the form

$$\sum_{i=1}^{n} \tilde{J}_k^i(x^i),$$

where each $\tilde{J}_k^i$ is a function that quantifies the contribution of the ith project to the total reward. The corresponding one-step lookahead policy selects at time k the project i that maximizes

$$R^i(x^i) + \sum_{j\neq i} \overline{R}^j(x^j) + E\left\{\tilde{J}_{k+1}^i\big(f^i(x^i,w^i)\big)\right\} + \sum_{j\neq i} E\left\{\tilde{J}_{k+1}^j\big(\overline{f}^j(x^j,\overline{w}^j)\big)\right\},$$

which can also be written as

$$R^i(x^i) - \overline{R}^i(x^i) + E\left\{\tilde{J}_{k+1}^i\big(f^i(x^i,w^i)\big) - \tilde{J}_{k+1}^i\big(\overline{f}^i(x^i,\overline{w}^i)\big)\right\}$$
$$+ \sum_{j=1}^{n}\left\{\overline{R}^j(x^j) + E\big\{\tilde{J}_{k+1}^j\big(\overline{f}^j(x^j,\overline{w}^j)\big)\big\}\right\}.$$

Noting that the last term in the above expression does not depend on i, it follows that the one-step lookahead policy takes the form

$$\text{work on project } i \qquad \text{if} \qquad \tilde{m}_k^i(x^i) = \max_j \big\{\tilde{m}_k^j(x^j)\big\},$$

where for all i,

$$\tilde{m}_k^i(x^i) = R^i(x^i) - \overline{R}^i(x^i) + E\big\{\tilde{J}_{k+1}^i\big(f^i(x^i,w^i)\big) - \tilde{J}_{k+1}^i\big(\overline{f}^i(x^i,\overline{w}^i)\big)\big\}.$$

---

† In the classical and simplest version of the problem, the state of a project that is not worked on remains unchanged and produces no reward, i.e.,

$$x_{k+1}^i = x_k^i, \qquad \overline{R}^i(x_k^i) = 0, \qquad \text{if } i \text{ is not worked on at time } k.$$

This problem admits optimal policies with a nice structure that can be computationally exploited. The problem has a long history and is discussed in many sources; we refer to [Ber12] and the references quoted there. In particular, in favorable instances of the problem, optimal policies have the character of an *index rule*, which is structurally similar to the decoupled suboptimal decision rules discussed in this section, and has been analyzed extensively, together with its variations and special cases. The term “restless” in the title of the present example, introduced by Whittle [Whi88], refers to the fact that the states of the projects that are not worked on may change.

An important question for the implementation of the preceding suboptimal control scheme is the determination of the separable reward function terms $\tilde{J}^i_{k+1}$. There are several possibilities here, and the best choice may strongly depend on the problem's structure. One possibility is to compute $\tilde{J}^i_{k+1}$ as the optimal cost-to-go function for a problem involving just project i, i.e., assuming that none of the other projects $j \neq i$ will be worked on for the remaining periods $k+1, \ldots, N-1$. This corresponds to restricting the control constraint set of the problem, and involves a single-project optimization that may be tractable.

An alternative possibility is to use a separable parametric approximation of the form

$$\sum_{i=1}^{n} \tilde{J}^i_{k+1}(x^i_{k+1}, r^i_{k+1}),$$

where r^i_{k+1} are vectors of "tunable" parameters. The values of r^i_{k+1} can be obtained by some "training" algorithm such as the ones to be discussed in Chapter 3.

Constraint Decoupling by Lagrangian Relaxation

Another approach to deal with coupled constraints is to replace them with linear Lagrange multiplier-related penalty functions that are added to the cost function. We illustrate this approach with an extension of the preceding multiarmed bandit Example 2.3.2.

Example 2.3.3 (Separable Lower Bound Approximation in Multiarmed Bandit Problems)

Let us consider a version of the multiple projects Example 2.3.2 involving a more general form of control. Here there are n subsystems, with a control u^i_k applied to subsystem i at time k. Instead of the requirement that only one subsystem is worked on at any one time, we assume a control constraint of the form

$$u_k = (u^1_k, \ldots, u^n_k) \in U, \qquad k = 0, 1, \ldots,$$

where the set U is given (Example 2.3.2 is obtained as the special case where U consists of the union of the coordinate vectors, i.e., those whose components are equal to 0, except for one component that is equal to 1). The ith subsystem is described by

$$x^i_{k+1} = f^i(x^i_k, u^i_k, w^i_k), \qquad i = 1, \ldots, n, \ k = 0, 1, \ldots,$$

where x^i_k is the state taking values in some space, u^i_k is the control, w^i_k is a random disturbance, and f^i is a given function. We assume that w^i_k is selected according to a probability distribution that may depend on x^i_k and u^i_k, but not on prior disturbances or the disturbances w^j_k of the other subsystems $j \neq i$. The cost incurred at the kth stage by the ith subsystem is

$$g^i(x^i_k, u^i_k, w^i_k), \tag{2.13}$$

where g^i is a given one-stage cost function. For notational convenience, we assume stationarity of the system equation and the cost per stage, but the approach to be discussed applies to the nonstationary case as well.

One possibility for a separable approximation of the problem is to replace the constraint $u_k \in U$ by a smaller or larger decoupled constraint, i.e., requiring that

$$u_k^i \in U^i, \qquad i = 1, \ldots, n, \; k = 0, 1, \ldots,$$

where the subsets $U^1, \ldots, U^n$ satisfy $U^1 \times \cdots \times U^n \subset U$ or $U \subset U^1 \times \cdots \times U^n$, respectively.

We discuss another possibility for the case where the constraint set U includes linear inequality constraints. As a simple example, let us focus on a constraint set U of the form

$$U = \left\{ (u^1, \ldots, u^n) \;\middle|\; u^i \in U^i \subset \Re, \; i = 1, \ldots, n, \; \sum_{i=1}^{n} c^i u^i \leq b \right\}, \tag{2.14}$$

where $c^1, \ldots, c^n$, and b are some scalars.† Here we replace the coupling constraint

$$\sum_{i=1}^{n} c^i u_k^i \leq b, \qquad k = 0, 1, \ldots, \tag{2.15}$$

by a "relaxed" (larger) constraint whereby we require that

$$\sum_{k=0}^{N-1} \sum_{i=1}^{n} c^i u_k^i \leq Nb. \tag{2.16}$$

Roughly speaking, the constraint (2.16) requires that the coupling constraint (2.15) is satisfied "on the average," over the N stages.

We may now obtain a lower bound approximation of the optimal cost of our problem by assigning a scalar Lagrange multiplier $\lambda \geq 0$ to the constraint (2.16), and add a Lagrangian term

$$\lambda \left(\sum_{k=0}^{N-1} \sum_{i=1}^{n} c^i u_k^i - Nb \right) \tag{2.17}$$

to the cost function. This amounts to replacing the kth stage cost (2.13) by

$$g^i(x_k^i, u_k^i, w_k^i) + \lambda c^i u_k^i,$$

while replacing the coupling constraint (2.14) with the decoupled constraint

$$u_k^i \in U^i, \qquad i = 1, \ldots, n,$$

† More general cases, where u^i and b are multi-dimensional, and c^i are replaced by matrices of appropriate dimension, can be handled similarly, albeit with greater computational complications.

cf. Eq. (2.14). This is a lower bound approximation, as is typical in Lagrange multiplier-based decomposition approaches in linear and nonlinear programming (see e.g., [BeT97], [Ber16b]). To see this, note that for every feasible solution of the original problem, the Lagrangian term (2.17) makes a nonpositive contribution when added to the cost function, while with the constraint relaxed, the resulting optimal cost can only be reduced further.

With the subsystems now decoupled, we may solve each single subsystem problem separately, thereby obtaining a separable lower bound approximation

$$\sum_{i=1}^{n} \tilde{J}_k^i(x_k^i, \lambda)$$

for every $k = 1, \ldots, N-1$. This approximation can in turn be used to obtain a suboptimal one-step lookahead policy. Note that we may also try to optimize the approximation over λ, either by ad hoc experimentation or by a more systematic optimization method.† Another possibility is to use a more general Lagrangian term of the form

$$\left(\sum_{k=0}^{N-1} \lambda_k \sum_{i=1}^{n} c^i u_k^i - Nb \right),$$

in place of the term (2.17), where $\lambda_0, \ldots, \lambda_{N-1} \geq 0$ are time-varying scalar multipliers.

2.3.2 Probabilistic Approximation - Certainty Equivalent Control

We will now consider problem approximation based on modifying the underlying probabilistic structure. The most common example of this type is the *certainty equivalent controller* (CEC). It replaces the stochastic disturbances with deterministic variables that are fixed at some "typical" values. Thus it acts as if a form of the certainty equivalence principle were holding, cf. the discussion of linear quadratic problems in Section 1.3.7.

The advantage of the CEC is that it involves a much less demanding computation than the stochastic DP algorithm: it requires the solution of a *deterministic* optimal control problem at each stage. This problem yields an optimal control sequence, the first component of which is used at the current stage, while the remaining components are discarded. Thus the CEC is able to deal with stochastic and even partial information problems by using the more flexible and potent methodology of deterministic optimal control.

---

† Maximization of the lower bound approximation over λ is an interesting possibility. This is common in duality-based optimization. Generally the approach of this example falls within the framework of *Lagrangian relaxation*, a decomposition method that is based on the use of Lagrange multipliers and duality theory; see e.g., [BeT97], [Ber15a], [Ber16a].

We will describe the CEC for the stochastic DP problem of Section 1.2. Suppose that for every state-control pair (x_k, u_k) we have selected a "typical" value of the disturbance, which we denote by $\tilde{w}_k(x_k, u_k)$. For example the expected values

$$\tilde{w}_k(x_k, u_k) = E\{w_k \mid x_k, u_k\},$$

can serve as typical values, if the disturbance spaces are convex subsets of Euclidean spaces [so that they include $\tilde{w}_k(x_k, u_k)$].

To implement the CEC at state x_k and stage k we solve a deterministic optimal control problem obtained from the original problem by replacing all uncertain quantities by their typical values. In particular, we solve the problem

$$\min_{\substack{x_{i+1}=f_i(x_i,u_i,\tilde{w}_i(x_i,u_i)) \\ u_i\in U_i(x_i),\ i=k,\ldots,N-1}} \left[g_N(x_N) + \sum_{i=k}^{N-1} g_i\big(x_i, u_i, \tilde{w}_i(x_i, u_i)\big) \right]. \tag{2.18}$$

If $\{\tilde{u}_k, \ldots, \tilde{u}_{N-1}\}$ is the optimal control sequence for this problem, we use the first control in this sequence and discard the remaining controls:

$$\tilde{\mu}_k(x_k) = \tilde{u}_k.$$

An alternative implementation of the CEC is to compute off-line an optimal policy

$$\big\{\mu_0^d(x_0), \ldots, \mu_{N-1}^d(x_{N-1})\big\}$$

for the deterministic problem

$$\begin{aligned} &\text{minimize} \quad g_N(x_N) + \sum_{k=0}^{N-1} g_k\big(x_k, \mu_k(x_k), \tilde{w}_k(x_k, u_k)\big) \\ &\text{subject to} \quad x_{k+1} = f_k\big(x_k, \mu_k(x_k), \tilde{w}_k(x_k, u_k)\big), \quad \mu_k(x_k) \in U_k, \quad k \geq 0, \end{aligned} \tag{2.19}$$

by using the DP algorithm. Then the control input $\tilde{\mu}_k(I_k)$ applied by the CEC at time k is given by

$$\tilde{\mu}_k(I_k) = \mu_k^d(x_k).$$

The two variants of the CEC just given are equivalent in terms of performance. The main difference is that the first variant is well-suited for on-line replanning, while the second variant is more suitable for an off-line implementation.

Finally, let us note that the CEC can be extended to imperfect state observation problems, where the state x_k is not known at time k, but instead an estimate of x_k is available, which is based on measurements that have been obtained up to time x_k. In this case, we find a suboptimal control similarly, as in Eqs. (2.18) and (2.19), but with x_k replaced by the estimate, as if this estimate were exact.

Certainty Equivalent Control with Heuristics

Even though the CEC approach simplifies a great deal the computations, it still requires the optimal solution of a deterministic tail subproblem at each stage [cf. Eq. (2.18)]. This problem may still be difficult, and a more convenient approach may be to solve it suboptimally using a heuristic algorithm. In particular, given the state x_k at time k, we may use some (easily implementable) heuristic to find a suboptimal control sequence $\{\tilde{u}_k, \tilde{u}_{k+1}, \ldots, \tilde{u}_{N-1}\}$ for the problem of Eq. (2.18), and then use $\tilde{u}_k$ as the control for stage k.

An important enhancement of this idea is to use minimization over the first control u_k and to use the heuristic only for the remaining stages $k+1, \ldots, N-1$. To implement this variant of the CEC, we must apply at time k a control $\tilde{u}_k$ that minimizes over $u_k \in U_k(x_k)$ the expression

$$g_k\big(x_k, u_k, \tilde{w}_k(x_k, u_k)\big) + H_{k+1}(x_{k+1}), \tag{2.20}$$

where

$$x_{k+1} = f_k\big(x_k, u_k, \tilde{w}_k(x_k, u_k)\big), \tag{2.21}$$

and H_{k+1} is the cost-to-go function corresponding to the heuristic, i.e., $H_{k+1}(x_{k+1})$ is the cost incurred over the remaining stages $k+1, \ldots, N-1$ starting from a state x_{k+1}, using the heuristic. This is a hybrid approach: it resembles one-step lookahead with lookahead function H_{k+1}, and it resembles certainty equivalence in that the uncertain quantities have been replaced by their typical values.

Note that for any next state x_{k+1}, it is not necessary to have a closed-form expression for the heuristic cost-to-go $H_{k+1}(x_{k+1})$. Instead we can generate this cost by running the heuristic from x_{k+1} and computing the corresponding cost. Thus all the possible next states x_{k+1} must be computed for all possible values of the control u_k, and then the heuristic must be run from each of these x_{k+1} to calculate $H_{k+1}(x_{k+1})$, which is needed in the minimization of the expression (2.20).

Example 2.3.4 (Parking with Probability Estimates)

Consider the one-directional parking problem of Example 1.3.3, where a driver is looking for a parking space along a line of N spaces, with a garage at the end of the line (position N). The driver starts at space 0 and traverses the parking spaces sequentially, i.e., continues to subsequent spaces, up to a decision to park in space k at cost $c(k)$, if space k is free, or upon reaching the garage where parking is mandatory at cost C. In Example 1.3.3, we assumed that space k is free with a given and fixed probability $p(k)$, independently of whether other parking spaces are free or not.

Assume instead that $p(k)$ is an estimate that is based on the driver's observations of the status of preceding spaces. For example, this estimation process may involve exploiting probabilistic relations that may exist between

the parking statuses of different spaces. In particular, let us assume that the driver, upon arrival at space k, can estimate the belief state of the spaces that lie ahead, i.e., the vector of conditional probabilities $\big(p(k+1), \ldots, p(N)\big)$ given the observations of the spaces $0, \ldots, k$.

The problem now becomes very hard to solve by exact DP, because the state space is infinite: at time k the state consists of the free/taken status of the current position k, plus the belief state of the spaces that lie ahead. However, a simple suboptimal approach to the problem can be based on certainty equivalence: at time k, we fix the free/taken probabilities of the spaces that lie ahead to their current belief values, and act as if these values will not change in the future. Then upon reaching space k, the fast DP algorithm of Example 1.3.3 can be used to solve on-line the corresponding fixed probabilities problem, and to find the corresponding suboptimal decision.

As an illustration, let $p(k)$ be estimated by using the ratio $R(k)$ of the number of free spaces encountered up to reaching space k divided by the total number $k+1$. Knowing $R(k)$, the driver adjusts the probabilities $p(m)$ for $m > k$ to the level

$$\hat{p}\big(m, R(k)\big) = \gamma p(m) + (1 - \gamma)R(k),$$

where γ is a known constant between 0 and 1. The problem can then be solved by exact DP, by using as state at time k the free/taken status of space k together with the ratio $R(k)$ [in the terminology of Section 1.3, $R(k)$ is a sufficient statistic, which contains all the relevant information for the purposes of control]. The number of possible values of $R(k)$ grows exponentially with k, so the solution by exact DP may become intractable for large N. However, by applying the probabilistic approximation approach of this example, the corresponding suboptimal policy may be easily obtained and implemented on-line.

Partial Certainty Equivalent Control

In the preceding descriptions of the CEC all future and present uncertain quantities are fixed at their typical values. A useful variation is to fix at typical values only *some* of these quantities. For example, a partial state information problem may be treated as one of perfect state information, using an estimate $\tilde{x}_k$ of x_k as if it were exact, while fully taking into account the stochastic nature of the disturbances. Thus, if $\big\{\mu_0^p(x_0), \ldots, \mu_{N-1}^p(x_{N-1})\big\}$ is an optimal policy obtained from the DP algorithm for the stochastic perfect state information problem

$$\text{minimize} \;\; E\left\{ g_N(x_N) + \sum_{k=0}^{N-1} g_k\big(x_k, \mu_k(x_k), w_k\big) \right\}$$

$$\text{subject to} \;\; x_{k+1} = f_k\big(x_k, \mu_k(x_k), w_k\big), \quad \mu_k(x_k) \in U_k, \quad k = 0, \ldots, N-1,$$

then the control input applied by this variant of CEC at time k is $\mu_k^p(\tilde{x}_k)$, where $\tilde{x}_k$ is the estimate of x_k given the information available up to time k. Let us provide an example.

Example 2.3.5 (The Unscrupulous Innkeeper)

Consider an innkeeper who charges one of m different rates $r_1, \ldots, r_m$ for a room as the day progresses, depending on whether he has many or few vacancies, so as to maximize his expected total income during the day. A quote of a rate r_i is accepted with probability p_i and is rejected with probability $1 - p_i$, in which case the customer departs, never to return during that day. When the number y of customers that will ask for a room during the rest of the day (including the customer currently asking for a room) is known and the number of vacancies is x, the optimal expected income $\tilde{J}(x, y)$ of the innkeeper is given by the DP algorithm

$$\tilde{J}(x, y) = \max_{i=1,\ldots,m} \Big[p_i\big(r_i + \tilde{J}(x-1, y-1)\big) + (1 - p_i)\tilde{J}(x, y-1)\Big], \tag{2.22}$$

for all $x \geq 1$ and $y \geq 1$, with initial conditions

$$\tilde{J}(x, 0) = \tilde{J}(0, y) = 0, \qquad \text{for all } x \text{ and } y.$$

This algorithm can be used to obtain the values of $\tilde{J}(x, y)$ for all pairs (x, y).

Consider now the case where the innkeeper does not know y at the times of decision, but instead only maintains a probability distribution for y. Then, it can be seen that the problem becomes a difficult partial state information problem. The exact DP algorithm should then be executed over the set of the pairs of x and the belief state of y. Yet a reasonable partially stochastic CEC is based on approximating the optimal cost-to-go of subsequent decisions with $\tilde{J}(x-1, \tilde{y}-1)$ or $\tilde{J}(x, \tilde{y}-1)$, where the function $\tilde{J}$ is calculated by the preceding recursion (2.22) and $\tilde{y}$ is an estimate of y, such as the closest integer to the expected value of y. In particular, according to this one-step lookahead policy, when the innkeeper has a number of vacancies $x \geq 1$, he quotes to the current customer the rate that maximizes

$$p_i\big(r_i + \tilde{J}(x-1, \tilde{y}-1) - \tilde{J}(x, \tilde{y}-1)\big).$$

Thus in this suboptimal scheme, the innkeeper acts as if the estimate $\tilde{y}$ were exact.

Other Variations of Certainty Equivalent Control

It is also possible to use more general approaches to implement one-step lookahead control based on simplification of the probabilistic structure of the problem. The possibilities are highly problem-dependent by nature, but we may distinguish a few general techniques, which we illustrate through examples.

Example 2.3.6 (Decoupling Disturbance Distributions)

Let us consider a CEC approach in the context of enforced decomposition (cf. Section 2.3.1) when the subsystems are coupled only through the disturbance. In particular, consider n subsystems of the form

$$x_{k+1}^i = f^i(x_k^i, u_k^i, w_k^i), \qquad i = 1, \ldots, n.$$

Here the ith subsystem has its own state x_k^i, control u_k^i, and cost per stage $g^i(x_k^i, u_k^i, w_k^i)$, but the probability distribution of w_k^i depends on the full state $x_k = (x_k^1, \ldots, x_k^n)$.

A natural form of suboptimal control is to solve at each stage k and for each i, the ith subsystem optimization problem where *the probability distribution of the future disturbances* $w_{k+1}^i, \ldots, w_{N-1}^i$ *is "decoupled,"* in the sense that it depends only on the corresponding "local" states $x_{k+1}^i, \ldots, x_{N-1}^i$. This distribution may be derived by using some nominal values $\tilde{x}_{k+1}^j, \ldots, \tilde{x}_{N-1}^j$, $j \neq i$, of the future states of the other subsystems, and these nominal values may in turn depend on the full current state x_k. The first control $\overline{u}_k^i$ in the optimal policy thus obtained is applied at the ith subsystem in stage k, and the remaining portion of this policy is discarded.

Example 2.3.7 (Approximation Using Scenarios)

We noted earlier the possibility to approximate the optimal cost-to-go with a CEC approach, whereby for a given state x_{k+1} at time $k+1$, we fix the remaining disturbances at some nominal values $\tilde{w}_{k+1}, \ldots, \tilde{w}_{N-1}$, and we compute an optimal control or heuristic-based trajectory starting from x_{k+1} at time $k+1$.

This CEC approximation involves a single nominal trajectory of the remaining uncertainty. To strengthen this approach, it is natural to consider multiple trajectories of the uncertainty, called *scenarios*, and to construct an approximation to the optimal cost-to-go that involves, for every one of the scenarios, the cost of either an optimal or a heuristic policy.

Mathematically, we assume that we have a method, which at a given state x_{k+1}, generates q uncertainty sequences

$$w^s(x_{k+1}) = (w_{k+1}^s, \ldots, w_{N-1}^s), \qquad s = 1, \ldots, q.$$

These are the scenarios considered at state x_{k+1}. The optimal cost $J_{k+1}^*(x_{k+1})$ is then approximated by

$$\tilde{J}_{k+1}(x_{k+1}, r) = \sum_{s=1}^{q} r_s C_s(x_{k+1}), \tag{2.23}$$

where $r = (r_1, \ldots, r_q)$ is a probability distribution, i.e., a vector of nonnegative parameters that add to 1, and $C_s(x_{k+1})$ is the cost corresponding to an occurrence of the scenario $w^s(x_{k+1})$, when starting from state x_{k+1} and using either an optimal or a given heuristic policy.

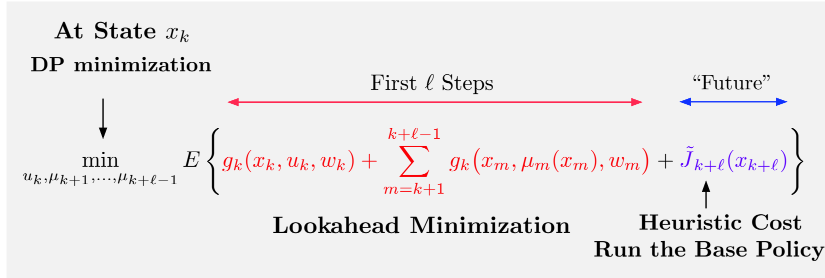

Figure 2.4.1 Schematic illustration of rollout with ℓ-step lookahead. The approximate cost $\tilde J_{k+\ell}(x_{k+\ell})$ is obtained by running a heuristic algorithm/base policy from state $x_{k+\ell}$.

There may be several problem-dependent ways to generate the scenarios, possibly including randomization and/or simulation. The parameters $r_1,\ldots,r_q$ may depend on the time index, and may be interpreted as "aggregate probabilities" that encode the aggregate effect on the cost-to-go function of uncertainty sequences that are similar to the scenario $w^s(x_{k+1})$. They may be computed using some ad hoc scheme, or some more systematic approach. The idea of simplifying the probabilistic model of the system, possibly using a model-free Monte-Carlo type of process, is also related to the rollout approach that is the subject of the next section.

2.4 ROLLOUT

The principal aim of rollout is *policy improvement*, i.e., start with a suboptimal/heuristic policy, called the *base policy* (or sometimes, the *default policy*), and produce an improved policy by limited lookahead minimization with use of the heuristic at the end. This policy is called the *rollout policy*, and the fact that it is indeed "improved" will be established, under various conditions, in what follows in this section and also in Chapter 4.

In its purest one-step lookahead form, rollout can be defined very simply: it is approximation in value space with the approximate cost-to-go values $\tilde J_{k+1}(x_{k+1})$ calculated by running the base policy, starting from each possible next state x_{k+1}. There is also an ℓ-step lookahead generalization, where the heuristic is used to obtain the approximate cost-to-go values $\tilde J_{k+\ell}(x_{k+\ell})$ from each possible next state $x_{k+\ell}$ (see Fig. 2.4.1). In a variant for problems involving a long horizon, the run of the base policy may be "truncated," i.e., it may be used for a limited number of steps, with some cost function approximation at the end to take into account the cost of the remaining steps.

The choice of base policy is of course important for the performance of the rollout approach. However, experimental evidence has shown that the choice of base policy may not be crucial for many contexts, and in

fact surprisingly good rollout performance may be attained even with a relatively poor base heuristic, particularly with a long lookahead.

Note also here a connection and overlap between the rollout and problem approximation approaches. Suppose that we use as base heuristic an optimal policy for the approximating problem. Then *the one-step (or multistep) rollout policy is the same as the one obtained by one-step (or multistep, respectively) lookahead with terminal cost function approximation equal to the optimal cost of the approximating problem.*

In Chapter 4 within the context of infinite horizon problems, we will view there the method of policy iteration as a *perpetual rollout algorithm*, whereby a sequence of suboptimal policies is generated, each one being a rollout policy obtained by using the preceding one as a base policy. In this chapter, however, we focus on a one-time policy improvement starting from a given base policy.

We will describe rollout first for finite-state deterministic problems and one-step lookahead, and then for stochastic problems, in Sections 2.4.1 and 2.4.2, respectively. We will discuss rollout for infinite-spaces problems, including model predictive control in Section 2.5. We will also discuss the use of rollout in combination with some other approximation scheme. This is to use one-step or multistep lookahead with the cost function approximation consisting of two parts:

(a) Rollout with the given base policy over a limited horizon.

(b) A terminal cost function approximation at the end of the rollout horizon, such as an estimate of the true cost function of the policy.

We will reconsider schemes of this type in Chapter 4, Section 4.5.3, in the context of infinite horizon problems.

2.4.1 On-Line Rollout for Deterministic Finite-State Problems

Let us consider a deterministic DP problem with a finite number of controls and a given initial state (so the number of states is also finite). Given a state x_k at time k, rollout considers all the tail subproblems that start at every possible next state x_{k+1}, and solves them suboptimally by using some algorithm, referred to as *base heuristic*.† Thus when at x_k, rollout generates on-line the next states x_{k+1} that correspond to all $u_k \in U_k(x_k)$, and uses the base heuristic to compute the sequence of states $\{x_{k+1}, \ldots, x_N\}$ and controls $\{u_{k+1}, \ldots, u_{N-1}\}$ such that

$$x_{i+1} = f_i(x_i, u_i), \qquad i = k, \ldots, N-1.$$

† For deterministic problems we prefer to use the term "base heuristic" rather than "base policy" for reasons to be explained later in this section, in the context of the notion of sequential consistency.

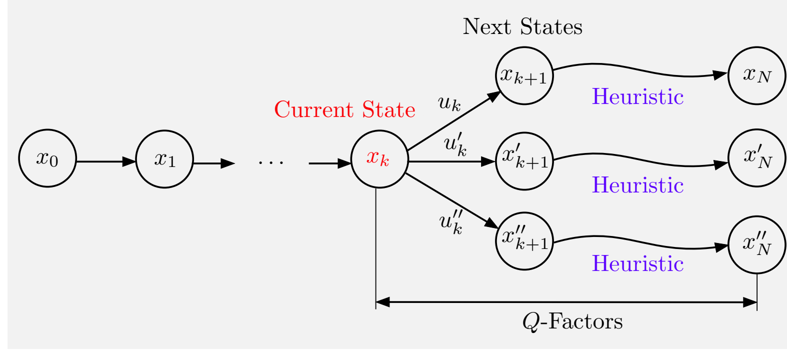

Figure 2.4.2 Schematic illustration of rollout with one-step lookahead for a deterministic problem. At state x_k, for every pair (x_k, u_k), $u_k \in U_k(x_k)$, the base heuristic generates a Q-factor $\tilde{Q}_k(x_k, u_k)$ [cf. Eq. (2.25)], and selects the control $\tilde{\mu}_k(x_k)$ with minimal Q-factor.

The rollout algorithm then applies the control that minimizes over $u_k \in U_k(x_k)$ the tail cost expression for stages k to N:

$$g_k(x_k, u_k) + g_{k+1}(x_{k+1}, u_{k+1}) + \cdots + g_{N-1}(x_{N-1}, u_{N-1}) + g_N(x_N). \quad (2.24)$$

Equivalently, and more succinctly, the rollout algorithm applies at state x_k the control $\tilde{\mu}_k(x_k)$ given by the minimization

$$\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k),$$

where $\tilde{Q}_k(x_k, u_k)$ is the approximate Q-factor defined by

$$\tilde{Q}_k(x_k, u_k) = g_k(x_k, u_k) + H_{k+1}\big(f_k(x_k, u_k)\big), \quad (2.25)$$

with $H_{k+1}(x_{k+1})$ denoting the cost of the base heuristic starting from state x_{k+1} [i.e., $H_{k+1}(x_{k+1})$ is the sum of all the terms in Eq. (2.24), except the first]; see Fig. 2.4.2. The rollout process defines a suboptimal policy $\tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$, referred to as the *rollout policy*.

Example 2.4.1 (Traveling Salesman Problem)

Let us consider the traveling salesman problem, whereby a salesman wants to find a minimum mileage/cost tour that visits each of N given cities $c = 0, \ldots, N-1$ exactly once and returns to the city he started from (cf. Example 1.3.1). With each pair of distinct cities c, c', we associate a traversal cost $g(c, c')$. Note that we assume that we can go directly from every city to every other city. There is no loss of generality in doing so because we can assign a very high cost $g(c, c')$ to any pair of cities (c, c') that is precluded from participation in the solution. The problem is to find a visit order that goes through each city exactly once and whose sum of costs is minimum.

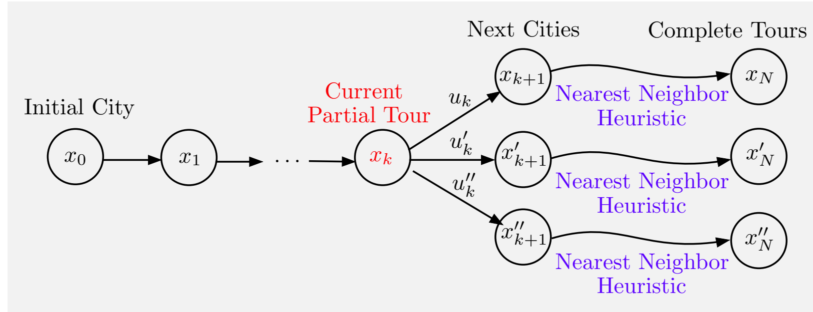

Figure 2.4.3 Schematic illustration of rollout with the nearest neighbor heuristic for the traveling salesman problem. The initial state x_0 consists of a single city. The final state x_N is a complete tour of N cities, containing each city exactly once.

There are many heuristic approaches for solving the traveling salesman problem. For illustration purposes, let us focus on the simple *nearest neighbor* heuristic, which constructs a sequence of *partial tours*, i.e., sequences of ordered collections of distinct cities. Here, we select a single city c_0 and at each iteration, we add to the current partial tour a city that does not close a cycle and minimizes the cost of the enlargement. In particular, after k iterations, we have a sequence $\{c_0, \ldots, c_k\}$ consisting of distinct cities, and at the next iteration, we add a new city c_{k+1} that minimizes $g(c_k, c_{k+1})$ over all cities $c_{k+1} \neq c_0, \ldots, c_k$. After the nearest neighbor heuristic selects city c_{N-1}, a complete tour is formed with total cost

$$g(c_0, c_1) + \cdots + g(c_{N-2}, c_{N-1}) + g(c_{N-1}, c_0). \tag{2.26}$$

We can formulate the traveling salesman problem as a DP problem as we discussed in Example 1.3.1. We choose a starting city, say c_0, as the initial state x_0. Each state x_k corresponds to a partial tour $(c_0, c_1, \ldots, c_k)$ consisting of distinct cities. The states x_{k+1}, next to x_k, are sequences of the form $(c_0, c_1, \ldots, c_k, c_{k+1})$ which correspond to adding one more unvisited city $c_{k+1} \neq c_0, c_1, \ldots, c_k$. The terminal states x_N are the complete tours of the form $(c_0, c_1, \ldots, c_{N-1}, c_0)$, and the cost of the corresponding sequence of city choices is the cost of the corresponding complete tour. Thus a state trajectory yields a complete tour with total cost given by Eq. (2.26).

Let us now use as a base heuristic the nearest neighbor method. The corresponding rollout algorithm operates as follows: After $k < N - 1$ iterations, we have a state x_k, i.e., sequence $\{c_0, \ldots, c_k\}$ consisting of distinct cities. At the next iteration, we add one more city as follows: We run the nearest neighbor heuristic starting from each of the sequences of the form $\{c_0, \ldots, c_k, c\}$ where $c \neq c_1, \ldots, c_k$. We then select as next city c_{k+1} the city c that yielded the minimum cost tour under the nearest neighbor heuristic; see Fig. 2.4.3.

Cost Improvement with a Rollout Algorithm - Sequential Consistency

The definition of the rollout algorithm leaves open the choice of the base heuristic. There are several types of suboptimal solution methods that can be used as base heuristics, such as greedy algorithms, local search, genetic algorithms, tabu search, and others. Clearly we want to choose a base heuristic that strikes a good balance between quality of solutions produced and computational tractability.

Intuitively, we expect that the rollout policy's performance is no worse than the one of the base heuristic. Since rollout optimizes over the first control before applying the heuristic, it makes sense to conjecture that it performs better than applying the heuristic without the first control optimization. However, some special conditions must hold in order to guarantee this cost improvement property. We provide two such conditions, *sequential consistency* and *sequential improvement*, and then show how to modify the algorithm to deal with the case where these conditions are not satisfied.

We say that the base heuristic is *sequentially consistent* if it has the property that when it generates the sequence

$$\{x_k, x_{k+1}, \dots, x_N\}$$

starting from state x_k, it also generates the sequence

$$\{x_{k+1}, \dots, x_N\}$$

starting from state x_{k+1}. In other words, the base heuristic is sequentially consistent if it "stays the course": when the starting state x_k is moved forward to the next state x_{k+1} of its state trajectory, the heuristic will not deviate from the remainder of the trajectory.

As an example, the reader may verify that the nearest neighbor heuristic described in the traveling salesman Example 2.4.1 is sequentially consistent. Similar examples include the use of many types of greedy heuristics (see [Ber17], Section 6.4). Generally most heuristics used in practice satisfy the sequential consistency condition at "most" states x_k. However, some heuristics of interest may violate this condition at some states.

Conceptually, it is important to note that sequential consistency is equivalent to the heuristic being a legitimate DP policy. By this we mean that there exists a policy $\{\mu_0, \dots, \mu_{N-1}\}$ such that the sequence generated by the base heuristic starting from any state x_k is the same as the one generated by $\{\mu_0, \dots, \mu_{N-1}\}$ starting from the same state x_k. To see this, note that a policy clearly has the sequential consistency property, and conversely, a sequentially consistent base heuristic defines a policy: the one that moves from x_k to the state x_{k+1} that lies on the path $\{x_k, x_{k+1}, \dots, x_N\}$ generated by the base heuristic.

Based on this fact, we can show that *the rollout algorithm obtained with a sequentially consistent base heuristic yields an improved cost over the base heuristic*. In particular, let us consider the rollout policy $\tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$, and let $J_{k,\tilde{\pi}}(x_k)$ denote the cost obtained with $\tilde{\pi}$ starting from x_k. We claim that

$$J_{k,\tilde{\pi}}(x_k) \leq \hat{J}_k(x_k), \qquad \text{for all } x_k \text{ and } k, \tag{2.27}$$

where $\hat{J}_k(x_k)$ denotes the cost of the base heuristic starting from x_k.

We prove this inequality by induction. Clearly it holds for $k = N$, since $J_{N,\tilde{\pi}} = H_N = g_N$. Assume that it holds for index $k+1$. For any state x_k, let $\overline{u}_k$ be the control applied by the base heuristic at x_k. Then we have

$$\begin{aligned} J_{k,\tilde{\pi}}(x_k) &= g_k\big(x_k, \tilde{\mu}_k(x_k)\big) + J_{k+1,\tilde{\pi}}\Big(f_k\big(x_k, \tilde{\mu}_k(x_k)\big)\Big) \\ &\leq g_k\big(x_k, \tilde{\mu}_k(x_k)\big) + H_{k+1}\big(f_k(x_k, \tilde{\mu}_k(x_k))\big) \\ &= \min_{u_k \in U_k(x_k)} \Big[g_k(x_k, u_k) + H_{k+1}\big(f_k(x_k, u_k)\big)\Big] \\ &\leq g_k\big(x_k, \overline{u}_k\big) + H_{k+1}\big(f_k(x_k, \overline{u}_k)\big) \\ &= H_k(x_k), \end{aligned} \tag{2.28}$$

where:

(a) The first equality is the DP equation for the rollout policy $\tilde{\pi}$.

(b) The first inequality holds by the induction hypothesis.

(c) The second equality holds by the definition of the rollout algorithm.

(d) The third equality is the DP equation for the policy that corresponds to the base heuristic (this is the step where we need sequential consistency).

This completes the induction proof of the cost improvement property (2.27).

Sequential Improvement

We will now show that the rollout policy has no worse performance than its base heuristic under a condition that is weaker than sequential consistency. Let us recall that the rollout algorithm $\tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$ is defined by the minimization

$$\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k)$$

where $\tilde{Q}_k(x_k, u_k)$ is the approximate Q-factor defined by

$$\tilde{Q}_k(x_k, u_k) = g_k(x_k, u_k) + H_{k+1}\big(f_k(x_k, u_k)\big),$$

[cf. Eqs. (2.25)], and $H_{k+1}(x_{k+1})$ denotes the cost of the base heuristic starting from state x_{k+1}.

We say that the base heuristic is *sequentially improving*, if for all x_k and k, we have

$$\min_{u_k \in U_k(x_k)} \Big[g_k(x_k, u_k) + H_{k+1}\big(f_k(x_k, u_k)\big) \Big] \le H_k(x_k). \qquad (2.29)$$

In words, the sequential improvement property (2.29) states that

$$\text{Best heuristic Q-factor at } x_k \le \text{Heuristic cost at } x_k. \qquad (2.30)$$

To show that a sequentially improving heuristic yields policy improvement, simply note that from the calculation of Eq. (2.28), replacing the last two steps (that rely on sequential consistency) with Eq. (2.29), we have

$$J_{k,\tilde{\pi}}(x_k) \le H_k(x_k), \qquad \text{for all } x_k \text{ and } k.$$

Thus the rollout algorithm obtained with a sequentially improving base heuristic, will improve or at least will perform no worse than the base heuristic, from every starting state x_k. Note that when the heuristic is sequentially consistent it is also sequentially improving, since in this case Eq. (2.29) is satisfied with equality. This also follows from the interpretation (2.30), since for a sequentially consistent heuristic, the heuristic cost is the Q-factor of the heuristic at x_k.

Empirically, it has been observed that the cost improvement obtained by rollout with a sequentially improving heuristic is typically considerable and often dramatic. Generally, however, it is hard to provide solid theoretical support for this observation. Several case studies support the consistently good performance of rollout (at least in the pure form described in this section); see the end of chapter references. The textbook [Ber17], Section 6.4, provides some detailed worked-out examples. The price for the performance improvement is extra computation that is typically equal to the computation time of the base heuristic times a factor that is a low order polynomial of the problem size.

On the other hand the sequential improvement condition may not hold for a given base heuristic. It is thus important to know that *any heuristic can be made to be sequentially improving with a simple modification*, as we explain next.

The Fortified Rollout Algorithm

We will describe a variant of the rollout algorithm that implicitly uses a sequentially improving base heuristic, so that it has the sequential improvement property (2.29). This variant, called the *fortified rollout algorithm*, starts at x_0, and generates step-by-step a sequence of states

$\{x_0, x_1, \ldots, x_N\}$ and corresponding sequence of controls. Upon reaching state x_k it stores the trajectory

$$\overline{P}_k = \{x_0, u_0, \ldots, u_{k-1}, x_k\}$$

that has been constructed up to stage k, called *permanent* trajectory, and it also stores a *tentative* trajectory

$$\overline{T}_k = \{x_k, \overline{u}_k, \overline{x}_{k+1}, \overline{u}_{k+1}, \ldots, \overline{u}_{N-1}, \overline{x}_N\}$$

with corresponding cost

$$C(\overline{T}_k) = g_k(x_k, \overline{u}_k) + g_{k+1}(\overline{x}_{k+1}, \overline{u}_{k+1}) + \cdots + g_{N-1}(\overline{x}_{N-1}, \overline{u}_{N-1}) + g_N(\overline{x}_N).$$

The tentative trajectory is such that $\overline{P}_k \cup \overline{T}_k$ is the best end-to-end trajectory computed up to stage k of the algorithm. Initially, $\overline{T}_0$ is the trajectory generated by the base heuristic starting at the initial state x_0. The idea now is to deviate from the rollout algorithm at every state x_k where the base heuristic produces a trajectory with larger cost than $\overline{T}_k$, and use $\overline{T}_k$ instead.

In particular, upon reaching state x_k, we run the rollout algorithm as earlier, i.e., for every $u_k \in U_k(x_k)$ and next state $x_{k+1} = f_k(x_k, u_k)$, we run the base heuristic from x_{k+1}, and find the control $\tilde{u}_k$ that gives the best trajectory, denoted

$$\tilde{T}_k = \{x_k, \tilde{u}_k, \tilde{x}_{k+1}, \tilde{u}_{k+1}, \ldots, \tilde{u}_{N-1}, \tilde{x}_N\}$$

with corresponding cost

$$C(\tilde{T}_k) = g_k(x_k, \tilde{u}_k) + g_{k+1}(\tilde{x}_{k+1}, \tilde{u}_{k+1}) + \cdots + g_{N-1}(\tilde{x}_{N-1}, \tilde{u}_{N-1}) + g_N(\tilde{x}_N).$$

Whereas the ordinary rollout algorithm would choose control $\tilde{u}_k$ and move to $\tilde{x}_{k+1}$, the fortified algorithm compares $C(\overline{T}_k)$ and $C(\tilde{T}_k)$, and depending on which of the two is smaller, chooses $\overline{u}_k$ or $\tilde{u}_k$ and moves to $\overline{x}_{k+1}$ or to $\tilde{x}_{k+1}$, respectively. In particular, if $C(\overline{T}_k) \leq C(\tilde{T}_k)$ the algorithm sets the next state and corresponding tentative trajectory to

$$x_{k+1} = \overline{x}_{k+1}, \qquad \overline{T}_{k+1} = \{\overline{x}_{k+1}, \overline{u}_{k+1}, \ldots, \overline{u}_{N-1}, \overline{x}_N\},$$

and if $C(\overline{T}_k) > C(\tilde{T}_k)$ it sets the next state and corresponding tentative trajectory to

$$x_{k+1} = \tilde{x}_{k+1}, \qquad \overline{T}_{k+1} = \{\tilde{x}_{k+1}, \tilde{u}_{k+1}, \ldots, \tilde{u}_{N-1}, \tilde{x}_N\}.$$

In other words the fortified rollout at x_k follows the current tentative trajectory $\overline{T}_k$ unless a lower cost trajectory $\tilde{T}_k$ is discovered by running the

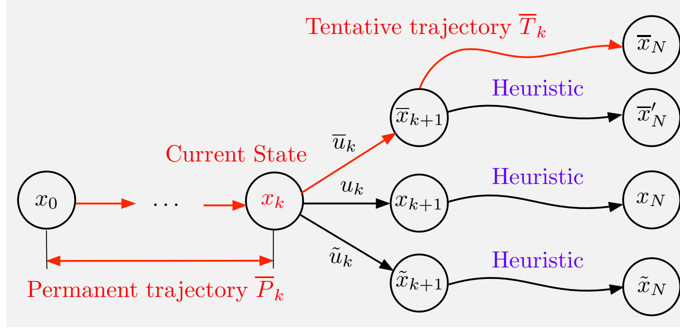

Figure 2.4.4 Schematic illustration of fortified rollout. After k steps, we have constructed the permanent trajectory

$$\overline{P}_k = \{x_0, u_0, \ldots, u_{k-1}, x_k\},$$

and the tentative trajectory

$$\overline{T}_k = \{x_k, \overline{u}_k, \overline{x}_{k+1}, \overline{u}_{k+1}, \ldots, \overline{u}_{N-1}, \overline{x}_N\}$$

such that $\overline{P}_k \cup \overline{T}_k$ is the best end-to-end trajectory computed so far. We now run the rollout algorithm at x_k, i.e., we find the control $\tilde{u}_k$ that minimizes over u_k the sum of $g_k(x_k, u_k)$ plus the heuristic cost from the state $x_{k+1} = f_k(x_k, u_k)$, and the corresponding trajectory

$$\tilde{T}_k = \{x_k, \tilde{u}_k, \tilde{x}_{k+1}, \tilde{u}_{k+1}, \ldots, \tilde{u}_{N-1}, \tilde{x}_N\}.$$

If the cost of the end-to-end trajectory $\overline{P}_k \cup \tilde{T}_k$ is lower than the cost of $\overline{P}_k \cup \overline{T}_k$, we use we add $(\tilde{u}_k, \tilde{x}_{k+1})$ to the permanent trajectory and set the tentative trajectory to

$$\overline{T}_{k+1} = \{\tilde{x}_{k+1}, \tilde{u}_{k+1}, \ldots, \tilde{u}_{N-1}, \tilde{x}_N\}.$$

Otherwise we add $(\overline{u}_k, \overline{x}_{k+1})$ to the permanent trajectory and set the tentative trajectory to

$$\overline{T}_{k+1} = \{\overline{x}_{k+1}, \overline{u}_{k+1}, \ldots, \overline{u}_{N-1}, \overline{x}_N\}.$$

Note that the fortified rollout will produce a different result than the ordinary rollout if the heuristic when started from $\overline{x}_{k+1}$ constructs a trajectory that is different than the tail portion of the tentative trajectory that starts at $\overline{x}_{k+1}$.

base heuristic from all possible next states x_{k+1}. It follows that at every state the trajectory that consists of the union of the permanent and the tentative trajectories, has lower cost than the initial tentative trajectory, which is the one produced by the base heuristic starting from x_0. Moreover, it can be seen that if the base heuristic is sequentially improving, the rollout algorithm and its fortified version coincide. Experimental evidence suggests that it is important to use the fortified version if the base heuristic is not sequentially improving.

Finally we note that the fortified rollout may be viewed as the ordinary rollout algorithm applied to a modified version of the original problem and modified base heuristic that has the sequential improvement property. The corresponding construction is somewhat tedious and will not be given; we refer to [BTW97] and [Ber17], Section 6.4.2.

Using Multiple Heuristics

In many problems, several promising heuristics may be available. It is then possible to use all of these heuristics in the rollout framework. The idea is to construct a *superheuristic*, which selects the best trajectory produced by all the base heuristic trajectories. The superheuristic can then be used as the base heuristic for a rollout algorithm.

In particular, let us assume that we have M base heuristics, and that the mth of these, given a state x_{k+1}, produces a trajectory

$$\tilde{T}_{k+1}^m = \{x_{k+1}, \tilde{u}_{k+1}^m, \ldots, \tilde{u}_{N-1}^m, \tilde{x}_N^m\},$$

and corresponding cost $C(\tilde{T}_{k+1}^m)$. The superheuristic then produces at x_{k+1} the trajectory $\tilde{T}_{k+1}^m$ for which $C(\tilde{T}_{k+1}^m)$ is minimum.

An interesting property, which can be readily verified by using the definitions, is that if all the base heuristics are sequentially improving, the same is true for the superheuristic. Moreover, there is a fortified version of the rollout algorithm, which has the property that it produces a trajectory with no worse cost than all the trajectories produced by the M base heuristics when started at the initial state x_0.

Rollout Algorithms with Multistep Lookahead

We may incorporate multistep lookahead into the deterministic rollout framework. To describe two-step lookahead in its most straightforward implementation, suppose that after k steps we have reached state x_k. We then consider the set of all two-step-ahead states x_{k+2} we run the base heuristic starting from each of them, and compute the two-stage cost to get from x_k to x_{k+2}, plus the cost of the base heuristic from x_{k+2}. We select the state, say $\tilde{x}_{k+2}$, that is associated with minimum cost, compute the controls $\tilde{u}_k$ and $\tilde{u}_{k+1}$ that lead from x_k to $\tilde{x}_{k+2}$, and choose $\tilde{u}_k$ as the next rollout control and $x_{k+1} = f_k(x_k, \tilde{u}_k)$ as the next state; see Fig. 2.4.5.

The extension of the algorithm to lookahead of more than two steps is straightforward: instead of the two-step-ahead states x_{k+2} we run the base heuristic starting from all the possible ℓ-step ahead states $x_{k+\ell}$, etc. For problems with a large number of stages, we can consider *truncated rollout with terminal cost approximation*. Here the rollout trajectories are obtained by running the base heuristic from the leaf nodes of the lookahead tree, and they are truncated after a given number of steps, while a terminal

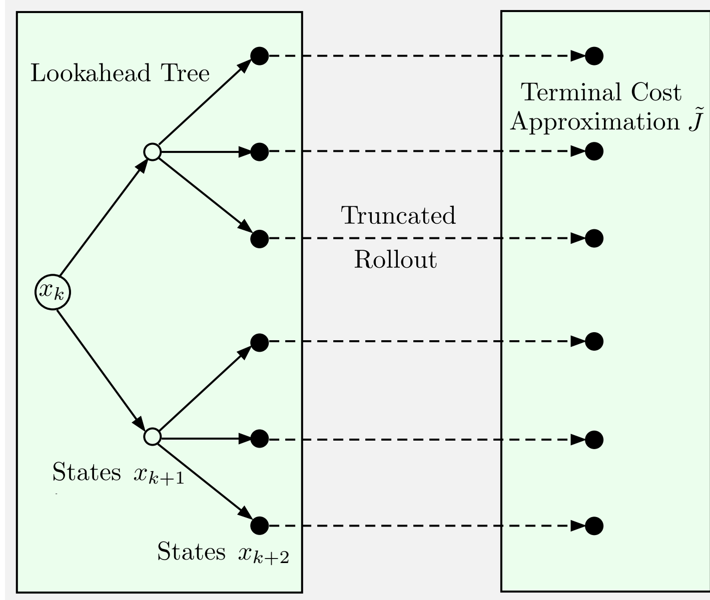

Figure 2.4.5 Illustration of truncated rollout with two-step lookahead and a terminal cost function approximation $\tilde{J}$. The base heuristic is used for a limited number of steps and the terminal cost is added to compensate for the costs of the remaining steps.

cost approximation is added to the heuristic cost to compensate for the resulting error.

Among other variations of deterministic multistep rollout, let us mention a fortified version, which maintains a tentative trajectory, along the lines described earlier for the one-step lookahead case. In still another version of ℓ-step lookahead rollout, we may consider disregarding some of the states that are ℓ steps or less ahead, which are judged less promising according to some criterion (for example the costs of the base heuristic after a one-step lookahead); see Fig. 2.4.6. This may be viewed as *selective depth lookahead*, and aims to limit the number of times that the base heuristic is applied, which can become overwhelming as the length of lookahead is increased. We will encounter again the idea of selective depth lookahead in the context of stochastic rollout and Monte Carlo tree search (see the next section), where in addition to the length of lookahead, the accuracy of the simulation to evaluate the cost of the base heuristic is adapted to the results of earlier computations.

Finally, let us mention a variant of rollout that maintains multiple trajectories, extending from a given state x_k to possibly multiple next states x_{k+1}. These states are the ones considered "most promising" based on the current results of the base heuristic (like being "ϵ-best"), but may be discarded later based on subsequent computations. Such extended forms

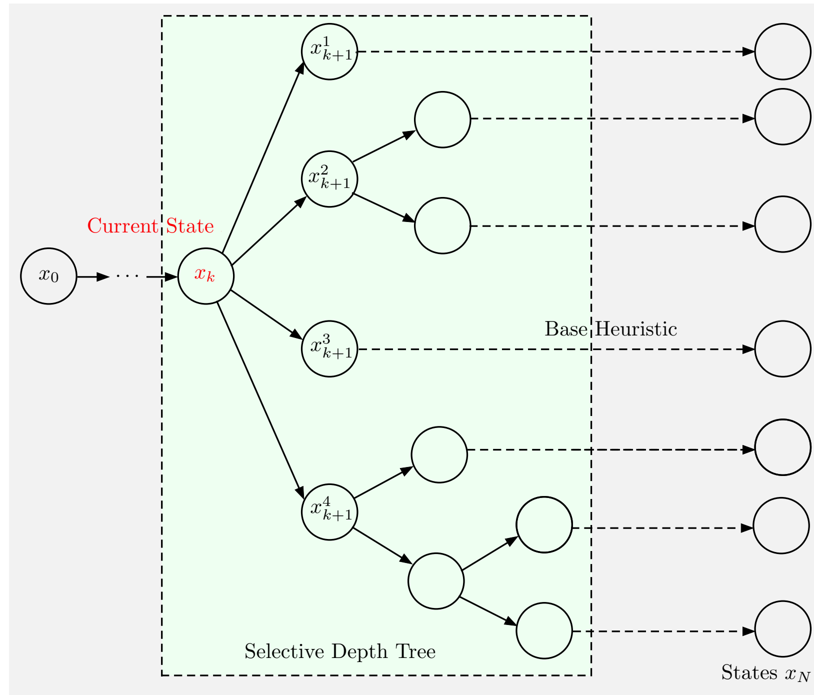

Figure 2.4.6 Illustration of deterministic rollout with selective depth lookahead. After k steps of the algorithm, we have a trajectory that starts at the initial state x_0 and ends at state x_k. We then generate the set of all possible next states (states x^1_{k+1}, x^2_{k+1}, x^3_{k+1}, x^4_{k+1} in the figure). We "evaluate" these states using the base heuristic, and select some of them for "expansion," i.e., we generate their next states x_{k+2}, evaluate them using the base heuristic, and continue. In the end we have a selective depth tree of next states, and the base heuristic costs from the leaves of the tree. The state x_{k+1} that corresponds to smallest overall cost is chosen by the selective depth lookahead rollout algorithm. For problems with a large number of stages, we can also truncate the rollout trajectories and add a terminal cost function approximation as compensation for the resulting error; cf. Fig. 2.4.5.

of rollout are restricted to deterministic problems, and tend to be problem-dependent. We will not consider them further in this book.

2.4.2 Stochastic Rollout and Monte Carlo Tree Search

We will now discuss the extension of the rollout algorithm to stochastic problems with a finite number of states. We will restrict ourselves to the case where the base heuristic is a policy $\pi = \{\mu_0, \ldots, \mu_{N-1}\}$ (i.e., is sequentially consistent, in the context of deterministic problems). It is possible to consider more general rollout algorithms that involve base heuristics with

a sequential improvement property, but we will not pursue this idea, as it does not seem to have been applied so far in interesting stochastic contexts.

We first note that the cost improvement property that we showed for deterministic problems under the sequential consistency condition carries through for stochastic problems. In particular, let us denote by $J_{k,\pi}(x_k)$ the cost corresponding to starting the base policy at state x_k, and by $J_{k,\tilde{\pi}}(x_k)$ the cost corresponding to starting the rollout algorithm at state x_k. We claim that

$$J_{k,\tilde{\pi}}(x_k) \leq J_{k,\pi}(x_k), \qquad \text{for all } x_k \text{ and } k.$$

We prove this inequality by induction similar to the deterministic case. Clearly it holds for $k = N$, since $J_{N,\tilde{\pi}} = J_{N,\pi} = g_N$. Assuming that it holds for index $k + 1$, we have for all x_k,

$$\begin{aligned} J_{k,\tilde{\pi}}(x_k) &= E\Big\{g_k\big(x_k, \tilde{\mu}_k(x_k), w_k\big) + J_{k+1,\tilde{\pi}}\Big(f_k\big(x_k, \tilde{\mu}_k(x_k), w_k\big)\Big)\Big\} \\ &\leq E\Big\{g_k\big(x_k, \tilde{\mu}_k(x_k), w_k\big) + J_{k+1,\pi}\big(f_k(x_k, \tilde{\mu}_k(x_k), w_k)\big)\Big\} \\ &= \min_{u_k \in U_k(x_k)} E\Big\{g_k(x_k, u_k, w_k) + J_{k+1,\pi}\big(f_k(x_k, u_k, w_k)\big)\Big\} \\ &\leq E\Big\{g_k\big(x_k, \mu_k(x_k), w_k\big) + J_{k+1,\pi}\big(f_k(x_k, u_k, w_k)\big)\Big\} \\ &= J_{k,\pi}(x_k), \end{aligned}$$

where:

(a) The first equality is the DP equation for the rollout policy $\tilde{\pi}$.

(b) The first inequality holds by the induction hypothesis.

(c) The second equality holds by the definition of the rollout algorithm.

(d) The third equality is the DP equation for the policy π that corresponds to the base heuristic.

The induction proof of the cost improvement property (2.27) is thus complete.

Similar to deterministic problems, it has been observed empirically that for stochastic problems the rollout policy not only does not deteriorate the performance of the base policy, but also typically produces substantial cost improvement; see the case studies referenced at the end of the chapter.

Simulation-Based Implementation of the Rollout Algorithm

A conceptually straightforward way to compute the rollout control at a given state x_k and time k is to consider each possible control $u_k \in U_k(x_k)$ and to generate a "large" number of simulated trajectories of the system starting from (x_k, u_k). Thus a simulated trajectory has the form

$$x_{i+1} = f_i\big(x_i, \mu_i(x_i), w_i\big), \qquad i = k + 1, \ldots, N - 1,$$

where $\{\mu_{k+1}, \ldots, \mu_{N-1}\}$ is the tail portion of the base policy, the first generated state is

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

and the disturbance sequences $\{w_k, \ldots, w_{N-1}\}$ are independent random samples. The costs of the trajectories corresponding to a pair (x_k, u_k) can be viewed as samples of the Q-factor

$$Q_k(x_k, u_k) = E\Big\{g_k(x_k, u_k, w_k) + J_{k+1,\tilde{\pi}}\big(f_k(x_k, u_k, w_k)\big)\Big\},$$

where $J_{k+1,\tilde{\pi}}$ is the cost-to-go function of the base policy, i.e., $J_{k+1,\tilde{\pi}}(x_{k+1})$ is the cost of using the base policy starting from x_{k+1}. For problems with a large number of stages, it is also common to truncate the rollout trajectories and add a terminal cost function approximation as compensation for the resulting error.

By Monte Carlo averaging of the costs of the sample trajectories plus the terminal cost (if any), we obtain an approximation to the Q-factor $Q_k(x_k, u_k)$ for each control $u_k \in U_k(x_k)$, which we denote by $\tilde{Q}_k(x_k, u_k)$. We then compute the (approximate) rollout control $\tilde{\mu}_k(x_k)$ with the minimization

$$\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k). \tag{2.31}$$

Example 2.4.2 (Backgammon)

The first impressive application of rollout was given for the ancient two-player game of backgammon, in the paper by Tesauro and Galperin [TeG96]; see Fig. 2.4.7. They implemented a rollout algorithm, which attained a level of play that was better than all computer backgammon programs, and eventually better than the best humans. Tesauro had proposed earlier the use of one-step and two-step lookahead with lookahead cost function approximation provided by a neural network, resulting in a backgammon program called TD-Gammon [Tes89a], [Tes89b], [Tes92], [Tes94], [Tes95], [Tes02]. TD-Gammon was trained with the use of the TD(λ) algorithm that will be discussed in Section 4.9, and was used as the base heuristic (for both players) to simulate game trajectories. The rollout algorithm also involved truncation of long game trajectories, using a terminal cost function approximation based on TD-Gammon. Game trajectories are of course random, since they involve the use of dice at each player's turn. Thus the scores of many trajectories have to be generated and Monte Carlo averaged to assess the probability of a win from a given position.

An important issue to consider here is that backgammon is a two-player game and not an optimal control problem that involves a single decision maker. While there is a DP theory for sequential zero-sum games, this theory has not been covered in this book. Thus how are we to interpret rollout algorithms in the context of two-player games? The answer is to treat the two players unequally: one player uses the heuristic policy exclusively (TD-Gammon in the present example). The other player takes the role of the

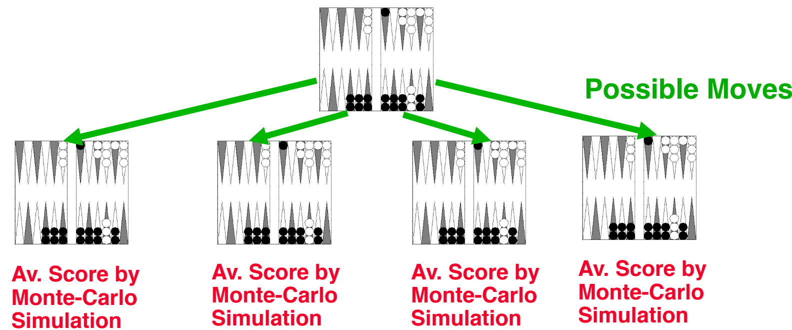

Figure 2.4.7 Illustration of rollout for backgammon. At a given position and roll of the dice, the set of all possible moves is generated, and the outcome of the game for each move is evaluated by "rolling out" (simulating to the end) many games using a suboptimal/heuristic backgammon player (the TD-Gammon player was used for this purpose in [TeG96]), and by Monte-Carlo averaging the scores. The move that results in the best average score is selected for play.

optimizer, and tries to improve on the heuristic policy (TD-Gammon) by using rollout. Thus "policy improvement" in the context of the present example means that when playing against a TD-Gammon opponent, the rollout player achieves a better score on the average than if he/she were to play with the TD-Gammon strategy. In particular, the theory does not guarantee that a rollout player that is trained using TD-Gammon for both players will do better than TD-Gammon would against a non-TD-Gammon opponent. This is a plausible hypothesis, albeit one that can only be tested empirically.

Most of the currently existing computer backgammon programs descend from TD-Gammon. Rollout-based backgammon programs are the most powerful in terms of performance, consistent with the principle that a rollout algorithm performs better than its base heuristic. However, they are too time-consuming for real-time play, because of the extensive on-line simulation requirement at each move (the situation in backgammon is exacerbated by its high branching factor, i.e., for a given position, the number of possible successor positions is quite large, as compared for example with chess). They have been used in a limited diagnostic way to assess the quality of neural network-based programs (many articles and empirical works on computer backgammon are posted on-line; see e.g., http://www.bkgm.com/articles/page07.html).

Monte Carlo Tree Search

In the rollout implementation just described, we implicitly assumed that once we reach state x_k, we generate the same large number of trajectories starting from each pair (x_k, u_k), with $u_k \in U(x_k)$, to the end of the horizon. The drawback of this is threefold:

(a) The trajectories may be too long because the horizon length N is large (or infinite, in an infinite horizon context).

(b) Some of the controls u_k may be clearly inferior to others, and may not be worth as much sampling effort.

(c) Some of the controls u_k that appear to be promising, may be worth exploring better through multistep lookahead.

This has motivated variants, generally referred to as *Monte Carlo tree search* (MCTS for short), which aim to trade off computational economy with a hopefully small risk of degradation in performance. Variants of this type involve, among others, early discarding of controls deemed to be inferior based on the results of preliminary calculations, and simulation that is limited in scope (either because of a reduced number of simulation samples, or because of a shortened horizon of simulation, or both).

In particular, a simple remedy for (a) above is to use rollout trajectories of reasonably limited length, with some terminal cost approximation at the end (in an extreme case, the rollout may be skipped altogether, i.e., rollout trajectories have zero length). The terminal cost function may be very simple (such as zero) or may be obtained through some auxiliary calculation. In fact the base policy used for rollout may be the one that provides the terminal cost function approximation, as noted for the rollout-based backgammon algorithm of Example 2.4.2.

A simple but less straightforward remedy for (b) is to use some heuristic or statistical test to discard some of the controls u_k, as soon as this is suggested by the early results of simulation. Similarly, to implement (c) one may use some heuristic to increase the length of lookahead selectively for some of the controls u_k. This is similar to the selective depth lookahead procedure for deterministic rollout that we illustrated in Fig. 2.4.6.

The MCTS approach can be based on sophisticated procedures for implementing and combining the ideas just described. The implementation is often adapted to the problem at hand, but the general idea is to use the interim results of the computation and statistical tests to focus the simulation effort along the most promising directions. Thus to implement MCTS one needs to maintain a lookahead tree, which is expanded as the relevant Q-factors are evaluated by simulation, and which balances *the competing desires of exploitation and exploration* (generate and evaluate controls that seem most promising in terms of performance versus assessing the potential of inadequately explored controls). Ideas that were developed in the context of multiarmed bandit problems have played an important role in the construction of this type of MCTS procedures (see the end-of-chapter references).

Example 2.4.3 (Statistical Tests for Adaptive Sampling with a One-Step Lookahead)

Let us consider a typical one-step lookahead selection strategy that is based on adaptive sampling. We are at a state x_k and we try to find a control $\tilde{u}_k$

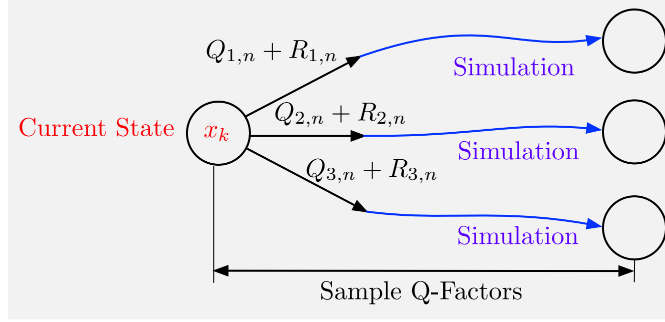

Figure 2.4.8 Illustration of one-step lookahead MCTS at a state x_k. The Q-factor sampled next corresponds to the control i with minimum sum of exploitation index (here taken to be the running average $Q_{i,n}$) and exploration index ($R_{i,n}$, possibly given by the UCB rule).

that minimizes an approximate Q-factor

$$\tilde{Q}_k(x_k, u_k) = E\Big\{g_k(x_k, u, w_k) + \tilde{J}_{k+1}\big(f_k(x_k, u, w_k)\big)\Big\}$$

over $u_k \in U_k(x_k)$, by averaging samples of $\tilde{Q}_k(x_k, u_k)$. We assume that $U_k(x_k)$ contains m elements, which for simplicity are denoted $1, \ldots, m$. At the ℓth sampling period, knowing the outcomes of the preceding sampling periods, we select one of the m controls, say i_ℓ, and we draw a sample of $\tilde{Q}_k(x_k, i_\ell)$, whose value is denoted by S_{i_ℓ}. Thus after the nth sampling period we have an estimate $Q_{i,n}$ of the Q-factor of each control $i = 1, \ldots, m$ that has been sampled at least once, given by

$$Q_{i,n} = \frac{\sum_{\ell=1}^{n} \delta(i_\ell = i) S_{i_\ell}}{\sum_{\ell=1}^{n} \delta(i_\ell = i)}$$

where

$$\delta(i_\ell = i) = \begin{cases} 1 & \text{if } i_\ell = i, \\ 0 & \text{if } i_\ell \neq i. \end{cases}$$

Thus $Q_{i,n}$ is the *empirical mean* of the Q-factor of control i (total sample value divided by total number of samples), assuming that i has been sampled at least once.

After n samples have been collected, with each control sampled at least once, we may declare the control i that minimizes $Q_{i,n}$ as the “best” one, i.e., the one that truly minimizes the Q-factor $Q_k(x_k, i)$. However, there is a positive probability that there is an error: the selected control may not minimize the true Q-factor. In adaptive sampling, roughly speaking, we want to design the sample selection strategy and the criterion to stop the sampling, in a way that keeps the probability of error small (by allocating some sampling effort to all controls), and the number of samples limited (by not wasting samples on controls i that appear inferior based on their empirical mean $Q_{i,n}$).

Intuitively, a good sampling policy will balance at time n the desires of exploitation and exploration (i.e., sample controls that seem most promising, in the sense that they have a small running average $Q_{i,n}$, versus assessing the potential of inadequately explored controls, those i that have been sampled a small number of times). Thus it makes sense to sample next the control i that minimizes the sum

$$T_{i,n} + R_{i,n}$$

of two indexes: an *exploitation index* $T_{i,n}$ and an *exploration index* $R_{i,n}$. Usually the exploitation index is chosen to be the empirical mean $Q_{i,n}$. The exploration index is based an a confidence interval formula and depends on the sample count

$$s_i = \sum_{\ell=1}^{n} \delta(i_\ell = i)$$

of control i. A frequently suggested choice is the UCB rule (upper confidence bound), which sets

$$R_{i,n} = -c\sqrt{\frac{\log n}{s_i}},$$

where c is a positive constant that is selected empirically (some analysis suggests values near $c = \sqrt{2}$, assuming that $Q_{i,n}$ is normalized to take values in the range $[-1, 0]$). The UCB rule, first proposed in the paper [ACF02], has been extensively discussed in the literature both for one-step and for multistep lookahead [where it is called UCT (UCB applied to trees; see [KoS16])]. Its justification is based on probabilistic analyses that relate to the multiarmed bandit problem, and is beyond our scope.

Sampling policies for MCTS with multistep lookahead are based on similar sampling ideas to the case of one-step lookahead. A simulated trajectory is run from a node i of the lookahead tree that minimizes the sum $T_{i,n} + R_{i,n}$ of an exploitation index and an exploration index. There are many schemes of this type, but the details are beyond our scope (see the end-of-chapter references).

A major success has been the use of MCTS in two-player game contexts, such as the AlphaGo computer program (Silver et al. [SHM16]), which performs better than the best humans in the game of Go. This program integrates several of the techniques discussed in this chapter, and in Chapters 3 and 4, including MCTS and rollout using a base policy that is trained off-line using a deep neural network. We will discuss neural network training techniques in Chapter 3. The AlphaZero program, which has performed spectacularly well against humans and other programs in the games of Go and chess (Silver et al. [SHS17]), bears some similarity with AlphaGo, and critically relies on MCTS, but does not use rollout.

Randomized Policy Improvement by MCTS

We have described rollout and MCTS as schemes for policy improvement: start with a base policy, and compute an improved policy based on the results of one-step lookahead or multistep lookahead followed by simulation with the base policy. We have implicitly assumed that both the base policy and the rollout policy are deterministic in the sense that they map each state x_k into a unique control $\tilde{\mu}_k(x_k)$ [cf. Eq. (2.31)]. In some contexts, success has been achieved with *randomized policies*, which map a state x_k to a probability distribution over the set of controls $U_k(x_k)$, rather than mapping onto a single control. In particular, the AlphaGo and AlphaZero programs use MCTS to generate and use for training purposes randomized policies, which specify at each board position the probabilities with which the various moves are selected.

A randomized policy can be used as a base policy in exactly the same way as a deterministic policy: for a given state x_k, we just generate sample trajectories and associated sample Q-factors, starting from each leaf-state of the lookahead tree that is rooted at x_k. We then average the corresponding Q-factor samples. However, the rollout/improved policy, as we have described it, is a deterministic policy, i.e., it applies at x_k the control $\tilde{\mu}_k(x_k)$ that is "best" according to the results of the rollout [cf. Eq. (2.31)]. Still, however, if we wish to generate an improved policy that is randomized, we can simply change the probabilities of different controls in the direction of the deterministic rollout policy. This can be done by increasing the probability of the "best" control $\tilde{\mu}_k(x_k)$ from its base policy level, while decreasing the probabilities of the other controls.

The use of MCTS provides a related method to "improve" a randomized policy. In the process of adaptive simulation that is used in MCTS, we generate *frequency counts* of the different controls, i.e., the proportion of rollout trajectories associated each $u_k \in U_k(x_k)$. We can then obtain the rollout randomized policy by moving the probabilities of the base policy in the direction suggested by the frequency counts, i.e., increase the probability of high-count controls and reduce the probability of the others. This type of policy improvement is reminiscent of gradient-type methods, and has been successful in a number of applications; see Section 4.11 for further discussion, and the end-of-chapter references for such policy improvement implementations in AlphaGo, AlphaZero, and other application contexts.

Variance Reduction in Rollout - Comparing Advantages

When using simulation, sampling is often organized to effect *variance reduction*. By this we mean that for a given problem, the collection and use of samples is structured so that the variance of the simulation error is made smaller, with the same amount of simulation effort. There are several

methods of this type for which we refer to textbooks on simulation (see, e.g., Ross [Ros12], Rubinstein and Kroese [RuK17]).

In this section we discuss a method to reduce the effects of the simulation error in the calculation of the Q-factors in the context of rollout. The key idea is that the selection of the rollout control depends on the values of the Q-factor differences

$$\tilde{Q}_k(x_k, u_k) - \tilde{Q}_k(x_k, \hat{u}_k)$$

for all pairs of controls $(u_k, \hat{u}_k)$. These values must be computed accurately, so that the controls u_k and $\hat{u}_k$ can be accurately compared. On the other hand, the simulation/approximation errors in the computation of the individual Q-factors $\tilde{Q}_k(x_k, u_k)$ may be magnified through the preceding differencing operation.

An alternative approach is possible in the case where the probability distribution of each disturbance w_k does not depend on x_k and u_k. In this case, we may approximate by simulation the Q-factor difference $\tilde{Q}_k(x_k, u_k) - \tilde{Q}_k(x_k, \hat{u}_k)$ by sampling the difference

$$C_k(x_k, u_k, \mathbf{w}_k) - C_k(x_k, \hat{u}_k, \mathbf{w}_k),$$

where $\mathbf{w}_k = (w_k, w_{k+1}, \ldots, w_{N-1})$ and

$$C_k(x_k, u_k, \mathbf{w}_k) = g_N(x_N) + g_k(x_k, u_k, w_k) + \sum_{i=k+1}^{N-1} g_i\big(x_i, \mu_i(x_i), w_i\big),$$

where $\{\mu_{k+1}, \ldots, \mu_{N-1}\}$ is the tail portion of the base policy.

This approximation may be far more accurate than the one obtained by differencing independent samples $C_k(x_k, u_k, \mathbf{w}_k)$ and $C_k(x_k, \hat{u}_k, \hat{\mathbf{w}}_k)$. Indeed, by introducing the zero mean sample errors

$$D_k(x_k, u_k, \mathbf{w}_k) = C_k(x_k, u_k, \mathbf{w}_k) - \tilde{Q}_k(x_k, u_k),$$

it can be seen that the variance of the error in estimating $\tilde{Q}_k(x_k, u_k) - \tilde{Q}_k(x_k, \hat{u}_k)$ with the former method will be smaller than with the latter method if and only if

$$E_{\mathbf{w}_k, \hat{\mathbf{w}}_k}\Big\{\big|D_k(x_k, u_k, \mathbf{w}_k) - D_k(x_k, \hat{u}_k, \hat{\mathbf{w}}_k)\big|^2\Big\} > E_{\mathbf{w_k}}\Big\{\big|D_k(x_k, u_k, \mathbf{w}_k) - D_k(x_k, \hat{u}_k, \mathbf{w}_k)\big|^2\Big\},$$

or equivalently

$$E\big\{D_k(x_k, u_k, \mathbf{w}_k) D_k(x_k, \hat{u}_k, \mathbf{w}_k)\big\} > 0; \tag{2.32}$$

i.e., if and only if the correlation between the errors $D_k(x_k, u_k, \mathbf{w}_k)$ and $D_k(x_k, \hat{u}_k, \mathbf{w}_k)$ is positive. A little thought should convince the reader that this property is likely to hold in many types of problems. Roughly speaking, the relation (2.32) holds if changes in the value of u_k (at the first stage) have little effect on the value of the error $D_k(x_k, u_k, \mathbf{w}_k)$ relative to the effect induced by the randomness of $\mathbf{w}_k$. To see this, suppose that there exists a scalar $\gamma < 1$ such that, for all x_k, u_k, and $\hat{u}_k$, there holds

$$E\left\{\big|D_k(x_k, u_k, \mathbf{w}_k) - D_k(x_k, \hat{u}_k, \mathbf{w}_k)\big|^2\right\} \leq \gamma E\left\{\big|D_k(x_k, u_k, \mathbf{w}_k)\big|^2\right\}. \tag{2.33}$$

Then we have

$$\begin{aligned} D_k(x_k, u_k, \mathbf{w}_k)&D_k(x_k, \hat{u}_k, \mathbf{w}_k) \\ &= \big|D_k(x_k, u_k, \mathbf{w}_k)\big|^2 \\ &\quad + D_k(x_k, u_k, \mathbf{w}_k)\big(D_k(x_k, \hat{u}_k, \mathbf{w}_k) - D_k(x_k, u_k, \mathbf{w}_k)\big) \\ &\geq \big|D_k(x_k, u_k, \mathbf{w}_k)\big|^2 \\ &\quad - \big|D_k(x_k, u_k, \mathbf{w}_k)\big| \cdot \big|D_k(x_k, \hat{u}_k, \mathbf{w}_k) - D_k(x_k, u_k, \mathbf{w}_k)\big|, \end{aligned}$$

from which we obtain

$$\begin{aligned} E\big\{D_k(x_k,&u_k, \mathbf{w}_k)D_k(x_k, \hat{u}_k, \mathbf{w}_k)\big\} \\ &\geq E\left\{\big|D_k(x_k, u_k, \mathbf{w}_k)\big|^2\right\} \\ &\quad - E\Big\{\big|D_k(x_k, u_k, \mathbf{w}_k)\big| \cdot \big|D_k(x_k, \hat{u}_k, \mathbf{w}_k) - D_k(x_k, u_k, \mathbf{w}_k)\big|\Big\} \\ &\geq E\left\{\big|D_k(x_k, u_k, \mathbf{w}_k)\big|^2\right\} - \frac{1}{2}E\left\{\big|D_k(x_k, u_k, \mathbf{w}_k)\big|^2\right\} \\ &\quad - \frac{1}{2}E\Big\{\big|D_k(x_k, \hat{u}_k, \mathbf{w}_k) - D_k(x_k, u_k, \mathbf{w}_k)\big|^2\Big\} \\ &\geq \frac{1-\gamma}{2}E\left\{\big|D_k(x_k, u_k, \mathbf{w}_k)\big|^2\right\}, \end{aligned}$$

where for the first inequality we use the generic relation $ab \geq a^2 - |a| \cdot |b-a|$ for two scalars a and b, for the second inequality we use the generic relation $|a| \cdot |b| \geq -\frac{1}{2}(a^2 + b^2)$ for two scalars a and b, and for the third inequality we use Eq. (2.33).

Thus, under the assumption (2.33) and the assumption

$$E\left\{\big|D_k(x_k, u_k, \mathbf{w}_k)\big|^2\right\} > 0,$$

the condition (2.32) holds and guarantees that by averaging cost difference samples rather than differencing (independently obtained) averages of cost samples, the simulation error variance decreases.

Let us finally note the potential benefit of using Q-factor differences in contexts other than rollout. In particular when approximating Q-factors $Q_k(x_k, u_k)$ using parametric architectures (Section 3.4 in the next chapter), it may be important to approximate and compare instead the differences

$$A_k(x_k, u_k) = Q_k(x_k, u_k) - \min_{u_k \in U_k(x_k)} Q_k(x_k, u_k).$$

The function $A_k(x_k, u_k)$ is also known as the *advantage of the pair* (x_k, u_k), and can serve just as well as $Q_k(x_k, u_k)$ for the purpose of comparing controls, but may work better $Q_k(x_k, u_k)$ in the presence of approximation errors. This question is discussed further in Section 3.4.

2.5 ON-LINE ROLLOUT FOR DETERMINISTIC INFINITE-SPACES PROBLEMS - OPTIMIZATION HEURISTICS

We have considered so far discrete-spaces applications of rollout, where the relevant Q-factors at each state x_k are evaluated by simulation and compared by exhaustive comparison. To implement this approach in a continuous-spaces setting, the control constraint set must first be discretized, which is often inconvenient and ineffective. In this section we will discuss an alternative approach for deterministic problems that can deal with an infinite number of controls and Q-factors at x_k without discretization. The idea is to *use a base heuristic that involves a continuous optimization*, and to rely on a nonlinear programming method to solve the corresponding lookahead optimization problem.

To get a sense of the basic idea, consider the one-step lookahead rollout minimization

$$\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k), \tag{2.34}$$

where $\tilde{Q}_k(x_k, u_k)$ is the approximate Q-factor

$$\tilde{Q}_k(x_k, u_k) = g_k(x_k, u_k) + H_{k+1}\big(f_k(x_k, u_k)\big), \tag{2.35}$$

with $H_{k+1}(x_{k+1})$ being the cost of the base heuristic starting from state x_{k+1} [cf. Eq. (2.25)]. Suppose that we have a differentiable closed-form expression for H_{k+1}, and the functions g_k and f_k are known and are differentiable with respect to u_k. Then the Q-factor $\tilde{Q}_k(x_k, u_k)$ of Eq. (2.35) is also differentiable with respect to u_k, and its minimization (2.34) may be addressed with one of the many gradient-based methods that are available for differentiable unconstrained and constrained optimization.

The preceding approach requires that the heuristic cost $H_{k+1}(x_{k+1})$ be available in closed form, which is highly restrictive, but this difficulty can

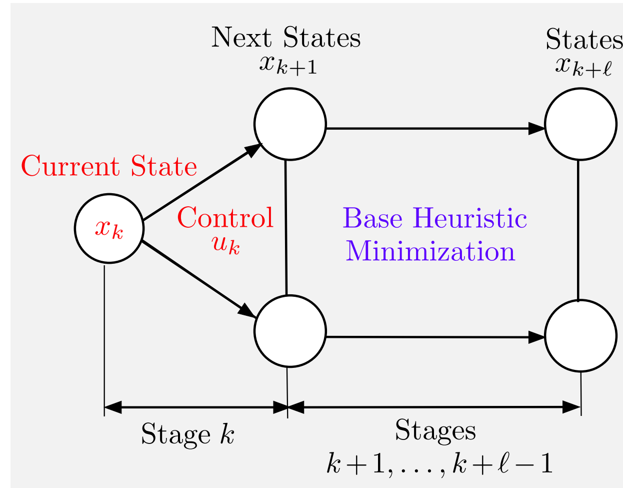

Figure 2.5.1 Schematic illustration of rollout for a deterministic problem with infinite control spaces. The base heuristic is an $(\ell-1)$-stage deterministic optimal control problem, which together with the kth stage minimization over $u_k \in U_k(x_k)$, seamlessly forms an ℓ-stage continuous spaces optimal control/nonlinear programming problem that starts at state x_k.

be circumvented by *using a base heuristic that is itself based on multistep optimization*. In particular, suppose that $H_{k+1}(x_{k+1})$ is the optimal cost of some $(\ell-1)$-stage deterministic optimal control problem that is related to the original problem. Then the rollout algorithm (2.34)-(2.35) can be implemented by solving the *ℓ-stage deterministic optimal control problem, which seamlessly concatenates the first stage minimization over u_k [cf. Eq. (2.34)], with the $(\ell-1)$-stage minimization of the base heuristic*; see Fig. 2.5.1. This ℓ-stage problem may be solvable on-line by standard continuous spaces nonlinear programming or optimal control methods. An important example of methods of this type arises in control system design and is discussed next.

2.5.1 Model Predictive Control

We will consider a classical control problem, where the objective is to keep the state of a deterministic system close to the origin of the state space or close to a given trajectory. This problem has a long history, and has been addressed by a variety of sophisticated methods. Starting in the late 50s and early 60s, approaches based on state variable system representations and optimal control became popular. The linear-quadratic approach, which we have illustrated by example in Section 1.3.7, was developed during this period, and is still used extensively. Unfortunately, however, linear-

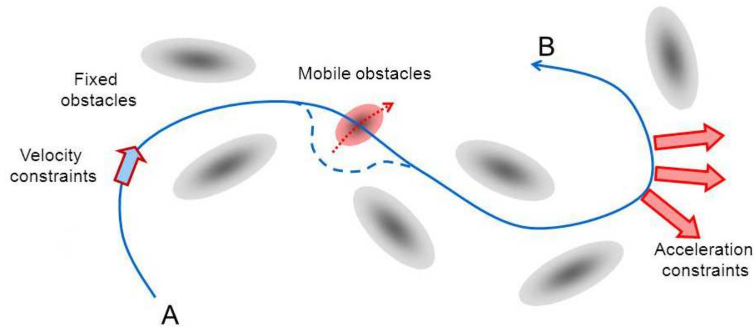

Figure 2.5.2 Illustration of constrained motion of a robot from point A to point B. There are state (position/velocity) constraints, and control (acceleration) constraints. When there are mobile obstacles, the state constraints may change unpredictably, necessitating on-line replanning.

quadratic models are often not satisfactory. There are two main reasons for this:

(a) The system may be nonlinear, and it may be inappropriate to use for control purposes a model that is linearized around the desired point or trajectory.

(b) There may be control and/or state constraints, which are not handled adequately through quadratic penalty terms in the cost function. For example, the motion of a robot may be constrained by the presence of obstacles and hardware limitations (see Fig. 2.5.2). The solution obtained from a linear-quadratic model may not be suitable for such a problem, because quadratic penalties treat constraints "softly" and may produce trajectories that violate the constraints.

These inadequacies of the linear-quadratic model have motivated a methodology, called *model predictive control* (MPC for short), which combines elements of several ideas that we have discussed so far: multistep lookahead, rollout with infinite control spaces, and certainty equivalence. Aside from resolving the difficulty with infinitely many Q-factors at x_k, while dealing adequately with state and control constraints, MPC is well-suited for on-line replanning, like all rollout methods.

We will focus primarily on the most common form of MPC, where the system is either deterministic, or else it is stochastic, but it is replaced with a deterministic version by using typical values in place of the uncertain quantities, similar to the certainty equivalent control approach. Moreover we will consider the case where the objective is to keep the state close to the origin; this is called the *regulation problem*. Similar approaches have been developed for the problem of maintaining the state of a nonstationary

system along a given state trajectory, and also, with appropriate modifications, to control problems involving disturbances.

In particular, we will consider a deterministic system

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

whose state x_k and control u_k are vectors that consist of a finite number of scalar components. The cost per stage is assumed nonnegative

$$g_k(x_k, u_k) \geq 0, \qquad \text{for all } (x_k, u_k),$$

(e.g., a quadratic cost). We impose state and control constraints

$$x_k \in X_k, \qquad u_k \in U_k(x_k), \qquad k = 0, 1, \ldots.$$

We also assume that the system can be kept at the origin at zero cost, i.e.,

$$f_k(0, \overline{u}_k) = 0, \quad g_k(0, \overline{u}_k) = 0 \qquad \text{for some control } \overline{u}_k \in U_k(0). \tag{2.36}$$

For a given initial state $x_0 \in X_0$, we want to obtain a sequence $\{u_0, u_1, \ldots\}$ such that the states and controls of the system satisfy the state and control constraints with small total cost.

The MPC Algorithm

Let us describe the MPC algorithm for the deterministic problem just described. At the current state x_k:

(a) MPC solves an ℓ-step lookahead version of the problem, which requires that $x_{k+\ell} = 0$.

(b) If $\{\tilde{u}_k, \ldots, \tilde{u}_{k+\ell-1}\}$ is the optimal control sequence of this problem, MPC applies $\tilde{u}_k$ and discards the other controls $\tilde{u}_{k+1}, \ldots, \tilde{u}_{k+\ell-1}$.

(c) At the next stage, MPC repeats this process, once the next state x_{k+1} is revealed.

In particular, at the typical stage k and state $x_k \in X_k$, the MPC algorithm solves an ℓ-stage optimal control problem involving the same cost function and the requirement $x_{k+\ell} = 0$. This is the problem

$$\min_{u_i,\, i=k,\ldots,k+\ell-1} \sum_{i=k}^{k+\ell-1} g_i(x_i, u_i), \tag{2.37}$$

subject to the system equation constraints

$$x_{i+1} = f_i(x_i, u_i), \qquad i = k, \ldots, k+\ell-1,$$

the state and control constraints

$$x_i \in X_i, \qquad u_i \in U_i(x_i), \qquad i = k, \ldots, k + \ell - 1,$$

and the terminal state constraint

$$x_{k+\ell} = 0.$$

Let $\{\tilde{u}_k, \tilde{u}_{k+1}, \ldots, \tilde{u}_{k+\ell-1}\}$ be a corresponding optimal control sequence. The MPC algorithm applies at stage k the first component $\tilde{u}_k$ of this sequence, and discards the remaining components; see Fig. 2.5.3.†

To guarantee that there is an integer ℓ such that the preceding MPC algorithm is feasible, we assume the following.

Constrained Controllability Condition

There exists an integer $\ell > 1$ such that for every initial state $x_k \in X_k$, we can find a sequence of controls $u_k, \ldots, u_{k+\ell-1}$ that drive to 0 the state $x_{k+\ell}$ of the system at time $k + \ell$, while satisfying all the intermediate state and control constraints

$$u_k \in U_k(x_k),\ x_{k+1} \in X_{k+1}, \ldots,\\ x_{k+\ell-1} \in X_{k+\ell-1},\ u_{k+\ell-1} \in U_{k+\ell-1}(x_{k+\ell-1}).$$

Finding an integer ℓ that satisfies the constrained controllability condition is an important issue to which we will return later.† Generally the constrained controllability condition tends to be satisfied if the control constraints are not too stringent, and the state constraints do not allow a large deviation from the origin. In this case not only we can implement MPC, but also the resulting closed-loop system will tend to be stable; see the following discussion of stability, and Example 2.5.2.

Note that the actual state trajectory produced by MPC may never reach the origin (see the subsequent Example 2.5.1). This is because we use only the first control $\tilde{u}_k$ of the kth stage sequence $\{\tilde{u}_k, \tilde{u}_{k+1}, \ldots, \tilde{u}_{k+\ell-1}\}$,

---

† In the case, where we want the system to follow a given nominal trajectory, rather than stay close to the origin, we should modify the MPC optimization to impose as a terminal constraint that the state $x_{k+\ell}$ should be a point on the nominal trajectory (instead of $x_{k+\ell} = 0$). We should also change the cost function to reflect a penalty for deviating from the given trajectory.

† In the case where we want the system to follow a given nominal trajectory, rather than stay close to the origin, we may want to use a time-dependent lookahead length ℓ_k, to exercise tighter control over critical parts of the nominal trajectory.

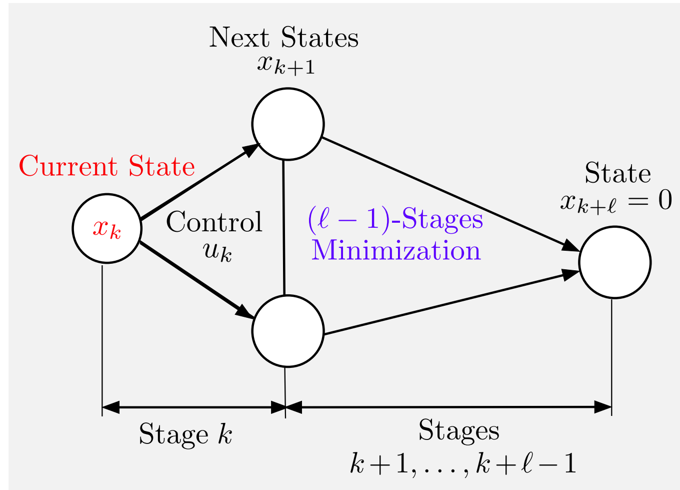

Figure 2.5.3 Illustration of the problem solved by MPC at state x_k. We minimize the cost function over the next ℓ stages while imposing the requirement that $x_{k+\ell} = 0$. We then apply the first control of the optimizing sequence.

which aims at $x_{k+\ell} = 0$. At the next stage $k+1$ the control generated by MPC may be different than $\tilde{u}_{k+1}$, because it will aim one step further to the terminal condition $x_{k+\ell+1} = 0$.

To make the connection of MPC with rollout, we note that *the one-step lookahead function $\tilde{J}$ implicitly used by MPC [cf. Eq. (2.37)] is the cost-to-go function of a certain base heuristic.* This is the heuristic that drives to 0 the state after $\ell-1$ stages (*not ℓ stages*) and keeps the state at 0 thereafter, while observing the state and control constraints, and minimizing the associated $(\ell-1)$ stages cost, in the spirit of our earlier discussion; cf. Fig. 2.5.1.

Sequential Improvement Property and Stability Analysis

It turns out that the base heuristic just described is sequentially improving, so MPC has a cost improvement property, of the type discussed in Section 2.4.1. To see this, let us denote by $\hat{J}_k(x_k)$ the optimal cost of the ℓ-stage problem solved by MPC when at a state $x_k \in X_k$. Let also $H_k(x_k)$ and $H_{k+1}(x_{k+1})$ be the optimal heuristic costs of the corresponding $(\ell-1)$-stage optimization problems that start at x_k and x_{k+1}, and drive the states $x_{k+\ell-1}$ and $x_{k+\ell}$, respectively, to 0. Thus, by the principle of optimality, we have the DP equation

$$\hat{J}_k(x_k) = \min_{u_k \in U_k(x_k)} \big[g_k(x_k,u_k) + H_{k+1}\big(f_k(x_k.u_k)\big)\big].$$

Since having one less stage at our disposal to drive the state to 0 cannot decrease the optimal cost, we have

$$\hat{J}_k(x_k) \leq H_k(x_k).$$

By combining the preceding two relations, we obtain

$$\min_{u_k \in U_k(x_k)} \big[g_k(x_k, u_k) + H_{k+1}\big(f_k(x_k.u_k)\big)\big] \leq H_k(x_k), \tag{2.38}$$

which is the sequential improvement condition for the base heuristic [cf. Eq. (2.29)].†

Often the primary objective in MPC, aside from fulfilling the state and control constraints, is to obtain a stable closed-loop system, i.e., a system that naturally tends to stay close to the origin. This is typically expressed adequately by the requirement of a finite cost over an infinite number of stages:

$$\sum_{k=0}^{\infty} g_k(x_k, u_k) < \infty, \tag{2.39}$$

where $\{x_0, u_0, x_1, u_1, \ldots\}$ is the state and control sequence generated by MPC.

We will now show that the stability condition (2.39) is satisfied by the MPC algorithm. Indeed, from the sequential improvement condition (2.38), we have

$$g_k(x_k, u_k) + H_{k+1}(x_{k+1}) \leq H_k(x_k), \qquad k = 0, 1, \ldots. \tag{2.40}$$

Adding this relation for all k in a range $[0, K]$, where $K = 0, 1, \ldots$, we obtain

$$H_{K+1}(x_{K+1}) + \sum_{k=0}^{K} g_k(x_k, u_k) \leq H_0(x_0).$$

Since $H_{K+1}(x_{K+1}) \geq 0$, it follows that

$$\sum_{k=0}^{K} g_k(x_k, u_k) \leq H_0(x_0), \qquad K = 0, 1, \ldots, \tag{2.41}$$

and taking the limit as $K \to \infty$,

$$\sum_{k=0}^{\infty} g_k(x_k, u_k) \leq H_0(x_0) < \infty,$$

[$H_0(x_0)$ is finite because the transfer from x_0 to $x_\ell = 0$ is feasible by the constrained controllability condition]. We have thus verified the stability condition (2.39).

† Note that the base heuristic is not sequentially consistent, as it fails to satisfy the definition given in Section 2.4.1 (see the subsequent Example 2.5.1).

Example 2.5.1

Consider a scalar linear system and a quadratic cost

$$x_{k+1} = x_k + u_k, \qquad g_k(x_k, u_k) = x_k^2 + u_k^2,$$

where the state and control constraints are

$$x_k \in X_k = \big\{x \mid |x| \leq 1.5\big\}, \qquad u_k \in U_k(x_k) = \big\{u \mid |u| \leq 1\big\}.$$

We apply the MPC algorithm with $\ell = 2$. For this value of ℓ, the constrained controllability assumption is satisfied, since the 2-step sequence of controls

$$u_0 = -\mathrm{sgn}(x_0), \qquad u_1 = -x_1 = -x_0 + \mathrm{sgn}(x_0)$$

drives the state x_2 to 0, for any x_0 with $|x_0| \leq 1.5$.

At state $x_k \in X_k$, MPC minimizes the two-stage cost

$$x_k^2 + u_k^2 + (x_k + u_k)^2 + u_{k+1}^2,$$

subject to the control constraints

$$|u_k| \leq 1, \qquad |u_{k+1}| \leq 1,$$

and the state constraints

$$|x_{k+1}| \leq 1.5, \qquad x_{k+2} = x_k + u_k + u_{k+1} = 0.$$

This is a quadratic programming problem, which can be solved with available software, and in this case analytically, because of its simplicity. In particular, it can be verified that the minimization yields

$$\tilde{u}_k = -\frac{2}{3}x_k, \qquad \tilde{u}_{k+1} = -(x_k + \tilde{u}_k).$$

Thus the MPC algorithm selects $\tilde{u}_k = -\frac{2}{3}x_k$, which results in the closed-loop system

$$x_{k+1} = \frac{1}{3}x_k, \qquad k = 0, 1, \ldots.$$

Note that while this closed-loop system is stable, its state is never driven to 0 if started from $x_0 \neq 0$. Moreover, it is easily verified that the base heuristic is not sequentially consistent. For example, starting from $x_k = 1$, the base heuristic generates the sequence

$$\left\{x_k = 1, u_k = -\frac{2}{3}, x_{k+1} = \frac{1}{3}, u_{k+1} = -\frac{1}{3}, x_{k+2} = 0, u_{k+2} = 0, \ldots\right\},$$

while starting from the next state $x_{k+1} = \frac{1}{3}$ it generates the sequence

$$\left\{x_{k+1} = \frac{1}{3}, u_{k+1} = -\frac{2}{9}, x_{k+2} = \frac{1}{9}, u_{k+2} = -\frac{1}{9}, x_{k+3} = 0, u_{k+3} = 0, \ldots\right\},$$

so the sequential consistency condition of Section 2.4.1 is violated.

Regarding the choice of the horizon length ℓ for the MPC calculations, note that if the constrained controllability assumption is satisfied for some value of ℓ, it is also satisfied for all larger values of ℓ. This argues for a large value of ℓ. On the other hand, the optimal control problem solved at each stage by MPC becomes larger and hence more difficult as ℓ increases. Thus, the horizon length is usually chosen on the basis of some experimentation: first ensure that ℓ is large enough for the constrained controllability assumption to hold, and then by further experimentation to ensure overall satisfactory performance.

2.5.2 Target Tubes and the Constrained Controllability Condition

We now return to the constrained controllability condition, which asserts that the state constraint sets X_k are such that starting from anywhere within it, it is possible to drive to 0 the state of the system within some number of steps ℓ, while staying within X_m at each intermediate step $m = k+1,\ldots,m = k+\ell-1$. Unfortunately, this assumption masks some major complications. In particular, the control constraint set may not be sufficiently rich to compensate for natural instability tendencies of the system. As a result it may be impossible to keep the state within X_k over a sufficiently long period of time, something that may be viewed as a form of instability. Here is an example.

Example 2.5.2

Consider the scalar linear system

$$x_{k+1} = 2x_k + u_k,$$

which is unstable, and the control constraint

$$|u_k| \leq 1.$$

Then if $0 \leq x_0 < 1$, it can be seen that by using the control $u_0 = -1$, the next state satisfies,

$$x_1 = 2x_0 - 1 < x_0,$$

and is closer to 0 than the preceding state x_0. Similarly, using controls $u_k = -1$, every subsequent state x_{k+1} will get closer to 0 that x_k. Eventually, after a sufficient number of steps $\bar{k}$, with controls $u_k = -1$ for $k < \bar{k}$, the state $x_{\bar{k}}$ will satisfy

$$0 \leq x_{\bar{k}} \leq \frac{1}{2}.$$

Once this happens, the feasible control $u_{\bar{k}} = -2x_{\bar{k}}$ will drive the state $x_{\bar{k}+1}$ to 0.

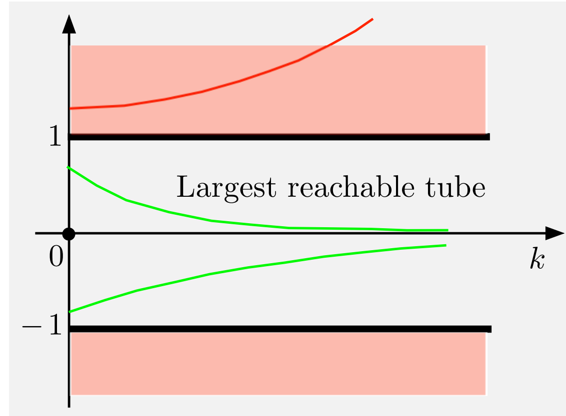

Figure 2.5.4 Illustration of state trajectories under MPC for Example 2.5.2. If the initial state lies within the set $(-1,1)$ the constrained controllability condition is satisfied for sufficiently large ℓ, and the MPC algorithm yields a stable controller. If the initial state lies outside this set, MPC cannot be implemented because the constrained controllability condition fails to holds. Moreover, the system is unstable starting from such an initial state. In this example, the largest reachable tube is $\{X, X, \ldots\}$ with $X = \big\{x \mid |x| \leq 1\big\}$.

Similarly, when $-1 < x_0 \leq 0$, by applying control $u_k = 1$ for a sufficiently large number of steps $\bar{k}$, the state $x_{\bar{k}}$ will be driven into the region $[-1/2, 0]$, and then the feasible control $u_{\bar{k}} = -2x_{\bar{k}}$ will drive the state $x_{\bar{k}+1}$ to 0.

Suppose now that the control constraint is $|u_k| \leq 1$ and the state constraint is of the form $X_k = [-\beta, \beta]$ for all k, and let us explore what happens for different values of β. The preceding discussion shows that if $0 < \beta < 1$ the constrained controllability assumption is satisfied, and for every state initial state $x_0 \in X_0$ the state can be kept within X_k and can be driven to 0 in a finite number ℓ of steps. The number ℓ depends on β, and in particular if $0 < \beta < 1/2$, ℓ can be taken equal to 1.

On the other hand, if $\beta \geq 1$, it is impossible to drive the state to 0 from every initial state $x_0 \in [1, \beta]$ without violating the constraint $|u_k| \leq 1$, so the constrained controllability assumption is violated. In fact if the initial state satisfies $|x_0| > 1$, the state trajectory diverges in the sense that $|x_k| \to \infty$ for any control sequence that satisfies the constraint $|u_k| \leq 1$; see Fig. 2.5.4. Thus if $\beta \geq 1$, either a larger control constraint set or an initial condition that is close to 0, or both, are needed to satisfy the constrained controllability condition and to improve the stability properties of the closed-loop system.

The critical construction in the preceding example is to identify sets of states $\overline{X}_k \subset X_k$ such that starting from within $\overline{X}_k$ we are guaranteed to stay within the "tube" $\{\overline{X}_{k+1}, \overline{X}_{k+2}, \ldots, \overline{X}_N\}$ for all subsequent times with appropriate choices of control. Tubes of this type can serve as state constraints in MPC. Moreover, when the sets $\overline{X}_k$ are bounded, *a guarantee*

that the state can stay within the tube amounts to a form of closed-loop stability guarantee. In the remainder if this section, we will address the issue of how such tubes can be constructed.

Formally, a tube $\{\overline{X}_0, \overline{X}_1, \ldots, \overline{X}_N\}$ is just a sequence of subsets with $\overline{X}_k \subset X_k$ for all $k = 0, \ldots, N$. The tube is called *reachable* if it has the property that for every k and $x_k \in \overline{X}_k$ there exists a $u_k \in U_k(x_k)$ such that $f_k(x_k, u_k) \in \overline{X}_{k+1}$. A reachable tube was also called an *effective target tube* in [Ber71], and for simplicity it will be called a *target tube* in this section; the latter name is widely used in the current literature.† If the original tube of state constraints $\{X_0, X_1, \ldots, X_N\}$ is not reachable, the constrained controllability condition cannot be satisfied, since then there will be states outside the tube starting from which we can never reenter the tube. In this case, it is necessary to compute a reachable tube to use as a set of state constraints in place of the original tube $\{X_0, X_1, \ldots, X_N\}$. Thus obtaining a reachable tube is a prerequisite towards satisfying the constrained controllability assumption, and serves as the first step in the analysis and design of MPC schemes with state constraints.

Given an N-stage deterministic problem with state constraints $x_k \in X_k$, $k = 0, \ldots, N$, we can obtain a reachable tube $\{\overline{X}_0, \overline{X}_1, \ldots, \overline{X}_N\}$ by a recursive algorithm that starts with

$$\overline{X}_N = X_N,$$

and generates $\overline{X}_k$, $k = 0, \ldots, N-1$, going backwards,

$$\overline{X}_k = \big\{x_k \in X_k \mid \text{for some } u_k \in U_k(x_k) \text{ we have } f_k(x_k, u_k) \in \overline{X}_{k+1}\big\}.$$

Generally, it is difficult to compute the sets $\overline{X}_k$ of the reachable tube, but algorithms that produce inner approximations have been constructed. The author's thesis [Ber71] and subsequent papers [BeR71], [Ber72], [BeR73], [Ber07], gave inner ellipsoidal approximations for both finite and infinite horizon problems with perfect and partial state information, which involve linear systems and ellipsoidal constraints. Other authors have developed polyhedral approximations; see the textbook by Borelli, Bemporad, and Morari [BBM17].

---

† The concept of target tubes and reachability in discrete-time systems was introduced in the author's 1971 Ph.D. thesis ([Ber71], available on-line) and associated papers [BeR71], [Ber72], [BeR73], where various methods for constructing target tubes were given, for both deterministic and minimax/game problems, including easily characterized ellipsoids for linear systems. Reachability can be defined for finite horizon as well as infinite horizon problems. Reachability concepts were studied by several authors much later, particularly in the context of MPC, and have been generalized to continuous-time systems (see the end-of-chapter references).

Example 2.5.2 (continued)

Here if the state constraints are

$$X_k = \Big\{x_k \mid |x_k| \leq 1\Big\}, \qquad k = 0, 1, \ldots, N, \tag{2.42}$$

the tube $\{X_k \mid k = 0, \ldots, N\}$ is reachable for any N. However, this is not true for the constraint sets

$$X_k = \Big\{x_k \mid |x_k| \leq 2\Big\}, \qquad k = 0, 1, \ldots, N.$$

For example for $x_0 = 2$ the next state $x_1 = 2x_0 + u_k = 4 + u_k$ will not satisfy $|x_1| \leq 2$ for any of the feasible controls u_k with $|u_k| \leq 1$. Thus it is necessary to replace the tube of original constraints $\{X_k \mid k = 0, \ldots, N\}$ with a reachable tube $\{\overline{X}_k \mid k = 0, \ldots, N\}$. It can be verified that the largest tube that is reachable for any value of N is the one with

$$\overline{X}_k = \Big\{x_k \mid |x_k| \leq 1\Big\}, \qquad k = 0, 1, \ldots, N.$$

Calculating a reachable tube is relatively easy for one-dimensional problems, but becomes complicated for multidimensional problems, where approximations are required in general.

Finally, let us consider the case of the quadratic cost per stage

$$g_k(x_k, u_k) = x_k^2 + u_k^2,$$

and MPC implemented with $\ell = 2$. As noted earlier, in order for the MPC minimizations to be feasible for $\ell = 2$, the initial condition must satisfy $|x_0| \leq 1$. A calculation very similar to the one of Example 2.5.1 shows that MPC applies at time k the control $\tilde{u}_k = -(5/3)x_k$. The state of the closed-loop system evolves according to

$$x_{k+1} = \frac{1}{3}x_k,$$

and tends to 0 asymptotically.

Note that there is a subtle difference between reachability of the tube $\{X_k \mid k = 0, \ldots, N\}$ and satisfying the constrained controllability condition. The latter implies the former, but the converse is not true, as the preceding example illustrates. In particular, if X_k is given by Eq. (2.42), the tube is reachable, but the constrained controllability assumption is not satisfied because starting at the boundary of X_k, we cannot drive the state to 0 in any number of steps, using controls u_k with $|u_k| \leq 1$. By contrast, if we remove from X_k the boundary points 1 and -1, the tube is still reachable while the constrained controllability condition is satisfied. Generally, except for boundary effects of this type, tube reachability typically implies the constrained controllability condition.

2.5.3 Variants of Model Predictive Control

The MPC scheme that we have described is just the starting point for a broad methodology with many variations, which often relate to the suboptimal control methods that we have discussed so far in this chapter. For example, in the problem solved by MPC at each stage, instead of the requirement of driving the system state to 0 in ℓ steps, one may use a large penalty for the state being nonzero after ℓ steps. Then, the preceding analysis goes through, as long as the terminal penalty is chosen so that the sequential improvement condition (2.38) is satisfied.

In another variant, instead of aiming to drive the state to 0 after ℓ steps, one aims to reach a sufficiently small neighborhood of the origin, within which a stabilizing controller, designed by other methods, may be used.

We finally mention variants of MPC methods, which combine rollout and terminal cost function approximation, and which can deal with uncertainty and system disturbances. A method of this type will be described in Section 4.6.1 in the context of infinite horizon problems, but can be adapted to finite horizon problems as well; see also the end-of-chapter references.

As an illustration, let us provide an example of a scheme that combines MPC with certainty equivalent control ideas (cf. Section 2.3.2).

Example 2.5.3 (MPC for Stochastic Systems and Certainty Equivalent Approximation)

Consider the stochastic system

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

with the expected cost of a policy $\pi = \{\mu_0, \ldots, \mu_{N-1}\}$ starting at x_0 defined by

$$J_\pi(x_0) = E\left\{ g_N(x_N) + \sum_{k=0}^{N-1} g\big(x_k, \mu_k(x_k), w_k\big) \right\};$$

cf. the framework of Section 1.2. We assume that for all k, there are state and control constraints of the form

$$x_k \in X_k, \qquad u_k \in U_k(x_k),$$

and that the stochastic disturbances w_k take values within some known set W_k.

An important characteristic of this problem is that a policy must maintain reachability of the tube $\{X_0, X_1, \ldots\}$, even under worst-case disturbance values. For this it is necessary that for each state $x_k \in X_k$, the control u_k is chosen from within the subset $\tilde{U}_k(x_k)$ given by

$$\tilde{U}_k(x_k) = \big\{ u_k \in U_k(x_k) \mid f(x_k, u_k, w_k) \in X_k, \text{ for all } w_k \in W_k \big\}.$$

We assume that $\tilde{U}_k(x_k)$ is nonempty for all $x_k \in X_k$ and is somehow available. This is not automatically satisfied; similar to the deterministic case discussed earlier, the target tube $\{X_0, X_1, \ldots\}$ must be properly constructed using reachability methods, the sets $U_k(x_k)$ must be sufficiently "rich" to ensure that this is possible, and the sets $\tilde{U}_k(x_k)$ must be computed.

We will now describe a rollout/MPC method that generalizes the one given earlier for deterministic problems. It satisfies the state and control constraints, and uses assumed certainty equivalence to define the base policy over $\ell - 1$ steps, where $\ell > 1$ is some integer. In particular, at a given state $x_k \in X_k$, this method first fixes the disturbances $w_{k+1}, \ldots, w_{k+\ell-1}$ to some typical values. It then applies the control $\tilde{u}_k$ that minimizes over $u_k \in \tilde{U}_k(x_k)$ the Q-factor

$$\tilde{Q}_k(x_k, u_k) = E\Big\{g(x_k, u_k, w_k) + H_{k+1}\big(f(x_k, u_k, w_k)\big)\Big\}, \tag{2.43}$$

where $H_{k+1}(x_{k+1})$ is the optimal cost of the deterministic transfer from x_{k+1} to 0 in $\ell - 1$ steps with controls $\tilde{u}_m$ from the sets $\tilde{U}_m(x_m)$, $m = k+1, \ldots, k+\ell-1$, and with the disturbances fixed at their typical values. Here we require a constrained controllability condition that guarantees that this transfer is possible.

Note that the minimization over $u_k \in \tilde{U}_k(x_k)$ of the Q-factor (2.43) can be implemented by optimizing over sequences $\{u_k, u_{k+1}, \ldots, u_{k+\ell-1}\}$ an ℓ-stage deterministic optimal control problem. This is the problem that seamlessly concatenates the first stage minimization over u_k [cf. Eq. (2.43)] with the $(\ell - 1)$-stage minimization of the base heuristic. Consistent with the general rollout approach of this section, it may be possible to address this problem with gradient-based optimization methods.

A drawback of the MPC method of the preceding example is that it may not be well suited for on-line implementation because of the substantial amount of computation required at each state x_k. An alternative is to introduce *approximation in policy space on top of approximation in value space*; cf. Section 2.1.5. To this end, we can generate a large number of sample states x_k^s, $s = 1, \ldots, q$, and calculate the corresponding controls u_k^s using the Q-factor minimization

$$u_k^s \in \arg\min_{u_k \in \tilde{U}(x_k^s)} E\Big\{g(x_k^s, u_k, w_k) + H_{k+1}\big(f(x_k^s, u_k, w_k)\big)\Big\},$$

[cf. Eq. (2.43)]. We can then use the pairs (x_k^s, u_k^s), $s = 1, \ldots, q$, and some form of regression to train a Q-factor parametric architecture $\tilde{Q}_k(x_k, u_k, \bar{r}_k)$ such as a neural network [cf. the approximation in policy space approach of Eq. (2.6)]. Once this is done, the MPC controls can be generated on-line using the minimization

$$\tilde{\mu}_k(x_k) \in \arg\min_{u_k \in \tilde{U}_k(x_k)} \tilde{Q}_k(x_k, u_k, \bar{r}_k);$$

cf. Eq. (2.7). This type of approximation in policy space approach may be applied more broadly in MPC methods where the on-line computational requirements are excessive.

2.6 NOTES AND SOURCES

Approximation in value space has been considered in an ad hoc manner since the early days of DP, motivated by the curse of dimensionality. The idea was reframed and coupled with model-free simulation methods that originated in the late 1980s in artificial intelligence. Since that time, approximation in value space has been one of the two pillars of approximate DP/RL. The other pillar is approximation in policy space, which was discussed briefly in Section 2.1.4 and will be revisited in Section 4.11.

The problem approximation approach has a long history in optimal control and operations research. The author's paper [Ber07] describes a few enforced decomposition schemes based on constraint relaxation. The book [Ber17], Section 6.2.1, provides some additional examples from flexible manufacturing and multiarmed bandit problems; see also the thesis by Kimemia [Kim82], and the papers by Kimemia, Gershwin, and Bertsekas [KGB82], and Whittle [Whi88].

The main idea of rollout algorithms, obtaining an improved policy starting from some other suboptimal policy, has appeared in several DP application contexts. The name "rollout" was coined by Tesauro in specific reference to rolling the dice in the game of backgammon [TeG96]. In Tesauro's proposal, a given backgammon position is evaluated by "rolling out" many games starting from that position, using a simulator, and the results are averaged to provide a "score" for the position; see Example 2.4.2. The use of the name "rollout" has gradually expanded beyond its original context; for example the samples collected through simulated trajectories are referred to as "rollouts" by some authors.

The application of rollout algorithms to discrete deterministic optimization problems, and the notions of sequential consistency, sequential improvement, fortified rollout, and the use of multiple heuristics (also called "parallel rollout") were given in the paper by Bertsekas, Tsitsiklis, and Wu [BTW97], and also in the neuro-dynamic book by Bertsekas and Tsitsiklis [BeT96]. Rollout algorithms for stochastic problems were further formalized in the papers by Bertsekas [Ber97b], and Bertsekas and Castanon [BeC99]. A discussion of rollout algorithms as applied to network optimization problems may be found in the author's textbook [Ber98].

For more recent work on rollout algorithms and related methods, see Secomandi [Sec00], [Sec01], [Sec03], Ferris and Voelker [FeV02], [FeV04], McGovern, Moss, and Barto [MMB02], Savagaonkar, Givan, and Chong [SGC02], Bertsimas and Popescu [BeP03], Guerriero and Mancini [GuM03], Tu and Pattipati [TuP03], Wu, Chong, and Givan [WCG03], Chang, Givan, and Chong [CGC04], Meloni, Pacciarelli, and Pranzo [MPP04], Yan, Diaconis, Rusmevichientong, and Van Roy [YDR04], Besse and Chaib-draa [BeC08], Sun et al. [SZL08], Bertazzi et al. [BBG13], Sun et al. [SLJ13], Tesauro et al. [TGL13], Beyme and Leung [BeL14], Goodson, Thomas, and Ohlmann [GTO15], Li and Womer [LiW15], Mastin and Jaillet [MaJ15],

Huang, Jia, and Guan [HJG16], Simroth, Holfeld, and Brunsch [SHB15], and Lan, Guan, and Wu [LGW16]. For a recent survey by the author, see [Ber13b]. These works discuss a broad variety of applications and case studies, and generally report positive computational experience.

The idea of rollout that uses limited lookahead, adaptive pruning of the lookahead tree, and cost function approximation at the end of the lookahead horizon was suggested by Tesauro and Galperin [TeG96] in the context of backgammon. Related ideas appeared earlier in the paper by Abramson [Abr90], in a game playing context. The paper and monograph by Chang, Hu, Fu, and Marcus [CFH05], [CFH13] proposed and analyzed adaptive sampling in connection with DP, including statistical tests to control the sampling process. The name "Monte Carlo tree search" (Section 2.4.2) has become popular, and in its current use, it encompasses a broad range of methods that involve adaptive sampling, rollout, extensions to sequential games, and the use and analysis of various statistical tests. We refer to the papers by Coulom [Cou06], the survey by Browne et al. [BPW12], and the discussion by Fu [Fu17]. The development of statistical tests for adaptive sampling has been influenced by works on multiarmed bandit problems; see the papers by Lai and Robbins [LaR85], Agrawal [Agr95], Burnetas and Katehakis [BuK97], Meuleau and Bourgine [MeB99], Auer, Cesa-Bianchi, and Fischer [ACF02], Peret and Garcia [PeG04], Kocsis and Szepesvari [KoS06], and the monograph by Munos [Mun14]. The technique for variance reduction in the calculation of Q-factor differences (Section 2.4.2) was given in the author's paper [Ber97].

The MPC approach is popular in a variety of control system design contexts, and particularly in chemical process control and robotics, where meeting explicit control and state constraints is an important practical issue. The connection of MPC with rollout algorithms was made in the author's review paper [Ber05a]. The stability analysis given here is based on the work of Keerthi and Gilbert [KeG88]. For an early survey of the field, which gives many of the early references, see Morari and Lee [MoL99], and for a more recent survey see Mayne [May14]. For related textbooks, see Maciejowski [Mac02], Camacho and Bordons [CaB04], Kouvaritakis and Cannon [KoC15], and Borelli, Bemporad, and Morari [BBM17].

In our account of MPC, we have restricted ourselves to deterministic problems possibly involving tight state constraints as well as control constraints. Problems with stochastic uncertainty and state constraints are more challenging because of the difficulty of guaranteeing that the constraints are satisfied; see the survey by Mayne [May14] for a review of various approaches that have been used in this context. The textbook [Ber17], Section 6.4, describes MPC for problems with set membership uncertainty and state constraints, using target tube/reachability concepts, which originated in the author's PhD thesis and subsequent papers [Ber71], [Ber72a], [BeR71a], [BeR71b], [BeR73]. Target tubes were used subsequently in MPC and other contexts by several authors; see the surveys by Blanchini [Bla99]

and Mayne [May14].

Reinforcement Learning and Optimal Control

by

Dimitri P. Bertsekas

Massachusetts Institute of Technology

Chapter 3
Parametric Approximation

DRAFT

This is Chapter 3 of the draft textbook "Reinforcement Learning and Optimal Control." The chapter represents "work in progress," and it will be periodically updated. It more than likely contains errors (hopefully not serious ones). Furthermore, its references to the literature are incomplete. Your comments and suggestions to the author at dimitrib@mit.edu are welcome. The date of last revision is given below.

The date of last revision is given below. (A "revision" is any version of the chapter that involves the addition or the deletion of at least one paragraph or mathematically significant equation.)

January 25, 2019

3

Parametric Approximation

Contents

3.1. Approximation Architectures . . . . p. 2
 3.1.1. Linear and Nonlinear Feature-Based Architectures . p. 2
 3.1.2. Training of Linear and Nonlinear Architectures . . p. 9
 3.1.3. Incremental Gradient and Newton Methods . . . . p. 10
3.2. Neural Networks . . . . p. 23
 3.2.1. Training of Neural Networks . . . . p. 27
 3.2.2. Multilayer and Deep Neural Networks . . . . p. 30
3.3. Sequential Dynamic Programming Approximation . . . . p. 34
3.4. Q-factor Parametric Approximation . . . . p. 36
3.5. Notes and Sources . . . . p. 39

Clearly, for the success of approximation in value space, it is important to select a class of lookahead functions $\tilde{J}_k$ that is suitable for the problem at hand. In the preceding chapter we discussed several methods for choosing $\tilde{J}_k$ based mostly on problem approximation, including rollout. In this chapter we discuss an alternative approach, whereby $\tilde{J}_k$ is chosen to be a member of a parametric class of functions, including neural networks, with the parameters "optimized" or "trained" by using some algorithm.

3.1 APPROXIMATION ARCHITECTURES

The starting point for the schemes of this chapter is a class of functions $\tilde{J}_k(x_k, r_k)$ that for each k, depend on the current state x_k and a vector $r_k = (r_{1,k}, \ldots, r_{m,k})$ of m "tunable" scalar parameters, also called *weights*. By adjusting the weights, one can change the "shape" of $\tilde{J}_k$ so that it is a reasonably good approximation to the true optimal cost-to-go function J_k^*. The class of functions $\tilde{J}_k(x_k, r_k)$ is called an *approximation architecture*, and the process of choosing the parameter vectors r_k is commonly called *training* or *tuning* the architecture.

The simplest training approach is to do some form of semi-exhaustive or semi-random search in the space of parameter vectors and adopt the parameters that result in best performance of the associated one-step lookahead controller (according to some criterion). More systematic approaches are based on numerical optimization, such as for example a least squares fit that aims to match the cost approximation produced by the architecture to a "training set," i.e., a large number of pairs of state and cost values that are obtained through some form of sampling process. Throughout this chapter we will focus on this latter approach.

3.1.1 Linear and Nonlinear Feature-Based Architectures

There is a large variety of approximation architectures, based for example on polynomials, wavelets, discretization/interpolation schemes, neural networks, and others. A particularly interesting type of cost approximation involves *feature extraction*, a process that maps the state x_k into some vector $\phi_k(x_k)$, called the *feature vector* associated with x_k at time k. The vector $\phi_k(x_k)$ consists of scalar components

$$\phi_{1,k}(x_k), \ldots, \phi_{m,k}(x_k),$$

called *features*. A feature-based cost approximation has the form

$$\tilde{J}_k(x_k, r_k) = \hat{J}_k\big(\phi_k(x_k), r_k\big),$$

where r_k is a parameter vector and $\hat{J}_k$ is some function. Thus, the cost approximation depends on the state x_k through its feature vector $\phi_k(x_k)$.

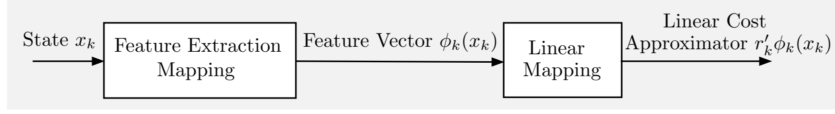

Figure 3.1.1 The structure of a linear feature-based architecture. We use a feature extraction mapping to generate an input $\phi_k(x_k)$ to a linear mapping defined by a weight vector r_k.

Note that we are allowing for different features $\phi_k(x_k)$ and different parameter vectors r_k for each stage k. This is necessary for nonstationary problems (e.g., if the state space changes over time), and also to capture the effect of proximity to the end of the horizon. On the other hand, for stationary problems with a long or infinite horizon, where the state space does not change with k, it is common to use the same features and parameters for all stages. The subsequent discussion can easily be adapted to infinite horizon methods, as we will discuss in Chapter 4.

Features are often handcrafted, based on whatever human intelligence, insight, or experience is available, and are meant to capture the most important characteristics of the current state. There are also systematic ways to construct features, including the use of neural networks, which we will discuss shortly. In this section, we provide a brief and selective discussion of architectures, and we refer to the specialized literature (e.g., Bertsekas and Tsitsiklis [BeT96], Bishop [Bis95], Haykin [Hay09], Sutton and Barto [SuB18]), and the author's [Ber12], Section 6.1.1, for more detailed presentations.

One idea behind using features is that the optimal cost-to-go functions J_k^* may be complicated nonlinear mappings, so it is sensible to try to break their complexity into smaller, less complex pieces. In particular, if the features encode much of the nonlinearity of J_k^*, we may be able to use a relatively simple architecture $\hat{J}_k$ to approximate J_k^*. For example, with a well-chosen feature vector $\phi_k(x_k)$, a good approximation to the cost-to-go is often provided by *linearly* weighting the features, i.e.,

$$\tilde{J}_k(x_k, r_k) = \hat{J}_k\big(\phi_k(x_k), r_k\big) = \sum_{\ell=1}^{m} r_{\ell,k}\phi_{\ell,k}(x_k) = r_k'\phi_k(x_k), \tag{3.1}$$

where $r_{\ell,k}$ and $\phi_{\ell,k}(x_k)$ are the ℓth components of r_k and $\phi_k(x_k)$, respectively, and $r_k'\phi_k(x_k)$ denotes the inner product of r_k and $\phi_k(x_k)$, viewed as column vectors of $\Re^m$ (a prime denoting transposition, so r_k' is a row vector; see Fig. 3.1.1).

This is called a *linear feature-based architecture*, and the scalar parameters $r_{\ell,k}$ are also called *weights*. Among other advantages, these architectures admit simpler training algorithms that their nonlinear counterparts. Mathematically, the approximating function $\tilde{J}_k(x_k, r_k)$ can be viewed as a

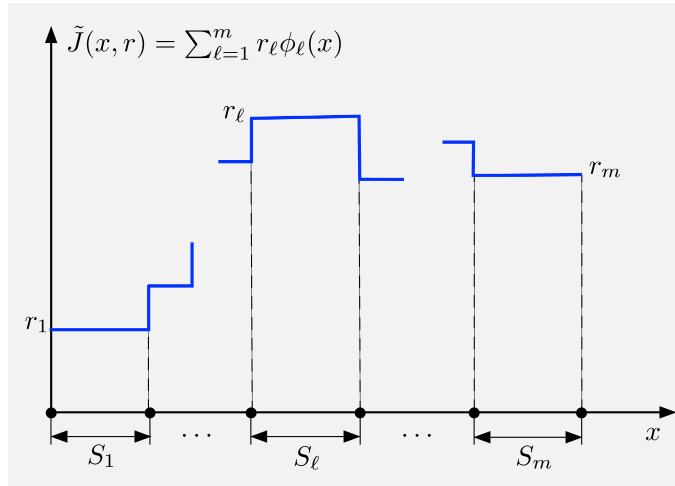

Figure 3.1.2 Illustration of a piecewise constant architecture. The state space is partitioned into subsets $S_1, \ldots, S_m$, with each subset S_ℓ defining the feature

$$\phi_\ell(x) = \begin{cases} 1 & \text{if } x \in S_\ell, \\ 0 & \text{if } x \notin S_\ell, \end{cases}$$

with its own weight r_ℓ.

member of the subspace spanned by the features $\phi_{\ell,k}(x_k)$, $\ell = 1, \ldots, m$, which for this reason are also referred to *basis functions*. We provide a few examples, where for simplicity we drop the index k.

Example 3.1.1 (Piecewise Constant Approximation)

Suppose that the state space is partitioned into subsets $S_1, \ldots, S_m$, so that every state belongs to one and only one subset. Let the ℓth feature be defined by membership to the set S_ℓ, i.e.,

$$\phi_\ell(x) = \begin{cases} 1 & \text{if } x \in S_\ell, \\ 0 & \text{if } x \notin S_\ell. \end{cases}$$

Consider the architecture

$$\tilde J(x,r) = \sum_{\ell=1}^m r_\ell \phi_\ell(x),$$

where r is the vector consists of the m scalar parameters $r_1, \ldots, r_m$. It can be seen that $\tilde J(x,r)$ is the piecewise constant function that has value r_ℓ for all states within the set S_ℓ; see Fig. 3.1.2.

The piecewise constant approximation is an example of a linear feature-based architecture that involves exclusively *local features*. These are features that take a nonzero value only for a relatively small subset of states. Thus a change of a single weight causes a change of the value of $\tilde{J}(x,r)$ for relatively few states x. At the opposite end we have linear feature-based architectures that involve *global features*. These are features that take nonzero values for a large number of states. The following is an example.

Example 3.1.2 (Polynomial Approximation)

An important case of linear architecture is one that uses polynomial basis functions. Suppose that the state consists of n components $x^1,\ldots,x^n$, each taking values within some range of integers. For example, in a queueing system, x^i may represent the number of customers in the ith queue, where $i=1,\ldots,n$. Suppose that we want to use an approximating function that is quadratic in the components x^i. Then we can define a total of $1+n+n^2$ basis functions that depend on the state $x=(x^1,\ldots,x^n)$ via

$$\phi_0(x)=1, \qquad \phi_i(x)=x^i, \qquad \phi_{ij}(x)=x^ix^j, \quad i,j=1,\ldots,n.$$

A linear approximation architecture that uses these functions is given by

$$\tilde{J}(x,r)=r_0+\sum_{i=1}^n r_ix^i+\sum_{i=1}^n\sum_{j=i}^n r_{ij}x^ix^j,$$

where the parameter vector r has components r_0, r_i, and r_{ij}, with $i=1,\ldots,n$, $j=k,\ldots,n$. Indeed, any kind of approximating function that is polynomial in the components $x^1,\ldots,x^n$ can be constructed similarly.

A more general polynomial approximation may be based on some other known features of the state. For example, we may start with a feature vector

$$\phi(x)=\big(\phi_1(x),\ldots,\phi_m(x)\big)',$$

and transform it with a quadratic polynomial mapping. In this way we obtain approximating functions of the form

$$\tilde{J}(x,r)=r_0+\sum_{i=1}^m r_i\phi_i(x)+\sum_{i=1}^m\sum_{j=k}^m r_{ij}\phi_i(x)\phi_j(x),$$

where the parameter r has components r_0, r_i, and r_{ij}, with $i,j=1,\ldots,m$. This can also be viewed as a linear architecture that uses the basis functions

$$w_0(x)=1, \qquad w_i(x)=\phi_i(x), \qquad w_{ij}(x)=\phi_i(x)\phi_j(x), \quad i,j=1,\ldots,m.$$

The preceding example architectures are generic in the sense that they can be applied to many different types of problems. Other architectures rely on problem-specific insight to construct features, which are then combibed into a relatively simple architecture. The following are two examples involving games.

Example 3.1.3 (Tetris)

Let us revisit the game of tetris, which we discussed in Example 1.3.4. We can model the problem of finding an optimal playing strategy as a finite horizon problem with a very large horizon.

In Example 1.3.4 we viewed as state the pair of the board position x and the shape of the current falling block y. We viewed as control, the horizontal positioning and rotation applied to the falling block. However, the DP algorithm can be executed over the space of x only, since y is an uncontrollable state component. The optimal cost-to-go function is a vector of huge dimension (there are 2^{200} board positions in a "standard" tetris board of width 10 and height 20). However, it has been successfully approximated in practice by low-dimensional linear architectures.

In particular, the following features have been proposed in [BeI96]: the heights of the columns, the height differentials of adjacent columns, the wall height (the maximum column height), the number of holes of the board, and the constant 1 (the unit is often included as a feature in cost approximation architectures, as it allows for a constant shift in the approximating function). These features are readily recognized by tetris players as capturing important aspects of the board position.† There are a total of 22 features for a "standard" board with 10 columns. Of course the $2^{200} \times 22$ matrix of feature values cannot be stored in a computer, but for any board position, the corresponding row of features can be easily generated, and this is sufficient for implementation of the associated approximate DP algorithms. For recent works involving approximate DP methods and the preceding 22 features, see [Sch13], [GGS13], and [SGG15], which reference several other related papers.

Example 3.1.4 (Computer Chess)

Computer chess programs that involve feature-based architectures have been available for many years, and are still used widely (they have been upstaged in the mid-2010s by alternative types of chess programs that are based on neural network-based techniques that will be discussed later). These programs are based on approximate DP for minimax problems,‡ a feature-based parametric architecture, and multistep lookahead. For the most part, however, the computer chess training methodology has been qualitatively different from the parametric approximation methods that we consider in this book.

In particular, with few exceptions, the training of chess architectures has been done with ad hoc hand-tuning techniques (as opposed to some form of optimization). Moreover, the features have traditionally been hand-crafted

† The use of feature-based approximate DP methods for the game of tetris was first suggested in the paper [TsV96], which introduced just two features (in addition to the constant 1): the wall height and the number of holes of the board. Most studies have used the set of features described here, but other sets of features have also been used; see [ThS09] and the discussion in [GGS13].

‡ We have not discussed DP for minimax problems and two-player games, but the ideas of approximation in value space apply to these contexts as well; see [Ber17] for a discussion that is focused on computer chess.

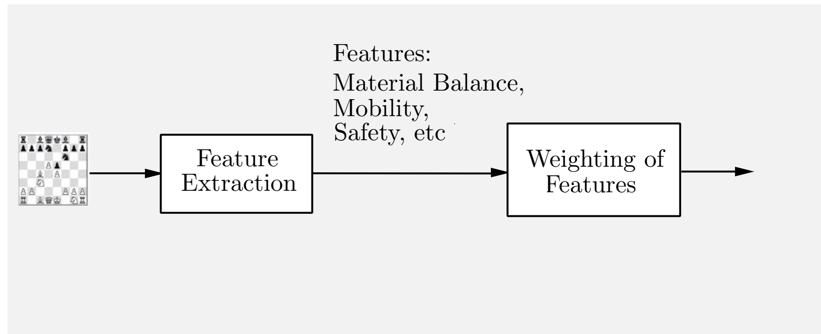

Figure 3.1.3 A feature-based architecture for computer chess.

based on chess-specific knowledge (as opposed to automatically generated through a neural network or some other method). Indeed, it has been argued that the success of chess programs in outperforming the best humans, can be more properly attributed to the brute-force calculating power of modern computers, which provides long and accurate lookahead, than to the ability of simulation-based algorithmic approaches, which can learn powerful playing strategies that humans have difficulty conceiving or executing. This assessment is changing rapidly following the impressive success of the AlphaZero chess program (Silver et al. [SHS17]). Still, however, computer chess with its use of long lookahead, provides an approximate DP paradigm that may be better suited for some practical contexts where a lot of computational power is available, but innovative and sophisticated algorithms are hard to construct.

The fundamental paper on which all computer chess programs are based was written by one of the most illustrious modern-day applied mathematicians, C. Shannon [Sha50]. Shannon proposed limited lookahead and evaluation of the end positions by means of a "scoring function" (in our terminology this plays the role of a cost function approximation). This function may involve, for example, the calculation of a numerical value for each of a set of major features of a position that chess players easily recognize (such as material balance, mobility, pawn structure, and other positional factors), together with a method to combine these numerical values into a single score. Shannon then went on to describe various strategies of exhaustive and selective search over a multistep lookahead tree of moves.

We may view the scoring function as a hand-crafted feature-based architecture for evaluating a chess position/state (cf. Fig. 3.1.3). In our earlier notation, it is a function of the form

$$\tilde{J}(x,r) = \hat{J}\big(\phi(x),r\big),$$

which maps a position x into a cost-to-go approximation $\hat{J}\big(\phi(x),r\big)$, a score whose value depends on the feature vector $\phi(x)$ of the position and a parameter vector r.†

† Because of the nature of the chess game, $\tilde{J}(x,r)$ does not depend critically

In most computer chess programs, the features are weighted linearly, i.e., the architecture $\tilde{J}(x,r)$ that is used for limited lookahead is linear [cf. Eq. (3.1)]. In many cases, the weights are determined manually, by trial and error based on experience. However, in some programs, the weights are determined with supervised learning techniques that use examples of grandmaster play, i.e., by adjustment to bring the play of the program as close as possible to the play of chess grandmasters. This is a technique that applies more broadly in artificial intelligence; see Tesauro [Tes89], [Tes01]. An example is the Deepchess program by David, Netanyahu, and Wolf [DNW16], estimated to perform at a strong grandmaster level, which uses a deep neural network to extract features for use in another approximation architecture.

In a recent computer chess breakthrough, the entire idea of extracting features of a position through human expertise was abandoned in favor of feature discovery through self-play and the use of neural networks. The first program of this type to attain supremacy not only over humans, but also over the best computer programs that use human expertise-based features, was AlphaZero (Silver et al. [SHS17]). This program is based on DP principles of policy iteration, and Monte Carlo tree search. It will be discussed further in Chapter 4.

The next example considers a feature extraction strategy that is particularly relevant to problems of partial state information.

Example 3.1.5 (Feature Extraction from Sufficient Statistics)

The concept of a sufficient statistic, which originated in inference methodologies, plays an important role in DP. As discussed in Section 1.3, it refers to quantities that summarize all the essential content of the state x_k for optimal control selection at time k.

In particular, consider a partial information context where at time k we have accumulated an *information* record

$$I_k = (z_0, \ldots, z_k, u_0, \ldots, u_{k-1}),$$

which consists of the past controls $u_0, \ldots, u_{k-1}$ and state-related measurements $z_0, \ldots, z_k$ obtained at the times $0, \ldots, k$, respectively. The control u_k is allowed to depend only on I_k, and the optimal policy is a sequence of the form $\big\{\mu_0^*(I_0), \ldots, \mu_{N-1}^*(I_{N-1})\big\}$. We say that a function $S_k(I_k)$ is a *sufficient statistic at time* k if the control function μ_k^* depends on I_k only through

---

on the time (or move) index. The duration of the game is unknown and so is the horizon of the problem. We are dealing essentially with an infinite horizon minimax problem, whose termination time is finite but unknown, similar to the stochastic shortest path problems to be discussed in Chapter 4. Still, however, chess programs often use features and weights that depend on the phase of the game (opening, middlegame, or endgame). Moreover the programs include specialized knowledge, such as opening and endgame databases. In our discussion we will ignore such possibilities.

$S_k(I_k)$, i.e., for some function $\hat{\mu}_k$, we have

$$\mu_k^*(I_k) = \hat{\mu}_k\big(S_k(I_k)\big).$$

where μ_k^* is optimal.

There are several examples of sufficient statistics, and they are typically problem-dependent. A trivial possibility is to view I_k itself as a sufficient statistic, and a more sophisticated possibility is to view the belief state $p_k(I_k)$ as a sufficient statistic (this is the conditional probability distribution of x_k given I_k; a more detailed discussion of these possibilities can be found in [Ber17], Chapter 4).

Since the sufficient statistic contains all the relevant information for control purposes, an idea that suggests itself is to introduce features of the sufficient statistic and to train a corresponding approximation architecture accordingly. As examples of potentially good features, one may consider a partial history (such as the last m measurements and controls in I_k), or some special characteristic of I_k (such as whether some alarm-like "special" event has been observed). In the case where the belief state $p_k(I_k)$ is used as a sufficient statistic, examples of good features may be a point estimate based on $p_k(I_k)$, the variance of this estimate, and other quantities that can be simply extracted from $p_k(I_k)$ [e.g., simplified or certainty equivalent versions of $p_k(I_k)$].

Unfortunately, in many situations an adequate set of features is not known, so it is important to have methods that construct features automatically, to supplement whatever features may already be available. Indeed, there are architectures that do not rely on the knowledge of good features. One of the most popular is *neural networks*, which we will describe in Section 3.2. Some of these architectures involve training that constructs simultaneously both the feature vectors $\phi_k(x_k)$ and the parameter vectors r_k that weigh them.

3.1.2 Training of Linear and Nonlinear Architectures

The process of choosing the parameter vector r of a parametric architecture $\tilde{J}(x,r)$ is generally referred to as training. The most common type of training is based on a least squares optimization, also known as *least squares regression*. Here a set of state-cost training pairs (x^s,β^s), $s=1,\ldots,q$, called the *training set*, is collected and r is determined by solving the problem

$$\min_r \sum_{s=1}^{q} \big(\tilde{J}(x^s,r)-\beta^s\big)^2. \tag{3.2}$$

Thus r is chosen to minimize the error between sample costs β^s and the architecture-predicted costs $\tilde{J}(x^s,r)$ in a least squares sense. Here typically, there is some "target" cost function J that we aim to approximate with

$\tilde{J}(\cdot, r)$, and the sample cost β^s is the value $J(x^s)$ plus perhaps some error or "noise."

The cost function of the training problem (3.2) is generally nonconvex, which may pose challenges, since there may exist multiple local minima. However, for a linear architecture the cost function is convex quadratic, and the training problem admits a closed-form solution. In particular, for the linear architecture

$$\tilde{J}(x, r) = r'\phi(x),$$

the problem becomes

$$\min_r \sum_{s=1}^{q} \big(r'\phi(x^s) - \beta^s\big)^2.$$

By setting the gradient of the quadratic objective to 0, we have the linear equation

$$\sum_{s=1}^{q} \phi(x^s)\big(r'\phi(x^s) - \beta^s\big) = 0,$$

or

$$\sum_{s=1}^{q} \phi(x^s)\phi(x^s)'r = \sum_{s=1}^{q} \phi(x^s)\beta^s.$$

Thus by matrix inversion we obtain the minimizing parameter vector

$$\hat{r} = \left(\sum_{s=1}^{q} \phi(x^s)\phi(x^s)'\right)^{-1} \sum_{s=1}^{q} \phi(x^s)\beta^s. \tag{3.3}$$

(Since sometimes the inverse above may not exist, an additional quadratic in r, called a *regularization* function, is added to the least squares objective to deal with this, and also to help with other issues to be discussed later.)

Thus a linear architecture has the important advantage that the training problem can be solved exactly and conveniently with the formula (3.3) (of course it may be solved by any other algorithm that is suitable for linear least squares problems). By contrast, if we use a nonlinear architecture, such as a neural network, the associated least squares problem is nonquadratic and also nonconvex. Despite this fact, through a combination of sophisticated implementation of special gradient algorithms, called *incremental*, and powerful computational resources, neural network methods have been successful in practice as we will discuss in Section 3.2.

3.1.3 Incremental Gradient and Newton Methods

We will now digress to discuss special methods for solution of the least squares training problem (3.2), assuming a parametric architecture that

is differentiable in the parameter vector. This methodology is properly viewed as a subject in nonlinear programming and iterative algorithms, and as such it can be studied independently of the approximate DP methods of this book. Thus the reader who has already some exposure to the subject may skip to the next section, and return later as needed.

Incremental methods have a rich theory, and our presentation in this section is brief, focusing primarily on implementation and intuition. Some surveys and book references are provided at the end of the chapter, which include a more detailed treatment. In particular, the neuro-dynamic programming book [BeT96] contains convergence analysis for both deterministic and stochastic training problems. Since, we want to cover problems that are more general than the specific least squares training problem (3.2), we will adopt a broader formulation and notation that are standard in nonlinear programming.

Incremental Gradient Methods

We view the training problem (3.2) as a special case of the minimization of a sum of component functions

$$f(y) = \sum_{i=1}^{m} f_i(y), \tag{3.4}$$

where each f_i is a differentiable scalar function of the n-dimensional vector y (this is the parameter vector). Thus we use the more common symbols y and m in place of r and q, respectively, and we replace the least squares terms $\tilde{J}(x^s, r)$ in the training problem (3.2) with the generic terms $f_i(y)$.

The (ordinary) gradient method for problem (3.4) has the form†

$$y^{k+1} = y^k - \gamma^k \nabla f(y^k) = y^k - \gamma^k \sum_{i=1}^{m} \nabla f_i(y^k), \tag{3.5}$$

where γ^k is a positive stepsize parameter. Incremental gradient methods have the general form

$$y^{k+1} = y^k - \gamma^k \nabla f_{i_k}(y^k), \tag{3.6}$$

where i_k is some index from the set $\{1, \ldots, m\}$, chosen by some deterministic or randomized rule. Thus a single component function f_{i_k} is used at

---

† We use standard calculus notation for gradients; see, e.g., [Ber16], Appendix A. In particular, $\nabla f(y)$ denotes the n dimensional vector whose components are the first partial derivatives $\partial f(y)/\partial y_i$ of f with respect to the components $y_1, \ldots, y_n$ of the vector y.

each iteration of the incremental method (3.6), with great economies in gradient calculation cost over the ordinary gradient method (3.5), particularly when m is large.

The method for selecting the index i_k of the component to be iterated on at iteration k is important for the performance of the method. Three common rules are:

(1) A *cyclic order*, the simplest rule, whereby the indexes are taken up in the fixed deterministic order $1, \ldots, m$, so that i_k is equal to (k modulo m) plus 1. A contiguous block of iterations involving the components

$$f_1, \ldots, f_m$$

in this order and exactly once is called a *cycle*.

(2) A *uniform random order*, whereby the index i_k chosen randomly by sampling over all indexes with a uniform distribution, independently of the past history of the algorithm. This rule may perform better that the cyclic rule in some circumstances.

(3) A *cyclic order with random reshuffling*, whereby the indexes are taken up one by one within each cycle, but their order after each cycle is reshuffled randomly (and independently of the past). This rule is used widely in practice, particularly when the number of components m is modest, for reasons to be discussed later.

Note that in the cyclic case, it is essential to include all components in a cycle, for otherwise some components will be sampled more often than others, leading to a bias in the convergence process. Simlarly, it is necessary to sample according to the uniform distribution in the random order case.

Focusing for the moment on the cyclic rule, we note that the motivation for the incremental gradient method is faster convergence: we hope that far from the solution, a single cycle of the method will be as effective as several (as many as m) iterations of the ordinary gradient method (think of the case where the components f_i are similar in structure). Near a solution, however, the incremental method may not be as effective.

To be more specific, we note that there are two complementary performance issues to consider in comparing incremental and nonincremental methods:

(a) *Progress when far from convergence.* Here the incremental method can be much faster. For an extreme case take m large and all components f_i identical to each other. Then an incremental iteration requires m times less computation than a classical gradient iteration, but gives exactly the same result, when the stepsize is scaled to be m times larger. While this is an extreme example, it reflects the essential mechanism by which incremental methods can be much superior: far from the minimum a single component gradient will point to "more

or less" the right direction, at least most of the time; see the following example.

(b) *Progress when close to convergence.* Here the incremental method can be inferior. In particular, the ordinary gradient method (3.5) can be shown to converge with a constant stepsize under reasonable assumptions, see e.g., [Ber16], Chapter 1. However, the incremental method requires a diminishing stepsize, and its ultimate rate of convergence can be much slower.

This type of behavior is illustrated in the following example.

Example 3.1.6

Assume that y is a scalar, and that the problem is

$$\text{minimize} \quad f(y) = \tfrac{1}{2}\sum_{i=1}^{m}(c_i y - b_i)^2$$
$$\text{subject to} \quad y \in \Re,$$

where c_i and b_i are given scalars with $c_i \neq 0$ for all i. The minimum of each of the components $f_i(y) = \frac{1}{2}(c_i y - b_i)^2$ is

$$y_i^* = \frac{b_i}{c_i},$$

while the minimum of the least squares cost function f is

$$y^* = \frac{\sum_{i=1}^m c_i b_i}{\sum_{i=1}^m c_i^2}.$$

It can be seen that y^* lies within the range of the component minima

$$R = \left[\min_i y_i^*, \max_i y_i^*\right],$$

and that for all y *outside* the range R, the gradient

$$\nabla f_i(y) = c_i(c_i y - b_i)$$

has the same sign as $\nabla f(y)$ (see Fig. 3.1.4). As a result, when outside the region R, the incremental gradient method

$$y^{k+1} = y^k - \gamma^k c_{i_k}(c_{i_k} y^k - b_{i_k})$$

approaches y^* at each step, provided the stepsize γ^k is small enough. In fact it can be verified that it is sufficient that

$$\gamma^k \leq \min_i \frac{1}{c_i^2}.$$

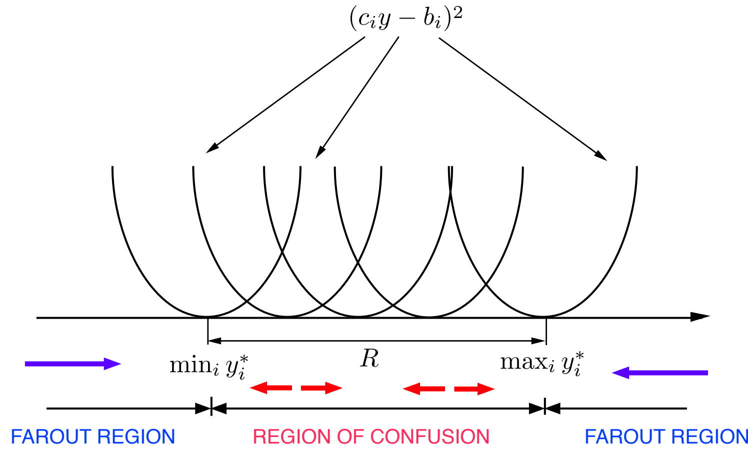

Figure 3.1.4. Illustrating the advantage of incrementalism when far from the optimal solution. The region of component minima

$$R = \left[\min_i y_i^*, \max_i y_i^*\right],$$

is labeled as the "region of confusion." It is the region where the method does not have a clear direction towards the optimum. The ith step in an incremental gradient cycle is a gradient step for minimizing $(c_i y - b_i)^2$, so if y lies outside the region of component minima $R = \left[\min_i y_i^*, \max_i y_i^*\right]$, (labeled as the "farout region") and the stepsize is small enough, progress towards the solution y^* is made.

However, for y *inside* the region R, the ith step of a cycle of the incremental gradient method need not make progress. It will approach y^* (for small enough stepsize γ^k) only if the current point y^k does not lie in the interval connecting y_i^* and y^*. This induces an oscillatory behavior within the region R, and as a result, the incremental gradient method will typically not converge to y^* unless $\gamma^k \to 0$. By contrast, the ordinary gradient method, which takes the form

$$y^{k+1} = y^k - \gamma \sum_{i=1}^{m} c_i (c_i y^k - b_i),$$

can be verified to converge to y^* for any constant stepsize γ with

$$0 < \gamma \leq \frac{1}{\sum_{i=1}^{m} c_i^2}.$$

However, for y outside the region R, a full iteration of the ordinary gradient method need not make more progress towards the solution than a single step of the incremental gradient method. In other words, with comparably intelligent

stepsize choices, *far from the solution (outside R), a single pass through the entire set of cost components by incremental gradient is roughly as effective as m passes by ordinary gradient.*

The discussion of the preceding example relies on y being one-dimensional, but in many multidimensional problems the same qualitative behavior can be observed. In particular, a pass through the ith component f_i by the incremental gradient method can make progress towards the solution in the region where the component gradient $\nabla f_{i_k}(y^k)$ makes an angle less than 90 degrees with the cost function gradient $\nabla f(y^k)$. If the components f_i are not "too dissimilar", this is likely to happen in a region of points that are not too close to the optimal solution set. This behavior has been verified in many practical contexts, including the training of neural networks (cf. the next section), where incremental gradient methods have been used extensively, frequently under the name *backpropagation methods*.

Stepsize Choice and Diagonal Scaling

The choice of the stepsize γ^k plays an important role in the performance of incremental gradient methods. On close examination, it turns out that the direction used by the method differs from the gradient direction by an error that is proportional to the stepsize, and for this reason a diminishing stepsize is essential for convergence to a local minimum of f (convergence to a local minimum is the best we hope for since the cost function may not be convex).

However, it turns out that a peculiar form of convergence also typically occurs for a constant but sufficiently small stepsize. In this case, the iterates converge to a "limit cycle", whereby the ith iterates ψ_i within the cycles converge to a different limit than the jth iterates ψ_j for $i \neq j$. The sequence $\{y^k\}$ of the iterates obtained at the end of cycles converges, except that the limit obtained *need not* be a minimum of f, even when f is convex. The limit tends to be close to the minimum point when the constant stepsize is small (see Section 2.4 of [Ber16] for analysis and examples). In practice, it is common to use a constant stepsize for a (possibly prespecified) number of iterations, then decrease the stepsize by a certain factor, and repeat, up to the point where the stepsize reaches a prespecified floor value.

An alternative possibility is to use a diminishing stepsize rule of the form

$$\gamma^k = \min\left\{\gamma,\, \frac{\beta_1}{k+\beta_2}\right\},$$

where γ, β_1, and β_2 are some positive scalars. There are also variants of the incremental gradient method that use a constant stepsize throughout, and generically converge to a stationary point of f at a linear rate. In one type of such method the degree of incrementalism gradually diminishes as

the method progresses (see [Ber97a]). Another incremental approach with similar aims, is the aggregated incremental gradient method, which will be discussed later in this section.

Regardless of whether a constant or a diminishing stepsize is ultimately used, to maintain the advantage of faster convergence when far from the solution, the incremental method must use a much larger stepsize than the corresponding nonincremental gradient method (as much as m times larger so that the size of the incremental gradient step is comparable to the size of the nonincremental gradient step).

One possibility is to use an adaptive stepsize rule, whereby the stepsize is reduced (or increased) when the progress of the method indicates that the algorithm is oscillating because it operates within (or outside, respectively) the region of confusion. There are formal ways to implement such stepsize rules with sound convergence properties (see [Tse98], [MYF03], [GOP15a]).

The difficulty with stepsize selection may also be addressed with *diagonal scaling*, i.e., using a stepsize γ_j^k that is different for each of the components y_j of y. Second derivatives can be very effective for this purpose. In generic nonlinear programming problems of unconstrained minimization of a function f, it is common to use diagonal scaling with stepsizes

$$\gamma_j^k = \gamma \left(\frac{\partial^2 f(y^k)}{\partial^2 y_j} \right)^{-1}, \qquad j = 1, \ldots, n,$$

where γ is a constant that is nearly equal 1 (the second derivatives may also be approximated by gradient difference approximations). However, in least squares training problems, this type of scaling is inconvenient because of the additive form of f as a sum of a large number of component functions:

$$f(y) = \sum_{i=1}^{m} f_i(y),$$

cf. Eq. (3.4). Later in this section, when we discuss incremental Newton methods, we will provide a type of diagonal scaling that uses second derivatives and is well suited to the additive character of f.

Stochastic Gradient Descent

Incremental gradient methods are related to methods that aim to minimize an expected value

$$f(y) = E\big\{F(y, w)\big\},$$

where w is a random variable, and $F(\cdot, w) : \Re^n \mapsto \Re$ is a differentiable function for each possible value of w. The *stochastic gradient method* for minimizing f is given by

$$y^{k+1} = y^k - \gamma^k \nabla_y F(y^k, w^k), \tag{3.7}$$

where w^k is a sample of w and $\nabla_y F$ denotes gradient of F with respect to y. This method has a rich theory and a long history, and it is strongly related to the classical algorithmic field of *stochastic approximation*; see the books [BeT96], [KuY03], [Spa03], [Mey07], [Bor08], [BPP13]. The method is also often referred to as *stochastic gradient descent*, particularly in the context of machine learning applications.

If we view the expected value cost $E\{F(y,w)\}$ as a weighted sum of cost function components, we see that the stochastic gradient method (3.7) is related to the incremental gradient method

$$y^{k+1} = y^k - \gamma^k \nabla f_{i_k}(y^k) \tag{3.8}$$

for minimizing a finite sum $\sum_{i=1}^m f_i$, when randomization is used for component selection. An important difference is that the former method involves stochastic sampling of cost components $F(y,w)$ from a possibly infinite population, under some probabilistic assumptions, while in the latter the set of cost components f_i is predetermined and finite. However, it is possible to view the incremental gradient method (3.8), with uniform randomized selection of the component function f_i (i.e., with i_k chosen to be any one of the indexes $1,\ldots,m$, with equal probability $1/m$, and independently of preceding choices), as a stochastic gradient method.

Despite the apparent similarity of the incremental and the stochastic gradient methods, the view that the problem

$$\begin{aligned} &\text{minimize} \quad f(y) = \sum_{i=1}^m f_i(y) \\ &\text{subject to} \quad y \in \Re^n, \end{aligned} \tag{3.9}$$

can simply be treated as a special case of the problem

$$\begin{aligned} &\text{minimize} \quad f(y) = E\{F(y,w)\} \\ &\text{subject to} \quad y \in \Re^n, \end{aligned}$$

is questionable.

One reason is that once we convert the finite sum problem to a stochastic problem, we preclude the use of methods that exploit the finite sum structure, such as the incremental aggregated gradient methods to be discussed in the next subsection. Another reason is that the finite-component problem (3.9) is often genuinely deterministic, and to view it as a stochastic problem at the outset may mask some of its important characteristics, such as the number m of cost components, or the sequence in which the components are ordered and processed. These characteristics may potentially be algorithmically exploited. For example, with insight into the problem's structure, one may be able to discover a special deterministic or partially randomized order of processing the component functions that is superior to a uniform randomized order. On the other hand

analysis indicates that in the absence of problem-specific knowledge that can be exploited to select a favorable deterministic order, a uniform randomized order (each component f_i chosen with equal probability $1/m$ at each iteration, independently of preceding choices) sometimes has superior worst-case complexity; see [NeB00], [NeB01], [BNO03], [Ber15a], [WaB16].

Finally, let us note the popular hybrid technique, which reshuffles randomly the order of the cost components after each cycle. Practical experience and recent analysis [GOP15c] indicates that it has somewhat better performance to the uniform randomized order when m is large. One possible reason is that random reshuffling allocates exactly one computation slot to each component in an m-slot cycle, while uniform sampling allocates one computation slot to each component *on the average*. A nonzero variance in the number of slots that any fixed component gets within a cycle, may be detrimental to performance. While it seems difficult to establish this fact analytically, a justification is suggested by the view of the incremental gradient method as a gradient method with error in the computation of the gradient. The error has apparently greater variance in the uniform sampling method than in the random reshuffling method, and heuristically, if the variance of the error is larger, the direction of descent deteriorates, suggesting slower convergence.

Incremental Aggregated Gradient Methods

Another algorithm that is well suited for least squares training problems is the *incremental aggregated gradient method*, which has the form

$$y^{k+1} = y^k - \gamma^k \sum_{\ell=0}^{m-1} \nabla f_{i_{k-\ell}}(y^{k-\ell}), \tag{3.10}$$

where f_{i_k} is the new component function selected for iteration k.† In the most common version of the method the component indexes i_k are selected in a cyclic order $[i_k = (k \text{ modulo } m) + 1]$. Random selection of the index i_k has also been suggested.

From Eq. (3.10) it can be seen that the method computes the gradient incrementally, one component per iteration. However, in place of the single component gradient $\nabla f_{i_k}(y^k)$, used in the incremental gradient method (3.6), it uses the sum of the component gradients computed in the past m iterations, which is an approximation to the total cost gradient $\nabla f(y^k)$.

The idea of the method is that by aggregating the component gradients one may be able to reduce the error between the true gradient $\nabla f(y^k)$ and the incrementally computed approximation used in Eq. (3.10), and thus attain a faster asymptotic convergence rate. Indeed, it turns out that

---

† In the case where $k < m$, the summation in Eq. (3.10) should go up to $\ell = k$, and the stepsize should be replaced by a corresponding larger value.

under certain conditions the method exhibits a linear convergence rate, just like in the nonincremental gradient method, without incurring the cost of a full gradient evaluation at each iteration (a strongly convex cost function and with a sufficiently small constant stepsize are required for this; see [Ber16], Section 2.4.2, and the references quoted there). This is in contrast with the incremental gradient method (3.6), for which a linear convergence rate can be achieved only at the expense of asymptotic error, as discussed earlier.

A disadvantage of the aggregated gradient method (3.10) is that it requires that the most recent component gradients be kept in memory, so that when a component gradient is reevaluated at a new point, the preceding gradient of the same component is discarded from the sum of gradients of Eq. (3.10). There have been alternative implementations that ameliorate this memory issue, by recalculating the full gradient periodically and replacing an old component gradient by a new one. More specifically, instead of the gradient sum

$$s^k = \sum_{\ell=0}^{m-1} \nabla f_{i_{k-\ell}}(y^{k-\ell}),$$

in Eq. (3.10), these methods use

$$\tilde{s}^k = \nabla f_{i_k}(y^k) - \nabla f_{i_k}(\tilde{y}^k) + \sum_{\ell=0}^{m-1} \nabla f_{i_{k-\ell}}(\tilde{y}^k),$$

where $\tilde{y}^k$ is the most recent point where the full gradient has been calculated. To calculate $\tilde{s}_k$ one only needs to compute the difference of the two gradients

$$\nabla f_{i_k}(y^k) - \nabla f_{i_k}(\tilde{y}^k)$$

and add it to the full gradient $\sum_{\ell=0}^{m-1} \nabla f_{i^{k-\ell}}(\tilde{y}^k)$. This bypasses the need for extensive memory storage, and with proper implementation, typically leads to small degradation in performance. However, periodically calculating the full gradient when m is very large can be a drawback. Another potential drawback of the aggregated gradient method is that for a large number of terms m, one hopes to converge within the first cycle through the components f_i, thereby reducing the effect of aggregating the gradients of the components.

Incremental Newton Methods

We will now consider an incremental version of Newton's method for unconstrained minimization of an additive cost function of the form

$$f(y) = \sum_{i=1}^{m} f_i(y),$$

where the functions f_i are convex and twice continuously differentiable.†

The ordinary Newton's method, at the current iterate y^k, obtains the next iterate y^{k+1} by minimizing over y the quadratic approximation/second order expansion

$$\tilde{f}(y; y^k) = \nabla f(y^k)'(y - y^k) + \tfrac{1}{2}(y - y^k)'\nabla^2 f(y^k)(y - y^k),$$

of f at y^k. Similarly, the incremental form of Newton's method minimizes a sum of quadratic approximations of components of the general form

$$\tilde{f}_i(y; \psi) = \nabla f_i(\psi)'(y - \psi) + \tfrac{1}{2}(y - \psi)'\nabla^2 f_i(\psi)(y - \psi), \tag{3.11}$$

as we will now explain.

As in the case of the incremental gradient method, we view an iteration as a cycle of m subiterations, each involving a single additional component f_i, and its gradient and Hessian at the current point within the cycle. In particular, if y^k is the vector obtained after k cycles, the vector y^{k+1} obtained after one more cycle is

$$y^{k+1} = \psi_{m,k},$$

where starting with $\psi_{0,k} = y^k$, we obtain $\psi_{m,k}$ after the m steps

$$\psi_{i,k} \in \arg\min_{y \in \Re^n} \sum_{\ell=1}^{i} \tilde{f}_\ell(y; \psi_{\ell-1,k}), \qquad i = 1, \ldots, m, \tag{3.12}$$

where $\tilde{f}_\ell$ is defined as the quadratic approximation (3.11). If all the functions f_i are quadratic, it can be seen that the method finds the solution in a single cycle.‡ The reason is that when f_i is quadratic, each $f_i(y)$ differs from $\tilde{f}_i(y; \psi)$ by a constant, which does not depend on y. Thus the difference

$$\sum_{i=1}^{m} f_i(y) - \sum_{i=1}^{m} \tilde{f}_i(y; \psi_{i-1,k})$$

---

† We will denote by $\nabla^2 f(y)$ the $n \times n$ Hessian matrix of f at y, i.e., the matrix whose (i,j)th component is the second partial derivative $\partial^2 f(y)/\partial y^i \partial y^j$. A beneficial consequence of assuming convexity of f_i is that the Hessian matrices $\nabla^2 f_i(y)$ are positive semidefinite, which facilitates the implementation of the algorithms to be described. On the other hand, the algorithmic ideas of this section may also be adapted for the case where f_i are nonconvex.

‡ Here we assume that the m quadratic minimizations (3.12) to generate $\psi_{m,k}$ have a solution. For this it is sufficient that the first Hessian matrix $\nabla^2 f_1(y^0)$ be positive definite, in which case there is a unique solution at every iteration. A simple possibility to deal with this requirement is to add to f_1 a small positive regularization term, such as $\frac{\epsilon}{2}\|y - y^0\|^2$. A more sound possibility is to lump together several of the component functions (enough to ensure that the sum of their quadratic approximations at y^0 is positive definite), and to use them in place of f_1. This is generally a good idea and leads to smoother initialization, as it ensures a relatively stable behavior of the algorithm for the initial iterations.

is a constant that is independent of y, and minimization of either sum in the above expression gives the same result.

It is important to note that the computations of Eq. (3.12) can be carried out efficiently. For simplicity, let as assume that $\tilde{f}_1(y;\psi)$ is a positive definite quadratic, so that for all i, $\psi_{i,k}$ is well defined as the unique solution of the minimization problem in Eq. (3.12). We will show that the incremental Newton method (3.12) can be implemented in terms of the incremental update formula

$$\psi_{i,k} = \psi_{i-1,k} - D_{i,k}\nabla f_i(\psi_{i-1,k}), \tag{3.13}$$

where $D_{i,k}$ is given by

$$D_{i,k} = \left(\sum_{\ell=1}^{i} \nabla^2 f_\ell(\psi_{\ell-1,k})\right)^{-1}, \tag{3.14}$$

and is generated iteratively as

$$D_{i,k} = \left(D_{i-1,k}^{-1} + \nabla^2 f_i(\psi_{i,k})\right)^{-1}. \tag{3.15}$$

Indeed, from the definition of the method (3.12), the quadratic function $\sum_{\ell=1}^{i-1} \tilde{f}_\ell(y;\psi_{\ell-1,k})$ is minimized by $\psi_{i-1,k}$ and its Hessian matrix is $D_{i-1,k}^{-1}$, so we have

$$\sum_{\ell=1}^{i-1} \tilde{f}_\ell(y;\psi_{\ell-1,k}) = \tfrac{1}{2}(y-\psi_{\ell-1,k})' D_{i-1,k}^{-1}(y-\psi_{\ell-1,k}) + \text{constant}.$$

Thus, by adding $\tilde{f}_i(y;\psi_{i-1,k})$ to both sides of this expression, we obtain

$$\begin{aligned}\sum_{\ell=1}^{i} \tilde{f}_\ell(y;\psi_{\ell-1,k}) &= \tfrac{1}{2}(y-\psi_{\ell-1,k})' D_{i-1,k}^{-1}(y-\psi_{\ell-1,k}) + \text{constant} \\ &\quad + \tfrac{1}{2}(y-\psi_{i-1,k})'\nabla^2 f_i(\psi_{i-1,k})(y-\psi_{i-1,k}) + \nabla f_i(\psi_{i-1,k})'(y-\psi_{i-1,k}).\end{aligned}$$

Since by definition $\psi_{i,k}$ minimizes this function, we obtain Eqs. (3.13)-(3.15).

The recursion (3.15) for the matrix $D_{i,k}$ can often be efficiently implemented by using convenient formulas for the inverse of the sum of two matrices. In particular, if f_i is given by

$$f_i(y) = h_i(a_i' y - b_i),$$

for some twice differentiable convex function $h_i : \Re \mapsto \Re$, vector a_i, and scalar b_i, we have

$$\nabla^2 f_i(\psi_{i-1,k}) = \nabla^2 h_i(\psi_{i-1,k})\, a_i a_i',$$

and the recursion (3.15) can be written as

$$D_{i,k} = D_{i-1,k} - \frac{D_{i-1,k} a_i a_i' D_{i-1,k}}{\nabla^2 h_i(\psi_{i-1,k})^{-1} + a_i' D_{i-1,k} a_i};$$

this is the well-known Sherman-Morrison formula for the inverse of the sum of an invertible matrix and a rank-one matrix.

We have considered so far a single cycle of the incremental Newton method. Similar to the case of the incremental gradient method, we may cycle through the component functions multiple times. In particular, we may apply the incremental Newton method to the extended set of components

$$f_1, f_2, \ldots, f_m, \ f_1, f_2, \ldots, f_m, \ f_1, f_2, \ldots .$$

The resulting method asymptotically resembles an incremental gradient method with diminishing stepsize of the type described earlier. Indeed, from Eq. (3.14)], the matrix $D_{i,k}$ diminishes roughly in proportion to $1/k$. From this it follows that the asymptotic convergence properties of the incremental Newton method are similar to those of an incremental gradient method with diminishing stepsize of order $O(1/k)$. Thus its convergence rate is slower than linear.

To accelerate the convergence of the method one may employ a form of restart, so that $D_{i,k}$ does not converge to 0. For example $D_{i,k}$ may be reinitialized and increased in size at the beginning of each cycle. For problems where f has a unique nonsingular minimum y^* [one for which $\nabla^2 f(y^*)$ is nonsingular], one may design incremental Newton schemes with restart that converge linearly to within a neighborhood of y^* (and even superlinearly if y^* is also a minimum of all the functions f_i, so there is no region of confusion). Alternatively, the update formula (3.15) may be modified to

$$D_{i,k} = \Big(\beta_k D_{i-1,k}^{-1} + \nabla^2 f_\ell(\psi_{i,k})\Big)^{-1}, \tag{3.16}$$

by introducing a parameter $\beta_k \in (0, 1)$, which can be used to accelerate the practical convergence rate of the method; cf. the discussion of the incremental Gauss-Newton methods.

Incremental Newton Method with Diagonal Approximation

Generally, with proper implementation, the incremental Newton method is often substantially faster than the incremental gradient method, in terms of numbers of iterations (there are theoretical results suggesting this property for stochastic versions of the two methods; see the end-of-chapter references). However, in addition to computation of second derivatives, the incremental Newton method involves greater overhead per iteration due to the matrix-vector calculations in Eqs. (3.13), (3.15), and (3.16), so it is suitable only for problems where n, the dimension of y, is relatively small.

A way to remedy in part this difficulty is to approximate $\nabla^2 f_i(\psi_{i,k})$ by a diagonal matrix, and recursively update a diagonal approximation of $D_{i,k}$ using Eqs. (3.15) or (3.16). In particular, one may set to 0 the off-diagonal components of $\nabla^2 f_i(\psi_{i,k})$. In this case, the iteration (3.13) becomes a diagonally scaled version of the incremental gradient method, and involves comparable overhead per iteration (assuming the required diagonal second derivatives are easily computed or approximated). As an additional scaling option, one may multiply the diagonal components with a stepsize parameter that is close to 1 and add a small positive constant (to bound them away from 0). Ordinarily this method is easily implementable, and requires little experimentation with stepsize selection.

3.2 NEURAL NETWORKS

There are several different types of neural networks that can be used for a variety of tasks, such as pattern recognition, classification, image and speech recognition, and others. We focus here on our finite horizon DP context, and the role that neural networks can play in approximating the optimal cost-to-go functions J_k^*. As an example within this context, we may first use a neural network to construct an approximation to J_{N-1}^*. Then we may use this approximation to approximate J_{N-2}^*, and continue this process backwards in time, to obtain approximations to all the optimal cost-to-go functions J_k^*, $k = 1, \ldots, N-1$, as we will discuss in more detail in Section 3.3.

To describe the use of neural networks in finite horizon DP, let us consider the typical stage k, and for convenience drop the index k; the subsequent discussion applies to each value of k separately. We consider parametric architectures $\tilde{J}(x, v, r)$ of the form

$$\tilde{J}(x, v, r) = r'\phi(x, v) \tag{3.17}$$

that depend on two parameter vectors v and r. Our objective is to select v and r so that $\tilde{J}(x, v, r)$ approximates some cost function that can be sampled (possibly with some error). The process is to collect a training set that consists of a large number of state-cost pairs (x^s, β^s), $s = 1, \ldots, q$, and to find a function $\tilde{J}(x, v, r)$ of the form (3.17) that matches the training set in a least squares sense, i.e., (v, r) minimizes

$$\sum_{s=1}^{q} \big(\tilde{J}(x^s, v, r) - \beta^s\big)^2.$$

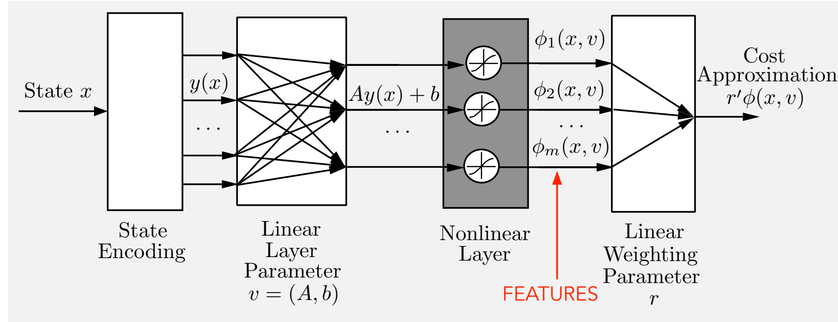

Figure 3.2.1 A perceptron consisting of a linear layer and a nonlinear layer. It provides a way to compute features of the state, which can be used for cost function approximation. The state x is encoded as a vector of numerical values $y(x)$, which is then transformed linearly as $Ay(x) + b$ in the linear layer. The m scalar output components of the linear layer, become the inputs to nonlinear functions that produce the m scalars $\phi_\ell(x, v) = \sigma\big((Ay(x) + b)_\ell\big)$, which can be viewed as features that are in turn linearly weighted with parameters r_ℓ.

We postpone for later the question of how the training pairs (x^s, β^s) are generated, and how the training problem is solved.† Notice the different roles of the two parameter vectors here: *v parametrizes* $\phi(x, v)$, which in some interpretation may be viewed as a feature vector, and *r is a vector of linear weighting parameters for the components of* $\phi(x, v)$.

A neural network architecture provides a parametric class of functions $\tilde J(x, v, r)$ of the form (3.17) that can be used in the optimization framework just described. The simplest type of neural network is the *single layer perceptron*; see Fig. 3.2.1. Here the state x is encoded as a vector of numerical values $y(x)$ with components $y_1(x), \ldots, y_n(x)$, which is then transformed linearly as

$$Ay(x) + b,$$

where A is an $m \times n$ matrix and b is a vector in $\Re^m$.‡ This transformation is called the *linear layer* of the neural network. We view the components of A and b as parameters to be determined, and we group them together into the parameter vector $v = (A, b)$.

---

† The least squares training problem used here is based on *nonlinear regression*. This is a classical method for approximating the expected value of a function with a parametric architecture, and involves a least squares fit of the architecture to simulation-generated samples of the expected value. We refer to machine learning and statistics textbooks for more discussion.

‡ The method of encoding x into the numerical vector $y(x)$ is problem-dependent, but it is important to note that some of the components of $y(x)$ could be some known interesting features of x that can be designed based on problem-specific knowledge.

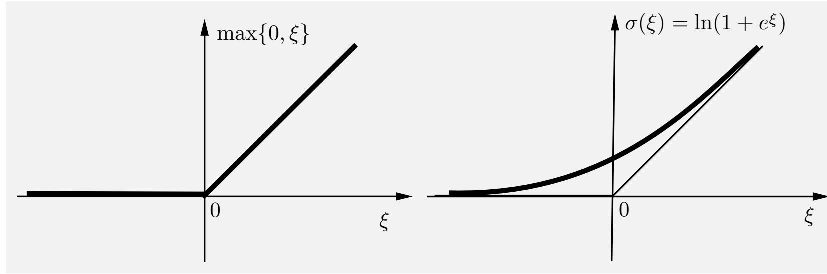

Figure 3.2.2 The rectified linear unit $\sigma(\xi) = \ln(1+e^{\xi})$. It is the rectifier function $\max\{0, \xi\}$ with its corner "smoothed out." Its derivative is $\sigma'(\xi) = e^{\xi}/(1 + e^{\xi})$, and approaches 0 and 1 as $\xi \to -\infty$ and $\xi \to \infty$, respectively.

Each of the m scalar output components of the linear layer,

$$\big(Ay(x) + b\big)_{\ell}, \qquad \ell = 1, \ldots, m,$$

becomes the input to a nonlinear differentiable function σ that maps scalars to scalars. Typically σ is differentiable and monotonically increasing. A simple and popular possibility is based on the *rectifier*, which is simply the function $\max\{0, \xi\}$. In neural networks it is approximated by a differentiable function σ by some form of smoothing operation, and it is called *rectified linear unit* (ReLU for short); for example $\sigma(\xi) = \ln(1 + e^{\xi})$, which illustrated in Fig. 3.2.2. Other functions, used since the early days of neural networks, have the property

$$-\infty < \lim_{\xi \to -\infty} \sigma(\xi) < \lim_{\xi \to \infty} \sigma(\xi) < \infty;$$

see Fig. 3.2.3. Such functions are called *sigmoids*, and some common choices are the *hyperbolic tangent* function

$$\sigma(\xi) = \tanh(\xi) = \frac{e^{\xi} - e^{-\xi}}{e^{\xi} + e^{-\xi}},$$

and the *logistic* function

$$\sigma(\xi) = \frac{1}{1 + e^{-\xi}}.$$

In what follows, we will ignore the character of the function σ (except for differentiability), and simply refer to it as a "nonlinear unit" and to the corresponding layer as a "nonlinear layer."

At the outputs of the nonlinear units, we obtain the scalars

$$\phi_{\ell}(x, v) = \sigma\big((Ay(x) + b)_{\ell}\big), \qquad \ell = 1, \ldots, m.$$

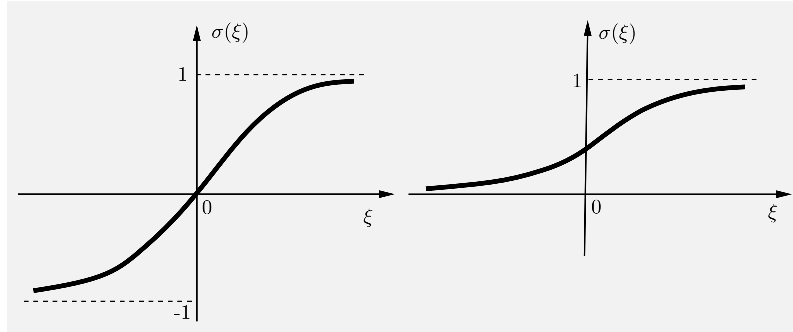

Figure 3.2.3 Some examples of sigmoid functions. The hyperbolic tangent function is on the left, while the logistic function is on the right.

One possible interpretation is to view $\phi_\ell(x, v)$ as features of x, which are linearly combined using weights r_ℓ, $\ell = 1, \ldots, m$, to produce the final output

$$\tilde{J}(x, v, r) = \sum_{\ell=1}^{m} r_\ell \phi_\ell(x, v) = \sum_{\ell=1}^{m} r_\ell \sigma\big((Ay(x) + b)_\ell\big).$$

Note that each value $\phi_\ell(x, v)$ depends on just the ℓth row of A and the ℓth component of b, not on the entire vector v. In some cases this motivates placing some constraints on individual components of A and b to achieve special problem-dependent "handcrafted" effects.

State Encoding and Direct Feature Extraction

The state encoding operation that transforms x into the neural network input $y(x)$ can be instrumental in the success of the approximation scheme. Examples of possible state encodings are components of the state x, numerical representations of qualitative characteristics of x, and more generally features of x, i.e., functions of x that aim to capture "important nonlinearities" of the optimal cost-to-go function. With the latter view of state encoding, we may consider the approximation process as consisting of a feature extraction mapping, followed by a neural network with input the extracted features of x, and output the cost-to-go approximation; see Fig. 3.2.4.

The idea here is that with a good feature extraction mapping, the neural network need not be very complicated (may involve few nonlinear units and corresponding parameters), and may be trained more easily. This intuition is borne out by simple examples and practical experience. However, as is often the case with neural networks, it is hard to support it with a quantitative analysis.

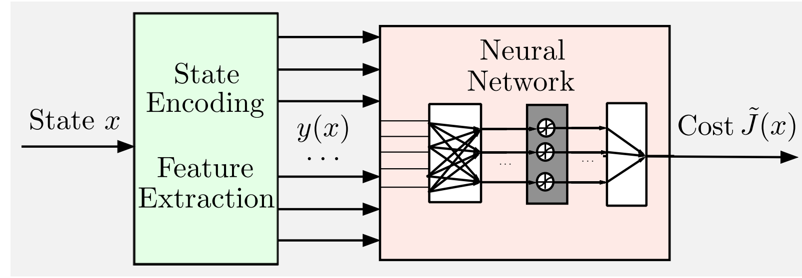

Figure 3.2.4 Nonlinear architecture with a view of the state encoding process as a feature extraction mapping preceding the neural network.

3.2.1 Training of Neural Networks

Given a set of state-cost training pairs (x^s, β^s), $s = 1, \ldots, q$, the parameters of the neural network A, b, and r are obtained by solving the training problem

$$\min_{A,b,r} \sum_{s=1}^{q} \left(\sum_{\ell=1}^{m} r_\ell \sigma\left(\big(Ay(x^s) + b\big)_\ell\right) - \beta^s \right)^2. \tag{3.18}$$

Note that the cost function of this problem is generally nonconvex, so there may exist multiple local minima.

In practice it is common to augment the cost function of this problem with a *regularization* function, such as a quadratic in the parameters A, b, and r. This is customary in least squares problems in order to make the problem easier to solve algorithmically. However, in the context of neural network training, regularization is primarily important for a different reason: it helps to avoid *overfitting*, which refers to a situation where a neural network model matches the training data very well but does not do as well on new data. This is a well known difficulty in machine learning, which may occur when the number of parameters of the neural network is relatively large (roughly comparable to the size of the training set). We refer to machine learning and neural network textbooks for a discussion of algorithmic questions regarding regularization and other issues that relate to the practical implementation of the training process. In any case, the training problem (3.18) is an unconstrained nonconvex differentiable optimization problem that can in principle be addressed with any of the standard gradient-type methods. Significantly, it is well-suited for the incremental methods discussed in Section 3.1.3.

Let us now consider a few issues regarding the neural network formulation and training process just described:

(a) The first issue is to select a method to solve the training problem (3.18). While we can use any unconstrained optimization method that

is based on gradients, in practice it is important to take into account the cost function structure of problem (3.18). The salient characteristic of this cost function is that it is the sum of a potentially very large number q of component functions. This makes the computation of the cost function value of the training problem and/or its gradient very costly. For this reason the incremental methods of Section 3.1.3 are universally used for training.† Experience has shown that these methods can be vastly superior to their nonincremental counterparts in the context of neural network training.

The implementation of the training process has benefited from experience that has been accumulated over time, and has provided guidelines for scaling, regularization, initial parameter selection, and other practical issues; we refer to books on neural networks such as Bishop [Bis95], Goodfellow, Bengio, and Courville [GBC16], and Haykin [Hay09] for related accounts. Still, incremental methods can be quite slow, and training may be a time-consuming process. Fortunately, the training is ordinarily done off-line, in which case computation time may not be a serious issue. Moreover, in practice the neural network training problem typically need not be solved with great accuracy.

(b) Another important question is how well we can approximate the optimal cost-to-go functions J_k^* with a neural network architecture, assuming we can choose the number of the nonlinear units m to be as large as we want. The answer to this question is quite favorable and is provided by the so-called *universal approximation theorem*. Roughly, the theorem says that assuming that x is an element of a Euclidean space X and $y(x) \equiv x$, a neural network of the form described can approximate arbitrarily closely (in an appropriate mathematical sense), over a closed and bounded subset $S \subset X$, any piecewise continuous function $J : S \mapsto \Re$, provided the number m of nonlinear units is sufficiently large. For proofs of the theorem at different levels of generality, we refer to Cybenko [Cyb89], Funahashi [Fun89], Hornik, Stinchcombe, and White [HSW89], and Leshno et al. [LLP93]. For intuitive explanations we refer to Bishop ([Bis95], pp. 129-130) and Jones [Jon90].

(c) While the universal approximation theorem provides some assurance about the adequacy of the neural network structure, it does not pre-

† The incremental methods are valid for an arbitrary order of component selection within the cycle, but it is common to randomize the order at the beginning of each cycle. Also, in a variation of the basic method, we may operate on a batch of several components at each iteration, called a *minibatch*, rather than a single component. This has an averaging effect, which reduces the tendency of the method to oscillate and allows for the use of a larger stepsize; see the end-of-chapter references.

dict how many nonlinear units we may need for "good" performance in a given problem. Unfortunately, this is a difficult question to even pose precisely, let alone to answer adequately. In practice, one is reduced to trying increasingly larger values of m until one is convinced that satisfactory performance has been obtained for the task at hand. Experience has shown that in many cases the number of required nonlinear units and corresponding dimension of A can be very large, adding significantly to the difficulty of solving the training problem. This has given rise to many suggestions for modifications of the neural network structure. One possibility is to concatenate multiple single layer perceptrons so that the output of the nonlinear layer of one perceptron becomes the input to the linear layer of the next, as we will now discuss.

What are the Features that can be Produced by Neural Networks?

The short answer is that just about any feature that can be of practical interest can be produced or be closely approximated by a neural network. What is needed is a single layer that consists of a sufficiently large number of nonlinear units, preceded and followed by a linear layer. This is a consequence of the universal approximation theorem. In particular it is not necessary to have more than one nonlinear layer (although it is possible that fewer nonlinear units may be needed with a deep neural network, involving more than one nonlinear layer).

To illustrate this fact, we will consider features of a scalar state variable x, and a neural network that uses the rectifier function

$$\sigma(\xi) = \max\{0, \xi\}$$

as the basic nonlinear unit. Thus a single rectifier preceded by a linear function

$$L(x) = \gamma(x - \beta),$$

where β and γ are scalars, produces the feature

$$\phi_{\beta,\gamma}(x) = \max\big\{0, \gamma(x - \beta)\big\}, \tag{3.19}$$

illustrated in Fig. 3.2.6.

We can now construct more complex features by adding or subtracting single rectifier features of the form (3.19). In particular, subtracting two rectifier functions with the same slope but two different horizontal shifts β_1 and β_2, we obtain the feature

$$\phi_{\beta_1,\beta_2,\gamma}(x) = \phi_{\beta_1,\gamma}(x) - \phi_{\beta_2,\gamma}(x)$$

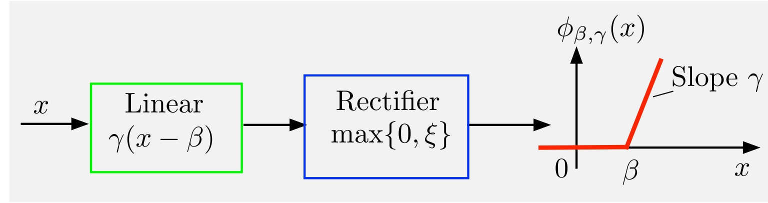

Figure 3.2.6 The feature

$$\phi_{\beta,\gamma}(x) = \max\Big\{0, \gamma(x-\beta)\Big\},$$

produced by a rectifier preceded by the linear function $L(x) = \gamma(x-\beta)$.

shown in Fig. 3.2.7(a). By subtracting again two features of the preceding form, we obtain the "pulse" feature

$$\phi_{\beta_1,\beta_2,\beta_3,\beta_4,\gamma}(x) = \phi_{\beta_1,\beta_2,\gamma}(x) - \phi_{\beta_3,\beta_4,\gamma}(x), \tag{3.20}$$

shown in Fig. 3.2.7(b). The "pulse" feature can be used in turn as a fundamental block to approximate any desired feature by a linear combination of "pulses." This explains how neural networks produce features of any shape by using linear layers to precede and follow nonlinear layers, at least in the case of a scalar state x. In fact, the mechanism for feature formation just described can be extended to the case of a multidimensional x, and is at the heart of the universal approximation theorem and its proof (see Cybenko [Cyb89]).

3.2.2 Multilayer and Deep Neural Networks

An important generalization of the single layer perceptron architecture involves a concatenation of multiple layers of linear and nonlinear functions; see Fig. 3.2.5. In particular the outputs of each nonlinear layer become the inputs of the next linear layer. In some cases it may make sense to add as additional inputs some of the components of the state x or the state encoding $y(x)$.

There are a few questions to consider here. The first has to do with the reason for having multiple nonlinear layers, when a single one is sufficient to guarantee the universal approximation property. Here are some qualitative explanations:

(a) If we view the outputs of each nonlinear layer as features, we see that the multilayer network provides a sequence of features, where each set of features in the sequence is a function of the preceding set of features in the sequence [except for the first set of features, which is a function of the encoding $y(x)$ of the state x]. Thus the

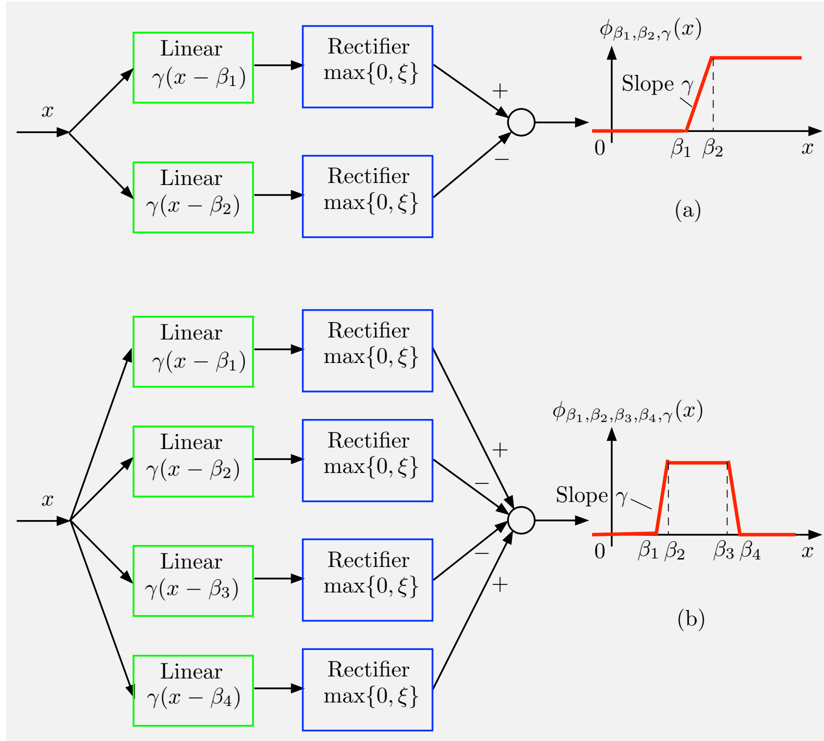

Figure 3.2.6 (a) Illustration of how the feature

$$\phi_{\beta_1,\beta_2,\gamma}(x) = \phi_{\beta_1,\gamma}(x) - \phi_{\beta_2,\gamma}(x)$$

is formed by a neural network of a two-rectifier nonlinear layer, preceded by a linear layer.
(b) Illustration of how the "pulse" feature

$$\phi_{\beta_1,\beta_2,\beta_3,\beta_4,\gamma}(x) = \phi_{\beta_1,\beta_2,\gamma}(x) - \phi_{\beta_3,\beta_4,\gamma}(x)$$

is formed by a neural network of a four-rectifier nonlinear layer, preceded by a linear layer.

network produces a hierarchy of features. In the context of specific applications, this hierarchical structure can be exploited in order to specialize the role of some of the layers and to enhance particular characteristics of the state.

(b) Given the presence of multiple linear layers, one may consider the possibility of using matrices A with a particular sparsity pattern, or other structure that embodies special linear operations such as convolution. When such structures are used, the training problem

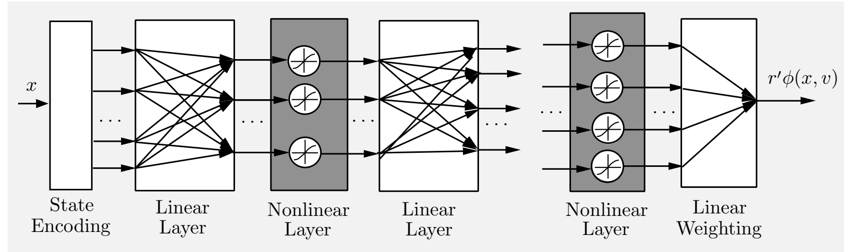

Figure 3.2.5 A neural network with multiple layers. Each nonlinear layer constructs the inputs of the next linear layer.

often becomes easier, because the number of parameters in the linear layers is drastically decreased.

Note that while in the early days of neural networks practitioners tended to use few nonlinear layers (say one to three), more recently a lot of success in certain problem domains (including image and speech processing, as well as approximate DP) has been achieved with so called *deep neural networks*, which involve a considerably larger number of layers. In particular, the use of deep neural networks, in conjunction with Monte Carlo tree search, has been an important factor for the success of the computer programs AlphaGo and AlphaZero, which perform better than the best humans in the games of Go and chess; see the papers by Silver et al. [SHM16], [SHS17]. By contrast, Tesauro's backgammon program and its descendants (cf. Section 2.4.2) do not require multiple nonlinear layers for good performance at present.

Training and Backpropagation

Let us now consider the training problem for multilayer networks. It has the form

$$\min_{v,r} \sum_{s=1}^{q} \left(\sum_{\ell=1}^{m} r_\ell \phi_\ell(x^s, v) - \beta^s \right)^2,$$

where v represents the collection of all the parameters of the linear layers, and $\phi_\ell(x,v)$ is the ℓth feature produced at the output of the final nonlinear layer. Various types of incremental gradient methods, which modify the weight vector in the direction opposite to the gradient of a single sample term

$$\left(\sum_{\ell=1}^{m} r_\ell \phi_\ell(x^s, v) - \beta^s \right)^2,$$

can also be applied here. They are the methods most commonly used in practice. An important fact is that the gradient with respect to v of

each feature $\phi_\ell(x, v)$ can be efficiently calculated using a special procedure known as *backpropagation*. This is just an intelligent way of applying the chain rule of differentiation, as we will explain now.

Multilayer perceptrons can be represented compactly by introducing certain mappings to describe the linear and the nonlinear layers. In particular, let $L_1, \ldots, L_{m+1}$ denote the matrices representing the linear layers; that is, at the output of the 1st linear layer we obtain the vector $L_1 x$ and at the output of the kth linear layer ($k > 1$) we obtain $L_k \xi$, where ξ is the output of the preceding nonlinear layer. Similarly, let $\Sigma_1, \ldots, \Sigma_m$ denote the mappings representing the nonlinear layers; that is, when the input of the kth nonlinear layer ($k > 1$) is the vector y with components $y(j)$, we obtain at the output the vector $\Sigma_k y$ with components $\sigma\big(y(j)\big)$. The output of the multilayer perceptron is

$$F(L_1, \ldots, L_{m+1}, x) = L_{m+1} \Sigma_m L_m \cdots \Sigma_1 L_1 x.$$

The special nature of this formula has an important computational consequence: the gradient (with respect to the weights) of the squared error between the output and a desired output y,

$$E(L_1, \ldots, L_{m+1}) = \frac{1}{2}\big(y - F(L_1, \ldots, L_{m+1}, x)\big)^2,$$

can be efficiently calculated using a special procedure known as *backpropagation*, which is just an intelligent way of using the chain rule.† In particular, the partial derivative of the cost function $E(L_1, \ldots, L_{m+1})$ with respect to $L_k(i, j)$, the ijth component of the matrix L_k, is given by

$$\frac{\partial E(L_1, \ldots, L_{m+1})}{\partial L_k(i, j)} = -e' L_{m+1} \overline{\Sigma}_m L_m \cdots L_{k+1} \overline{\Sigma}_k I_{ij} \Sigma_{k-1} L_{k-1} \cdots \Sigma_1 L_1 x, \tag{3.21}$$

where e is the error vector

$$e = y - F(L_1, \ldots, L_{m+1}, x),$$

$\overline{\Sigma}_n$, $n = 1, \ldots, m$, is the diagonal matrix with diagonal terms equal to the derivatives of the nonlinear functions σ of the nth hidden layer evaluated at the appropriate points, and I_{ij} is the matrix obtained from L_k by setting all of its components to 0 except for the ijth component which is set to 1. This formula can be used to obtain efficiently all of the terms needed in the partial derivatives (3.21) of E using a two-step calculation:

---

† The name *backpropagation* is used in several different ways in the neural networks literature. For example feedforward neural networks of the type shown in Fig. 3.2.5 are sometimes referred to as *backpropagation networks*. The somewhat abstract derivation of the backpropagation formulas given here comes from Section 3.1.1 of the book [BeT96].

(a) Use a forward pass through the network to calculate sequentially the outputs of the linear layers

$$L_1x, L_2\Sigma_1 L_1 x, \ldots, L_{m+1}\Sigma_m L_m \cdots \Sigma_1 L_1 x.$$

This is needed in order to obtain the points at which the derivatives in the matrices $\overline{\Sigma}_n$ are evaluated, and also in order to obtain the error vector $e = y - F(L_1, \ldots, L_{m+1}, x)$.

(b) Use a backward pass through the network to calculate sequentially the terms

$$e' L_{m+1} \overline{\Sigma}_m L_m \cdots L_{k+1} \overline{\Sigma}_k$$

in the derivative formulas (3.21), starting with $e' L_{m+1}\overline{\Sigma}_m$, proceeding to $e' L_{m+1}\overline{\Sigma}_m L_m \overline{\Sigma}_{m-1}$, and continuing to $e' L_{m+1}\overline{\Sigma}_m \cdots L_2 \overline{\Sigma}_1$.

As a final remark, we mention that the ability to simultaneously extract features and optimize their linear combination is not unique to neural networks. Other approaches that use a multilayer architecture have been proposed (see the survey by Schmidhuber [Sch15]), and they admit similar training procedures based on appropriately modified forms of backpropagation. An example of an alternative multilayer architecture approach is the Group Method for Data Handling (GMDH), which is principally based on the use of polynomial (rather than sigmoidal) nonlinearities. The GMDH approach was investigated extensively in the Soviet Union starting with the work of Ivakhnenko in the late 60s; see e.g., [Iva68]. It has been used in a large variety of applications, and its similarities with the neural network methodology have been noted (see the survey by Ivakhnenko [Iva71], and the large literature summary at the web site http://www.gmdh.net). Most of the GMDH research relates to inference-type problems, and there have not been any applications to approximate DP to date. However, this may be a fruitful area of investigation, since in some applications it may turn out that polynomial nonlinearities are more suitable than sigmoidal or rectified linear unit nonlinearities.

3.3 SEQUENTIAL DYNAMIC PROGRAMMING APPROXIMATION

Let us describe a popular approach for training an approximation architecture $\tilde{J}_k(x_k, r_k)$ for a finite horizon DP problem. The parameter vectors r_k are determined sequentially, with an algorithm known as *fitted value iteration*, starting from the end of the horizon, and proceeding backwards as in the DP algorithm: first r_{N-1} then r_{N-2}, and so on, see Fig. 3.3.1. The algorithm samples the state space for each stage k, and generates a large number of states x_k^s, $s = 1, \ldots, q$. It then determines sequentially

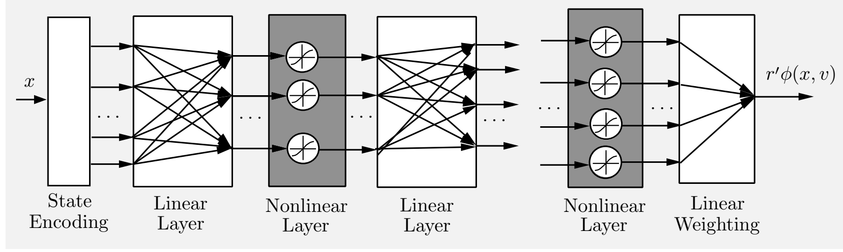

Figure 3.3.1 Illustration of fitted value iteration.

the parameter vectors r_k to obtain a good "least squares fit" to the DP algorithm.

In particular, each r_k is determined by generating a large number of sample states and solving a least squares problem that aims to minimize the error in satisfying the DP equation for these states at time k. At the typical stage k, having obtained r_{k+1}, we determine r_k from the least squares problem

$$r_k \in \arg\min_r \sum_{s=1}^{q} \Bigg(\tilde{J}_k(x_k^s, r) - \min_{u \in U_k(x_k^s)} E\Big\{ g(x_k^s, u, w_k) + \tilde{J}_{k+1}\big(f_k(x_k^s, u, w_k), r_{k+1}\big) \Big\} \Bigg)^2$$

where x_k^s, $i = 1, \ldots, q$, are the sample states that have been generated for the kth stage. Since r_{k+1} is assumed to be already known, the complicated minimization term in the right side of this equation is the known scalar

$$\beta_k^s = \min_{u \in U_k(x_k^s)} E\Big\{ g(x_k^s, u, w_k) + \tilde{J}_{k+1}\big(f_k(x_k^s, u, w_k), r_{k+1}\big) \Big\},$$

so that r_k is obtained as

$$r_k \in \arg\min_r \sum_{s=1}^{q} \big(\tilde{J}_k(x_k^s, r) - \beta_k^s\big)^2. \tag{3.22}$$

The algorithm starts at stage $N-1$ with the minimization

$$r_{N-1} \in \arg\min_r \sum_{s=1}^{q} \Bigg(\tilde{J}_{N-1}(x_{N-1}^s, r) - \min_{u \in U_{N-1}(x_{N-1}^s)} E\Big\{ g_{N-1}(x_{N-1}^s, u, w_{N-1}) + g_N\big(f_{N-1}(x_{N-1}^s, u, w_{N-1})\big) \Big\} \Bigg)^2$$

and ends with the calculation of r_0 at time $k=0$.

In the case of a linear architecture, where the approximate cost-to-go functions are

$$\tilde{J}_k(x_k, r_k) = r_k'\phi_k(x_k), \qquad k = 0, \ldots, N-1,$$

the least squares problem (3.22) greatly simplifies, and admits the closed form solution

$$r_k = \left(\sum_{s=1}^{q} \phi_k(x_k^s)\phi_k(x_k^s)'\right)^{-1} \sum_{s=1}^{q} \beta_k^s \phi_k(x_k^s);$$

cf. Eq. (3.3). For a nonlinear architecture such as a neural network, incremental gradient algorithms may be used.

An important implementation issue is how to select the sample states x_k^s, $s = 1, \ldots, q$, $k = 0, \ldots, N-1$. In practice, they are typically obtained by some form of Monte Carlo simulation, but the distribution by which they are generated is important for the success of the method. In particular, it is important that the sample states are "representative" in the sense that they are visited often under a nearly optimal policy. More precisely, the frequencies with which various states appear in the sample should be roughly proportional to the probabilities of their occurrence under an optimal policy. This point will be discussed later in Chapter 4, in the context of infinite horizon problems, and the notion of "representative" state will be better quantified in probabilistic terms.

Aside from the issue of selection of the sampling distribution that we have just described, a difficulty with fitted value iteration arises when the horizon is very long. In this case, however, the problem is often stationary, in the sense that the system and cost per stage do not change as time progresses. Then it may be possible to treat the problem as one with an infinite horizon and bring to bear additional methods for training approximation architectures; see the discussion in Chapter 4.

3.4 Q-FACTOR PARAMETRIC APPROXIMATION

We will now consider an alternative form of approximation in value space. It involves approximation of the optimal Q-factors of state-control pairs (x_k, u_k) at time k, without an intermediate approximation of cost-to-go functions. The optimal Q-factors are defined by

$$Q_k^*(x_k, u_k) = E\Big\{g_k(x_k, u_k, w_k) + J_{k+1}^*\big(f_k(x_k, u_k, w_k)\big)\Big\}, \quad k = 0, \ldots, N-1, \tag{3.23}$$

where J_{k+1}^* is the optimal cost-to-go function for stage $k+1$. Thus $Q_k^*(x_k, u_k)$ is the cost attained by using u_k at state x_k, and subsequently using an optimal policy.

As noted in Section 1.2, the DP algorithm can be written as

$$J_k^*(x_k) = \min_{u \in U_k(x_k)} Q_k^*(x_k, u_k), \tag{3.24}$$

and by using this equation, we can write Eq. (3.23) in the following equivalent form that relates Q_k^* with Q_{k+1}^*:

$$Q_k^*(x_k, u_k) = E\Big\{ g_k(x_k, u_k, w_k) + \min_{u \in U_{k+1}(f_k(x_k,u_k,w_k))} Q_{k+1}^*\big(f_k(x_k, u_k, w_k), u\big)\Big\}. \tag{3.25}$$

This suggests that in place of the Q-factors $Q_k^*(x_k, u_k)$, we may use Q-factor approximations and Eq. (3.25) as the basis for suboptimal control.

We can obtain such approximations by using methods that are similar to the ones we have considered so far (parametric approximation, enforced decomposition, certainty equivalent control, etc). Parametric Q-factor approximations $\tilde{Q}_k(x_k, u_k, r_k)$ may involve a neural network, or a feature-based linear architecture. The feature vector may depend on just the state, or on both the state and the control. In the former case, the architecture has the form

$$\tilde{Q}_k(x_k, u_k, r_k) = r_k(u_k)'\phi_k(x_k), \tag{3.26}$$

where $r_k(u_k)$ is a separate weight vector for each control u_k. In the latter case, the architecture has the form

$$\tilde{Q}_k(x_k, u_k, r_k) = r_k'\phi_k(x_k, u_k), \tag{3.27}$$

where r_k is a weight vector that is independent of u_k. The architecture (3.26) is suitable for problems with a relatively small number of control options at each stage. In what follows, we will focus on the architecture (3.27), but the discussion with few modifications, also applies to the architecture (3.26).

We may adapt the fitted value iteration approach of the preceding section to compute sequentially the parameter vectors r_k in Q-factor parametric approximations, starting from $k = N - 1$. This algorithm is based on Eq. (3.25), with r_k obtained by solving least squares problems similar to the ones of Eq. (3.22). As an example, the parameters r_k of the architecture (3.27) are computed sequentially by collecting sample state-control pairs (x_k^s, u_k^s), $s = 1, \ldots, q$, and solving the linear least squares problems

$$r_k \in \arg\min_r \sum_{s=1}^{q} \big(r'\phi_k(x_k^s, u_k^s) - \beta_k^s\big)^2, \tag{3.28}$$

where

$$\beta_k^s = E\Big\{ g_k(x_k^s, u_k^s, w_k) + \min_{u \in U_{k+1}(f_k(x_k^s,u_k^s,w_k))} r_{k+1}'\phi_k\big(f_k(x_k^s, u_k^s, w_k), u\big)\Big\}. \tag{3.29}$$

Thus r_k is obtained through a least squares fit that aims to minimize the squared errors in satisfying Eq. (3.25). Note that the solution of the least squares problem (3.28) can be obtained in closed form as

$$r_k = \left(\sum_{s=1}^{q} \phi_k(x_k^s, u_k^s) \phi_k(x_k^s, u_k^s)' \right)^{-1} \sum_{s=1}^{q} \beta_k^s \phi_k(x_k^s, u_k^s);$$

[cf. Eq. (3.3)]. Once r_k has been computed, the one-step lookahead control $\tilde{\mu}_k(x_k)$ is obtained on-line as

$$\tilde{\mu}_k(x_k) \in \arg \min_{u \in U_k(x_k)} \tilde{Q}_k(x_k, u, r_k), \tag{3.30}$$

without the need to calculate any expected value. This latter property is a primary incentive for using Q-factors in approximate DP, particularly when there are tight constraints on the amount of on-line computation that is possible in the given practical setting.

The samples β_k^s in Eq. (3.29) involve the computation of an expected value. In an alternative implementation, we may replace β_k^s with an average of just a few samples (even a single sample) of the random variable

$$g_k(x_k^s, u_k^s, w_k) + \min_{u \in U_{k+1}(f_k(x_k^s, u_k^s, w_k))} r_{k+1}' \phi_k\big(f_k(x_k^s, u_k^s, w_k), u\big),$$

collected according to the probability distribution of w_k. This distribution may either be known explicitly, or in a model-free situation, through a computer simulator; cf. the discussion of Section 2.1.4. In particular, to implement this scheme, we only need a simulator that for any pair (x_k, u_k) generates a sample of the stage cost $g_k(x_k, u_k, w_k)$ and the next state $f_k(x_k, u_k, w_k)$ according to the distribution of w_k.

Having obtained the one-step lookahead policy $\tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$, a further possibility is to approximate it with a parametric architecture. This is *approximation in policy space built on top of approximation in value space*; see the discussion of Section 2.1.3. The idea here is to simplify even further the on-line computation of the suboptimal controls by avoiding the minimization of Eq. (3.30).

Finally, let us note an alternative to computing Q-factor approximations. It is motivated by the potential benefit of using Q-factor differences in contexts involving approximation. In this method, called *advantage updating*, instead of computing and comparing $Q_k^*(x_k, u_k)$ for all $u_k \in U_k(x_k)$, we compute

$$A_k(x_k, u_k) = Q_k^*(x_k, u_k) - \min_{u_k \in U_k(x_k)} Q_k^*(x_k, u_k).$$

The function $A_k(x_k, u_k)$ can serve just as well as $Q_k^*(x_k, u_k)$ for the purpose of comparing controls, but may have a much smaller range of values

than $Q_k^*(x_k, u_k)$. In the absence of approximations, advantage updating is clearly equivalent to selecting controls by comparing their Q-factors. However, when approximation is involved, using Q-factor differences can be important, because $Q_k^*(x_k, u_k)$ may embody sizable quantities that are independent of u, and which may interfere with algorithms such as the fitted value iteration (3.28)-(3.29). This question is discussed further and is illustrated with an example in the neuro-dynamic programming book [BeT96], Section 6.6.2.

3.5 NOTES AND SOURCES

Our discussion of approximation architectures, neural networks, and training has been limited, and aimed just to provide the connection with approximate DP and a starting point for further exploration. The literature on the subject is vast, and the textbooks mentioned in the references to Chapter 1 provide detailed accounts and many references in addition to the ones given in Sections 3.1.3 and 3.2.1.

There are two broad directions of inquiry in parametric architectures:

(1) The design of architectures, either in a general or a problem-specific context. Research in this area has intensified following the increased popularity of deep neural networks.

(2) The algorithms for training of neural networks as well as linear architectures.

Both of these issues have been extensively investigated and research relating to these and other related issues is continuing.

Incremental algorithms are supported by substantial theoretical analysis, which addresses issues of convergence, rate of convergence, stepsize selection, and component order selection. It is beyond our scope to cover this analysis, and we refer to the book [BeT96] and paper [BeT00] by Bertsekas and Tsitsiklis, and the recent survey by Bottou, Curtis, and Nocedal [BCN18] for theoretically oriented accounts. The author's surveys [Ber10] and [Ber15b], and convex optimization and nonlinear programming textbooks [Ber15a], [Ber16a], collectively contain an extensive account of theoretical and practical issues regarding incremental methods, including the Kaczmarz, incremental gradient, incremental subgradient, incremental aggregated gradient, incremental Newton, and incremental Gauss-Newton methods.

Fitted value iteration has a long history; it has been mentioned by Bellman among others. It has interesting properties, and at times exhibits pathological behavior, such as instability over a long horizon. We will discuss this behavior in Section 4.4 in the context of infinite horizon problems. Advantage updating (Section 3.4) was proposed by Baird [Bai93], [Bai94], and is discussed in Section 6.6 of the book [BeT96].

Reinforcement Learning and Optimal Control

by

Dimitri P. Bertsekas

Massachusetts Institute of Technology

Chapter 4
Infinite Horizon Reinforcement Learning

DRAFT

This is Chapter 4 of the draft textbook "Reinforcement Learning and Optimal Control." The chapter represents "work in progress," and it will be periodically updated. It more than likely contains errors (hopefully not serious ones). Furthermore, its references to the literature are incomplete. Your comments and suggestions to the author at dimitrib@mit.edu are welcome.

The date of last revision is given below. (A "revision" is any version of the chapter that involves the addition or the deletion of at least one paragraph or mathematically significant equation.)

February 13, 2019

4

Infinite Horizon Reinforcement Learning

Contents

4.1. An Overview of Infinite Horizon Problems . . . . p. 3
4.2. Stochastic Shortest Path Problems . . . . p. 6
4.3. Discounted Problems . . . . p. 16
4.4. Exact and Approximate Value Iteration . . . . p. 21
4.5. Policy Iteration . . . . p. 25
4.5.1. Exact Policy Iteration . . . . p. 26
4.5.2. Optimistic and Multistep Lookahead Policy Iteration p. 30
4.5.3. Policy Iteration for Q-factors . . . . p. 32
4.6. Approximation in Value Space - Performance Bounds . . . p. 34
4.6.1. Limited Lookahead Performance Bounds . . . . . . p. 36
4.6.2. Rollout . . . . p. 39
4.6.3. Approximate Policy Iteration . . . . p. 43
4.7. Simulation-Based Policy Iteration with Parametric . . . . .
Approximation . . . . p. 46
4.7.1. Self-Learning and Actor-Critic Systems . . . . . . p. 46
4.7.2. A Model-Based Variant . . . . p. 47
4.7.3. A Model-Free Variant . . . . p. 50
4.7.4. Implementation Issues of Parametric Policy Iteration p. 52
4.8. Q-Learning . . . . p. 55
4.9. Additional Methods - Temporal Differences . . . . p. 58
4.10. Exact and Approximate Linear Programming . . . . . . p. 69

4.11. Approximation in Policy Space . . . . p. 71
4.11.1. Training by Cost Optimization - Policy Gradient and Random Search Methods . . . . p. 73
4.11.2. Expert Supervised Training . . . . p. 80
4.12. Notes and Sources . . . . p. 81
4.13. Appendix: Mathematical Analysis . . . . p. 84
4.13.1. Proofs for Stochastic Shortest Path Problems . . p. 85
4.13.2. Proofs for Discounted Problems . . . . p. 90
4.13.3. Convergence of Exact and Optimistic Policy Iteration . . . . p. 91
4.13.4. Performance Bounds for One-Step Lookahead, Rollout, and Approximate Policy Iteration . . . . p. 93

In this chapter, we first provide an introduction to the theory of infinite horizon problems, and then consider the use of approximate DP/RL methods for suboptimal solution. Infinite horizon problems differ from their finite horizon counterparts in two main respects:

(a) The number of stages is infinite.

(b) The system is stationary, i.e., the system equation, the cost per stage, and the random disturbance statistics do not change from one stage to the next.

The assumption of an infinite number of stages is never satisfied in practice, but is a reasonable approximation for problems involving a finite but very large number of stages. The assumption of stationarity is often satisfied in practice, and in other cases it approximates well a situation where the system parameters vary relatively slowly with time.

Infinite horizon problems give rise to elegant and insightful analysis, and their optimal policies are often simpler than their finite horizon counterparts. For example, optimal policies are typically stationary, i.e., the optimal rule for choosing controls does not change from one stage to the next.

On the other hand, infinite horizon problems generally require a more sophisticated mathematical treatment. Our discussion will be limited to relatively simple finite-state problems. Still some theoretical results will be needed in this chapter. They will be explained intuitively to the extent possible, and their mathematical proofs will be provided in the end-of-chapter appendix.

4.1 AN OVERVIEW OF INFINITE HORIZON PROBLEMS

We will focus on two types of infinite horizon problems, where we aim to minimize the total cost over an infinite number of stages, given by

$$J_\pi(x_0) = \lim_{N\to\infty} \mathop{E}_{\substack{w_k\\ k=0,1,\ldots}} \left\{ \sum_{k=0}^{N-1} \alpha^k g\big(x_k, \mu_k(x_k), w_k\big) \right\};$$

see Fig. 4.1.1. Here, $J_\pi(x_0)$ denotes the cost associated with an initial state x_0 and a policy $\pi = \{\mu_0, \mu_1, \ldots\}$, and α is a positive scalar. The meaning of $\alpha < 1$ is that future costs matter to us less than the same costs incurred at the present time.

Thus the infinite horizon costs of a policy is the limit of its finite horizon costs as the horizon tends to infinity. (We assume that the limit exists for the moment, and address the issue later.) The two types of problems, considered in Sections 4.2 and 4.3, respectively, are:

(a) *Stochastic shortest path problems* (SSP for short). Here, $\alpha = 1$ but there is a special cost-free termination state; once the system reaches

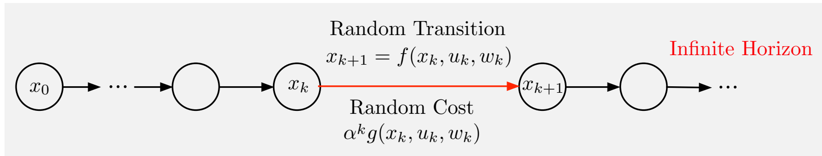

Figure 4.1.1 Illustration of an infinite horizon problem. The syatem and cost per stage are stationary, except for the use of a discount factor α. If $\alpha = 1$, there is typically a special cost-free termination state that we aim to reach.

that state it remains there at no further cost. We will assume a problem structure such that termination is inevitable. Thus the horizon is in effect finite, but its length is random and may be affected by the policy being used.

(b) *Discounted problems*. Here, $\alpha < 1$ and there need not be a termination state. However, we will see that a discounted problem can be readily converted to an SSP problem. This can be done by introducing an artificial termination state to which the system moves with probability $1 - \alpha$ at every stage, thus making termination inevitable. As a result, our algorithms and analysis for SSP problems can be easily adapted to discounted problems.

A Preview of Infinite Horizon Theory

There are several analytical and computational issues regarding our infinite horizon problems. Many of them revolve around the relation between the optimal cost-to-go function J^* of the infinite horizon problem and the optimal cost-to-go functions of the corresponding N-stage problems.

In particular, consider the SSP case and let $J_N(x)$ denote the optimal cost of the problem involving N stages, initial state x, cost per stage $g(x, u, w)$, and zero terminal cost. This cost is generated after N iterations of the DP algorithm

$$J_{k+1}(x) = \min_{u \in U(x)} \mathop{E}_{w} \Big\{ g(x, u, w) + J_k\big(f(x, u, w)\big) \Big\}, \qquad k = 0, 1, \ldots, \quad (4.1)$$

starting from the initial condition $J_0(x) = 0$ for all x.† The algorithm (4.1) is known as the *value iteration* algorithm (VI for short). Since the infinite horizon cost of a given policy is, by definition, the limit of the corresponding N-stage costs as $N \to \infty$, it is natural to speculate that:

† This is just the finite horizon DP algorithm of Chapter 1. However, we have reversed the time indexing to suit our purposes. Thus the index of the cost functions produced by the algorithm is incremented with each iteration, and not decremented as in the case of finite horizon.

(1) The optimal infinite horizon cost is the limit of the corresponding N-stage optimal costs as $N \to \infty$; i.e.,

$$J^*(x) = \lim_{N\to\infty} J_N(x) \tag{4.2}$$

for all states x.

(2) The following equation should hold for all states x,

$$J^*(x) = \min_{u\in U(x)} \mathop{E}_{w} \Big\{ g(x,u,w) + J^*\big(f(x,u,w)\big)\Big\}. \tag{4.3}$$

This is obtained by taking the limit as $N \to \infty$ in the VI algorithm (4.1) using Eq. (4.2). Equation (4.3) is really a system of equations (one equation per state x), which has as solution the costs-to-go of all the states. It can also be viewed as a *functional equation* for the optimal cost function J^*, and it is called *Bellman's equation.*

(3) If $\mu(x)$ attains the minimum in the right-hand side of the Bellman equation (4.3) for each x, then the policy $\{\mu, \mu, \ldots\}$ should be optimal. This type of policy is called *stationary*. Intuitively, optimal policies can be found within this class of policies, since the future optimization problem when starting at a given state looks the same regardless of the time when we start.

All three of the preceding results hold for SSP problems under our assumptions, as we will state later in Section 4.2 and prove in the appendix to this chapter. They also hold for discounted problems in suitably modified form that incorporates the discount factor. In fact the algorithms and analysis of this chapter are quite similar for SSP and discounted problems, to the point where we may discuss a particular method for one of the two problems with the understanding that its application to the other problem can be straightforwardly adapted.

Transition Probability Notation for Infinite Horizon Problems

Throughout this chapter we assume a finite-state discrete-time dynamic system, and we will use a special transition probability notation that is suitable for such a system. We generally denote states by the symbol i and successor states by the symbol j. We will assume that there are n states (in addition to the termination state for SSP problems). These states are denoted $1, \ldots, n$, and the termination state is denoted t. The control u is constrained to take values in a given finite constraint set $U(i)$, which may

depend on the current state i. The use of a control u at state i specifies the transition probability $p_{ij}(u)$ to the next state j, at a cost $g(i, u, j)$.†

Given an admissible policy $\pi = \{\mu_0, \mu_1, \ldots\}$ [one with $\mu_k(i) \in U(i)$ for all i and k] and an initial state i_0, the system becomes a Markov chain whose generated trajectory under π, denoted $\{i_0, i_1, \ldots\}$, has a well-defined probability distribution. The total expected cost associated with an initial state i is

$$J_\pi(i) = \lim_{N\to\infty} E\left\{ \sum_{k=0}^{N-1} \alpha^k g\big(i_k, \mu^k(i_k), i_{k+1}\big) \,\Big|\, i_0 = i,\, \pi \right\},$$

where α is either 1 (for SSP problems) or less than 1 for discounted problems. The expected value is taken with respect to the joint distribution of the states $i_1, i_2, \ldots$, conditioned on $i_0 = i$ and the use of π. The optimal cost from state i, i.e., the minimum of $J_\pi(i)$ over all policies π, is denoted by $J^*(i)$.

The cost function of a stationary policy $\pi = \{\mu, \mu, \ldots\}$ is denoted by $J_\mu(i)$. For brevity, we refer to π as the stationary policy μ. We say that μ is optimal if

$$J_\mu(i) = J^*(i) = \min_\pi J_\pi(i), \qquad \text{for all states } i.$$

As noted earlier, under our assumptions, we will show that there will always exist an optimal policy, which is stationary.

4.2 STOCHASTIC SHORTEST PATH PROBLEMS

In the SSP problem we assume that there is no discounting ($\alpha = 1$), and that *there is a special cost-free termination state* t. Once the system reaches that state, it remains there at no further cost, i.e.,

$$p_{tt}(u) = 1, \quad g(t, u, t) = 0, \qquad \text{for all } u \in U(t).$$

We denote by $1, \ldots, n$ the states other than the termination state t; see Fig. 4.2.1.

With this notation, the Bellman equation (4.3) and the VI algorithm (4.1) take the following form.

---

† To convert from the transition probability format to the system equation format used in the preceding chapters, we can simply use the system equation

$$x_{k+1} = w_k,$$

where w_k is the disturbance that takes values according to the transition probabilities $p_{x_k w_k}(u_k)$.

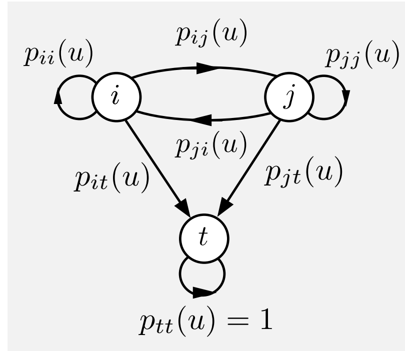

Figure 4.2.1 The transition graph of an SSP problem. There are n states, plus the termination state t, with transition probabilities $p_{ij}(u)$. The termination state is cost-free and absorbing.

Bellman Equation and Value Iteration for SSP Problems:

For all $i = 1, \ldots, n$, we have

$$J^*(i) = \min_{u \in U(i)} \left[p_{it}(u) g(i, u, t) + \sum_{j=1}^{n} p_{ij}(u) \big(g(i, u, j) + J^*(j) \big) \right]. \tag{4.4}$$

For all $i = 1, \ldots, n$, and any initial conditions $J_0(1), \ldots, J_0(n)$, the VI algorithm generates the sequence $\{J_k\}$ according to

$$J_{k+1}(i) = \min_{u \in U(i)} \left[p_{it}(u) g(i, u, t) + \sum_{j=1}^{n} p_{ij}(u) \big(g(i, u, j) + J_k(j) \big) \right]. \tag{4.5}$$

The right-hand bracketed expression in the Bellman equation (4.4) represents an expected value, which is similar to the expectation we have seen in earlier DP expressions. It is the sum of three terms:

(a) The contribution

$$p_{it}(u) g(i, u, t)$$

to the expected cost of the current stage of the terminating i-to-t transition.

(b) The contribution

$$\sum_{j=1}^{n} p_{ij}(u) g(i, u, j)$$

to the expected cost of the current stage of the nonterminating i-to-j transitions.

(c) The optimal expected cost-to-go

$$\sum_{j=1}^{n} p_{ij}(u) J^*(j)$$

starting from the next state j [if the next state is t, the corresponding optimal cost is $J^*(t)$, which is zero, so it does not appear in the sum].

Note that the deterministic shortest path problem of Section 1.3.1 is obtained as the special case of the SSP problem where for each state-control pair (i, u), the transition probability $p_{ij}(u)$ is equal to 1 for a unique state j that depends on (i, u). Moreover, any deterministic or stochastic finite-state, finite horizon problem with a termination state (cf. Section 1.3.3) can be converted to an SSP problem. In particular, the reader may verify that the finite-state N-step horizon problem of Chapter 1 can be obtained as a special case of an SSP problem by viewing as state the pair (x_k, k) and lumping all pairs (x_N, N) into a termination state t.

We are interested in problems where reaching the termination state t is inevitable. Thus, the essence of the problem is to reach t with minimum expected cost. Throughout this chapter, when discussing SSP problems, we will make the following assumption, which will be shown to guarantee eventual termination under *all* policies.†

---

† The main analytical and algorithmic results for SSP problems are valid under more general conditions, which involve the notion of a proper policy (see the end-of-chapter references). In particular, a stationary policy is called *proper* if starting from every state, it is guaranteed to eventually reach the destination. The policy is called *improper* if it is not proper.

It can be shown that Assumption 4.2.1 is equivalent to the seemingly weaker assumption that all stationary policies are proper. However, the subsequent four propositions can also be shown under the genuinely weaker assumption that there exists at least one proper policy, and furthermore, every improper policy is "bad" in the sense that it results in infinite expected cost from at least one initial state (see [BeT89], [BeT91], or [Ber12], Chapter 3). These assumptions, when specialized to deterministic shortest path problems, are similar to the assumptions of Section 1.3.1. They imply that there is at least one path to the destination from every starting state and that all cycles have positive cost. In the absence of these assumptions, the Bellman equation may have no solution or an infinite number of solutions (see [Ber18a], Section 3.1.1 for discussion of a simple example, which in addition to t, involves a single state 1 at which we can either stay at cost a or move to t at cost b; anomalies occur when $a = 0$ and when $a < 0$).

Assumption 4.2.1: (Termination is Inevitable Under All Policies) There exists an integer m such that regardless of the policy used and the initial state, there is positive probability that the termination state will be reached after no more that m stages; i.e., for all admissible policies π we have

$$\rho_\pi = \max_{i=1,\ldots,n} P\{x_m \neq t \mid x_0 = i, \pi\} < 1.$$

Let ρ be the maximum probability of not reaching t, over all starting states and policies:

$$\rho = \max_\pi \rho_\pi.$$

Note that ρ_π depends only on the first m components of the policy π. Furthermore, since the number of controls available at each state is finite, the number of distinct m-stage policies is also finite. It follows that there can be only a finite number of distinct values of ρ_π, so that

$$\rho < 1.$$

This implies that *the probability of not reaching t over a finite horizon diminishes to 0 as the horizon becomes longer*, regardless of the starting state and policy used.

To see this, note that for any π and any initial state i

$$\begin{aligned} P\{x_{2m} \neq t \mid x_0 = i, \pi\} &= P\{x_{2m} \neq t \mid x_m \neq t,\, x_0 = i, \pi\} \\ &\quad \cdot P\{x_m \neq t \mid x_0 = i, \pi\} \\ &\leq \rho^2. \end{aligned}$$

More generally, for each π, the probability of not reaching the termination state after km stages diminishes like ρ^k regardless of the initial state, i.e.,

$$P\{x_{km} \neq t \mid x_0 = i, \pi\} \leq \rho^k, \qquad i = 1, \ldots, n. \tag{4.6}$$

This fact implies that the limit defining the associated total cost vector J_π exists and is finite, and is central in the proof of the following results (given in the appendix to this chapter).

We now describe the main theoretical results for SSP problems; the proofs are given in the appendix to this chapter. Our first result is that the infinite horizon version of the DP algorithm, which is VI [cf. Eq. (4.1)], converges to the optimal cost function J^*. The optimal cost $J^*(t)$ starting from t is of course 0, so it is just neglected where appropriate in the subsequent analysis. Generally, J^* is obtained in the limit, after an infinite

number of iterations. However, there are important cases where convergence is obtained in finitely many iterations (see [Ber12], Chapter 3).

Proposition 4.2.1: (Convergence of VI) Given any initial conditions $J_0(1),\ldots,J_0(n)$, the sequence $\big\{J_k(i)\big\}$ generated by the VI algorithm

$$J_{k+1}(i)=\min_{u\in U(i)}\left[p_{it}(u)g(i,u,t)+\sum_{j=1}^{n}p_{ij}(u)\big(g(i,u,j)+J_k(j)\big)\right],$$

converges to the optimal cost $J^*(i)$ for each $i=1,\ldots,n$.

Our next result is that the limiting form of the DP equation, Bellman's equation, has J^* as its unique solution.

Proposition 4.2.2: (Bellman's Equation) The optimal cost function

$$J^*=\big(J^*(1),\ldots,J^*(n)\big)$$

satisfies for all $i=1,\ldots,n$, the equation

$$J^*(i)=\min_{u\in U(i)}\left[p_{it}(u)g(i,u,t)+\sum_{j=1}^{n}p_{ij}(u)\big(g(i,u,j)+J^*(j)\big)\right], \tag{4.7}$$

and is the unique solution of this equation.

Our next result expresses that by restricting attention to a single policy μ, we obtain a Bellman equation specific to μ, which has J_μ as its unique solution.

Proposition 4.2.3: (VI and Bellman's Equation for Policies) For any stationary policy μ, the corresponding cost function $J_\mu = \big(J_\mu(1),\ldots,J_\mu(n)\big)$ satisfies for all $i=1,\ldots,n$ the equation

$$J_\mu(i)=p_{it}\big(\mu(i)\big)g\big(i,\mu(i),t\big)+\sum_{j=1}^{n}p_{ij}\big(\mu(i)\big)\Big(g\big(i,\mu(i),j\big)+J_\mu(j)\Big),$$

and is the unique solution of this equation. Furthermore, given any initial conditions $J_0(1), \ldots, J_0(n)$, the sequence $\{J_k(i)\}$ generated by the VI algorithm that is specific to μ,

$$J_{k+1}(i) = p_{it}\big(\mu(i)\big)g\big(i, \mu(i), t\big) + \sum_{j=1}^{n} p_{ij}\big(\mu(i)\big)\Big(g\big(i, \mu(i), j\big) + J_k(j)\Big),$$

converges to the cost $J_\mu(i)$ for each i.

Our final result provides a necessary and sufficient condition for optimality of a stationary policy.

Proposition 4.2.4: (Optimality Condition) A stationary policy μ is optimal if and only if for every state i, $\mu(i)$ attains the minimum in the Bellman equation (4.7).

We provide an example illustrating Bellman's equation.

Example 4.2.1 (Maximum Expected Time to Termination)

The case where

$$g(i, u, j) = -1, \qquad \text{for all } i,\ u \in U(i), \text{ and } j,$$

corresponds to a problem where the objective is to terminate as late as possible on the average, while the opposite of the optimal cost, $-J^*(i)$, is the maximum expected time to termination starting from state i. Under our assumptions, the optimal costs $J^*(i)$ uniquely solve Bellman's equation, which has the form

$$J^*(i) = \min_{u \in U(i)} \left[-1 + \sum_{j=1}^{n} p_{ij}(u) J^*(j)\right], \qquad i = 1, \ldots, n.$$

In the special case of a single policy μ, where there is only one control at each state, $-J_\mu(i)$ represents the expected time to reach t starting from i. This is known as the mean first passage time from i to t, and is given as the unique solution of the corresponding Bellman equation

$$J_\mu(i) = -1 + \sum_{j=1}^{n} p_{ij}\big(\mu(i)\big)J_\mu(j)\big), \qquad i = 1, \ldots, n.$$

We will now provide an insightful mathematical result about SSP problems, which is proved in the appendix with the aid of the preceding

example. To this end let us introduce for any vector $J = \big(J(1), \ldots, J(n)\big)$, the notation

$$(TJ)(i) = \min_{u \in U(i)} \left[p_{it}(u) g(i,u,t) + \sum_{j=1}^{n} p_{ij}(u) \big(g(i,u,j) + J(j) \big) \right], \quad (4.8)$$

for all $i = 1, \ldots, n$, and

$$(T_\mu J)(i) = p_{it}\big(\mu(i)\big) g\big(i, \mu(i), t\big) + \sum_{j=1}^{n} p_{ij}(\mu(i)) \Big(g\big(i, \mu(i), j\big) + J(j) \Big), \quad (4.9)$$

for all policies μ and states $i = 1, \ldots, n$. Here T and T_μ are the DP operators that map the vector J into the vectors

$$TJ = \big((TJ)(1), \ldots, (TJ)(n)\big), \qquad T_\mu J = \big((T_\mu J)(1), \ldots, (T_\mu J)(n)\big),$$

respectively. Bellman's equations can be written in terms of these operators as the fixed point equations $J^* = TJ^*$ and $J_\mu = T_\mu J_\mu$.

The next proposition states that T and T_μ are contraction mappings, so the unique fixed point property of this mapping follows from general mathematical results about contraction mappings (see e.g., [Ber12], [Ber18a]). Moreover the contraction property provides a convergence rate estimate for VI, and is the basis for further analysis of exact and approximate methods for SSP problems (see the author's monograph [Ber18a] for a theoretical development of DP, which is based on fixed point theory and an abstract operator viewpoint).

Proposition 4.2.5: (Contraction Property of the DP Operator) The DP operators T and T_μ of Eqs. (4.8) and (4.9) are contraction mappings with respect to some weighted norm

$$\|J\| = \max_{i=1,\ldots,n} \frac{\big|J(i)\big|}{v(i)},$$

defined by some vector $v = \big(v(1), \ldots, v(n)\big)$ with positive components. In other words, there exist positive scalar $\rho < 1$ and $\rho_\mu < 1$ such that for any two n-dimensional vectors J and J', we have

$$\|TJ - TJ'\| \leq \rho \, \|J - J'\|, \qquad \|T_\mu J - T_\mu J'\| \leq \rho_\mu \, \|J - J'\|.$$

Note that the weight vector v and the corresponding weighted norm may be different for T and for T_μ. The proof of the proposition, given in

the appendix, shows that the weights $v(i)$ and the modulus of contraction ρ are related to the maximum expected number of steps $-m^*(i)$ to reach t from i (cf. Example 4.2.1). In particular, we have

$$v(i) = -m^*(i), \qquad \rho = \max_{i=1,\ldots,n} \frac{v(i)-1}{v(i)}.$$

Among others, the preceding contraction property provides a convergence rate estimate for VI, namely that the generated sequence $\{J_k\}$ satisfies

$$\|J_k - J^*\| \leq \rho^k \|J_0 - J^*\|.$$

This follows from the fact that J_k and J^* can be viewed as the results of the k-fold application of T to the vectors J_0 and J^*, respectively.

Bellman Equation and Value Iteration for Q-Factors

The results just given have counterparts involving Q-factors. The optimal Q-factors are defined for all $i = 1, \ldots, n$, and $u \in U(i)$ by

$$Q^*(i,u) = p_{it}(u)g(i,u,t) + \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + J^*(j)\big).$$

As in the finite horizon case, $Q^*(i,u)$ can be interpreted as the cost of starting at i, using u for the first stage, and using an optimal policy afterwards. Once Q^* is computed by some method, an optimal policy μ^* can be obtained from the minimization

$$\mu^*(i) \in \arg\min_{u \in U(i)} Q^*(i,u), \qquad i = 1, \ldots, n.$$

Similarly, if approximately optimal Q-factors $\tilde{Q}(i,u)$ are obtained by some method (model-based or model-free), a suboptimal policy $\tilde{\mu}$ can be obtained from the minimization

$$\tilde{\mu}(i) \in \arg\min_{u \in U(i)} \tilde{Q}(i,u), \qquad i = 1, \ldots, n.$$

Our basic results relating Bellman's equation and the VI algorithm are stated as follows.

Bellman Equation and Value Iteration for Q-Factors and SSP Problems:

For all $i = 1, \ldots, n$, and $u \in U(i)$ we have

$$Q^*(i,u) = p_{it}(u)g(i,u,t) + \sum_{j=1}^{n} p_{ij}(u) \left(g(i,u,j) + \min_{v \in U(j)} Q^*(j,v) \right).$$

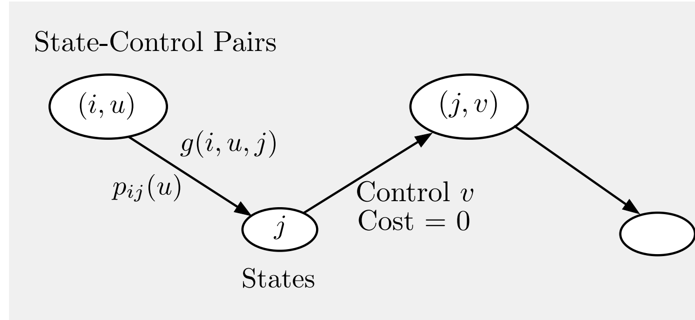

Figure 4.2.2 States, transition probabilities, and stage costs corresponding to a modified SSP problem, which yields the optimal Q-factors as well as the optimal costs. The states of this problem are the pairs (i, u), $u \in U(i)$, the original problem states $i = 1, \ldots, n$, and the termination state t. A control $v \in U(j)$ is available only at the original system states j, leading to the pair (j, v) at cost 0. The transition from a pair (i, u) leads to j with probability $p_{ij}(u)$ and cost 0. The Bellman equation for this modified problem is

$$Q^*(i, u) = p_{it}(u)g(i, u, t) + \sum_{j=1}^{n} p_{ij}(u) \left(g(i, u, j) + \min_{v \in U(j)} Q^*(j, v) \right),$$

for the states (i, u), $u \in U(i)$, and

$$J^*(j) = \min_{v \in U(j)} Q^*(j, v),$$

for the states $j = 1, \ldots, n$. Note that a policy μ for this problem leads from a state j to the state $\big(j, \mu(j)\big)$, so in any system trajectory, only pairs of the form $\big(j, \mu(j)\big)$ are visited after the first transition.

For all $i = 1, \ldots, n$, and $u \in U(i)$, and any initial conditions $Q_0(i, u)$, the VI algorithm generates the sequence $\{Q_k\}$ according to

$$Q_{k+1}(i, u) = p_{it}(u)g(i, u, t) + \sum_{j=1}^{n} p_{ij}(u) \left(g(i, u, j) + \min_{v \in U(j)} Q_k(j, v) \right).$$

Actually, the optimal Q-factors $Q^*(i, u)$ can be viewed as optimal state costs associated with a modified SSP problem, which involves a new state for each pair (i, u) with transition probabilities $p_{ij}(u)$ to the states $j = 1, \ldots, n, t$; see Fig. 4.2.2. Then the preceding Bellman equation for the optimal Q-factors, together with the Bellman equation (4.7) for the optimal costs $J^*(j)$, can be viewed as the Bellman equation for the modified SSP

problem.

Temporal Differences and Cost Shaping

Bellman's equation can be written in an alternative form, which involves the differential

$$\hat{J} = J^* - V,$$

where $V = \big(V(1), \ldots, V(n)\big)$ is *any* n-dimensional vector and $V(t) = 0$. In particular, by subtracting $V(i)$ from both sides of the Bellman equation (4.7), and adding and subtracting $V(j)$ within the right-hand side summation, we obtain

$$\hat{J}(i) = p_{it}(u)\hat{g}(i,u,t) + \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(\hat{g}(i,u,j) + \hat{J}(j)\big), \tag{4.10}$$

for all $i = 1, \ldots, n$, where

$$\hat{g}(i,u,j) = \begin{cases} g(i,u,j) + V(j) - V(i) & \text{if } i, j = 1, \ldots, n, \\ g(i,u,t) - V(i) & \text{if } i = 1, \ldots, n,\ j = t. \end{cases} \tag{4.11}$$

We refer to Eq. (4.10) as the *variational form of Bellman's equation*, and to the modified cost per stage $\hat{g}$ as the *temporal difference* corresponding to V. Temporal differences play a significant role in several algorithmic RL contexts; see Section 4.9, and the approximate DP/RL books referenced earlier.

Note that Eq. (4.10) is the Bellman equation for a cost-modified problem, where the cost per stage g has been replaced by the temporal difference $\hat{g}$. Thus by applying Prop. 4.2.2 we have that $\hat{J} = J^* - V$ is the unique solution of this equation, so that J^* can be obtained by solving either the original or the cost-modified version of the problem. Moreover, a policy μ, has cost function $\hat{J}_\mu = J_\mu - V$ in the cost-modified problem. It follows that the original and the cost-modified SSP problems are essentially equivalent, and the choice of V does not matter when exact DP methods are used to solve them. However, when approximate methods are used, different results may be obtained, which can be more favorable with an appropriate choice of V.

In particular, we have the option to choose V and an approximation architecture methodology that matches the differential $\hat{J} = J^* - V$ better than it matches J^*. For example, we may obtain V with some problem approximation scheme as a rough estimate of J^*, and then use a different approximation in value space scheme, based on different principles, for the corresponding cost-modified problem. We refer to this as *cost shaping* (the name "reward shaping" in used in the RL literature, for problems involving reward maximization). While cost shaping does not change the optimal policies of the original DP problem, it may change significantly the suboptimal policies produced by approximate DP methods, such as the ones that we will discuss in this chapter and the next.

4.3 DISCOUNTED PROBLEMS

We now consider the discounted problem, where there is a discount factor $\alpha < 1$. Using our transition probability notation, the Bellman equation and the VI algorithm take the following form.

Bellman Equation and Value Iteration for Discounted Problems:

For all $i = 1, \ldots, n$, we have

$$J^*(i) = \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha J^*(j)\big).$$

For all $i = 1, \ldots, n$, and any initial conditions $J_0(1), \ldots, J_0(n)$, the VI algorithm generates the sequence $\{J_k\}$ according to

$$J_{k+1}(i) = \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha J_k(j)\big).$$

We will now show that the discounted problem can be converted to an SSP problem for which the analysis of the preceding section applies. To see this, let $i = 1, \ldots, n$ be the states, and consider an associated SSP problem involving the states $1, \ldots, n$ plus an artificial termination state t, with state transitions and costs obtained as follows: From a state $i \neq t$, when control u is applied, the next state is j with probability $\alpha p_{ij}(u)$ at a cost $g(i,u,j)$, and t with probability $1 - \alpha$ at zero cost; see Fig. 4.3.1. Note that Assumption 4.2.1 of the preceding section is satisfied for this SSP problem, since t is reached with probability $1 - \alpha > 0$ from every state in a single step.

Suppose now that we use the same policy in the discounted problem and in the associated SSP problem. Then, as long as termination has not occurred, the state evolution in the two problems is governed by the same transition probabilities. Furthermore, the expected cost of the kth stage of the associated shortest path problem is the expected value of $g\big(i_k, \mu^k(i_k), i_{k+1}\big)$ multiplied by the probability that state t has not yet been reached, which is α^k. This is also the expected cost of the kth stage for the discounted problem. Thus the cost of any policy starting from a given state, is the same for the original discounted problem and for the associated SSP problem.

It follows that we can apply Props. 4.2.1-4.2.5 of the preceding section to the associated SSP problem and obtain corresponding results for

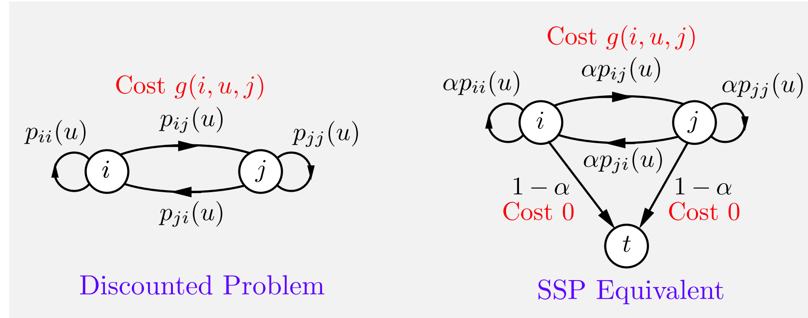

Figure 4.3.1 Transition probabilities for an α-discounted problem and its associated SSP problem. In the latter problem, the probability that the state is not t after k stages is α^k. The transition costs at the kth stage are $g(i,u,j)$ for both problems, but they must be multiplied by α^k because of discounting (in the discounted case) or because it is incurred with probability α^k when termination has not yet been reached (in the SSP case).

the discounted problem, which properly incorporate the discount factor in accordance with the SSP-to-discounted equivalence just established.

Proposition 4.3.1: (Convergence of VI) Given any initial conditions $J_0(1), \ldots, J_0(n)$, the sequence $\big\{J_k(i)\big\}$ generated by the VI algorithm

$$J_{k+1}(i) = \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha J_k(j)\big), \qquad i = 1, \ldots, n,$$

converges to the optimal cost $J^*(i)$ for each i.

Proposition 4.3.2: (Bellman's Equation) The optimal cost function

$$J^* = \big(J^*(1), \ldots, J^*(n)\big)$$

satisfies for all $i = 1, \ldots, n$, the equation

$$J^*(i) = \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha J^*(j)\big), \tag{4.12}$$

and is the unique solution of this equation.

Proposition 4.3.3: (VI and Bellman's Equation for Policies) For any stationary policy μ, the corresponding cost function $J_\mu = \big(J_\mu(1),\ldots,J_\mu(n)\big)$ is the unique solution of the equation

$$J_\mu(i) = \sum_{j=1}^{n} p_{ij}\big(\mu(i)\big)\Big(g\big(i,\mu(i),j\big) + \alpha J_\mu(j)\Big), \qquad i = 1,\ldots,n.$$

Furthermore, given any initial conditions $J_0(1),\ldots,J_0(n)$, the sequence $\big\{J_k(i)\big\}$ generated by the VI algorithm

$$J_{k+1}(i) = \sum_{j=1}^{n} p_{ij}\big(\mu(i)\big)\Big(g\big(i,\mu(i),j\big) + \alpha J_k(j)\Big), \qquad i = 1,\ldots,n,$$

converges to the cost $J_\mu(i)$ for each i.

Proposition 4.3.4: (Optimality Condition) A stationary policy μ is optimal if and only if for every state i, $\mu(i)$ attains the minimum in the Bellman equation (4.12).

Bellman's equation (4.12) has a familiar DP interpretation. At state i, the optimal cost $J^*(i)$ is the minimum over all controls of the sum of the expected current stage cost and the expected optimal cost of all future stages. The former cost is $g(i,u,j)$. The latter cost is $J^*(j)$, but since this cost starts accumulating after one stage, it is discounted by multiplication with α.

Similar to Prop. 4.2.5, there is a contraction mapping result and convergence rate estimate for value iteration. To this end we introduce the mappings

$$(TJ)(i) = \min_{u\in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha J(j)\big), \qquad i = 1,\ldots,n, \tag{4.13}$$

and

$$(T_\mu J)(i) = \sum_{j=1}^{n} p_{ij}\big(\mu(i)\big)\Big(g\big(i,\mu(i),j\big) + \alpha J(j)\Big), \qquad i = 1,\ldots,n, \tag{4.14}$$

in analogy with their SSP counterparts of Eqs. (4.8) and (4.9). Similar to the SSP case, Bellman's equations can be written in terms of these

operators as the fixed point equations $J^* = TJ^*$ $J_\mu = T_\mu J_\mu$. The following contraction result is useful for the analysis of exact and approximate methods for discounted problems.

Proposition 4.3.5: (Contraction Property of the DP Operator) The DP operators T and T_μ of Eqs. (4.13) and (4.14) are contraction mappings of modulus α with respect to the maximum norm

$$\|J\| = \max_{i=1,\ldots,n} \big|J(i)\big|. \tag{4.15}$$

In particular, for any two n-dimensional vectors J and J', we have

$$\|TJ - TJ'\| \leq \alpha\|J - J'\|, \qquad \|T_\mu J - T_\mu J'\| \leq \alpha\|J - J'\|.$$

Let us also mention that the cost shaping idea discussed for SSP problems, extends readily to discounted problems. In particular, the variational form of Bellman's equation takes the form

$$\hat{J}(i) = \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(\hat{g}(i,u,j) + \alpha\hat{J}(j)\big), \qquad i = 1,\ldots,n,$$

for any given vector V, where

$$\hat{g}(i,u,j) = g(i,u,j) + \alpha V(j) - V(i), \qquad i = 1,\ldots,n,$$

is the temporal difference corresponding to V; cf. Eqs. (4.10) and (4.11).

Example 4.3.1 (Asset Selling)

Consider of problem of selling an asset over an infinite number of periods. At each period an offer becomes available. We assume that offers at different periods are independent and that they can take n values $v_1,\ldots,v_n$ with corresponding probabilities according to given probability $p(1),\ldots,p(n)$. Here, if accepted, the amount i_k offered in period k, will be invested at a rate of interest r. By depreciating the sale amount to period 0 dollars, we view $(1+r)^{-k}i_k$ as the reward for selling the asset in period k at a price i_k, where $r > 0$ is the rate of interest. Then we have a discounted reward problem with discount factor $\alpha = 1/(1+r)$. The analysis of the present section is applicable, and the optimal value function J^* is the unique solution of Bellman's equation

$$J^*(i) = \max\left[i,\ \frac{1}{1+r}\sum_{j=1}^{n} p_j J^*(j)\right].$$

Thus the optimal reward function is characterized by the critical number

$$c = \frac{1}{1+r}\sum_{j=1}^{n} p_j J^*(j).$$

An optimal policy is obtained by minimizing over the two controls. It is to sell if and only if the current offer i is greater than c. The critical number c can be obtained by a simple form of VI (see [Ber17], Section 3.4.

A far more difficult version of the problem is one where the offers are correlated, so the offer at each stage may be viewed as an observation that provides information about future offers. A related difficult version of the problem is one where the probability distribution $p = \big(p(1),\ldots,p(n)\big)$ of the offers is unknown, and is estimated as new offers are revealed. In both cases the problem can be formulated as a partial state information problem involving a belief state: the estimate of the distribution p given the past offers (suitable conditions are of course needed to ensure that the estimate of p can be in principle computed exactly or can be approximated as a practical matter). Some of the finite horizon approximation methods of Chapters 2 and 3 can be adapted to solve such a problem. However, an exact solution is practically impossible, since this would involve DP calculations over an infinite dimensional space of belief states.

Bellman Equation and Value Iteration for Q-Factors

As in the SSP case, the results just given have counterparts involving the optimal Q-factors, defined by

$$Q^*(i,u) = \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha J^*(j)\big), \qquad i = 1,\ldots,n,\ u \in U(i).$$

They can be obtained from the corresponding SSP results, by viewing the discounted problem as a special case of the SSP problem. Once Q^* or an approximation $\tilde{Q}$ is computed by some method (model-based or model-free), an optimal policy μ^* or approximately optimal policy $\tilde{\mu}$ can be obtained from the minimization

$$\mu^*(i) \in \arg\min_{u\in U(i)} Q^*(i,u), \qquad i = 1,\ldots,n,$$

or the approximate version

$$\tilde{\mu}(i) \in \arg\min_{u\in U(i)} \tilde{Q}(i,u), \qquad i = 1,\ldots,n.$$

Our basic results relating Bellman's equation and the VI algorithm are stated as follows.

Bellman Equation and Value Iteration for Q-Factors and Discounted Problems:

For all $i = 1, \ldots, n$, and $u \in U(i)$ we have

$$Q^*(i,u) = \sum_{j=1}^{n} p_{ij}(u) \left(g(i,u,j) + \alpha \min_{v \in U(j)} Q^*(j,v) \right). \tag{4.16}$$

For all $i = 1, \ldots, n$, and $u \in U(i)$, and any initial conditions $Q_0(i,u)$, the VI algorithm generates the sequence $\{Q_k\}$ according to

$$Q_{k+1}(i,u) = \sum_{j=1}^{n} p_{ij}(u) \left(g(i,u,j) + \alpha \min_{v \in U(j)} Q_k(j,v) \right). \tag{4.17}$$

The VI algorithn (4.17) forms the basis for various Q-learning methods to be discussed later.

4.4 EXACT AND APPROXIMATE VALUE ITERATION

We have already encountered the VI algorithm for SSP problems,

$$J_{k+1}(i) = \min_{u \in U(i)} \left[p_{it}(u) g(i,u,t) + \sum_{j=1}^{n} p_{ij}(u) \big(g(i,u,j) + J_k(j) \big) \right], \tag{4.18}$$

and its discounted version

$$J_{k+1}(i) = \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u) \big(g(i,u,j) + \alpha J_k(j) \big). \tag{4.19}$$

It is one of the principal methods for calculating the optimal cost function J^*.

Unfortunately, when the number of states is large, the iterations (4.18) and (4.19) may be prohibitively time consuming. This motivates an approximate version of VI, which is patterned after the least squares regression/fitted VI scheme of Section 3.3. We start with some initial approximation to J^*, call it $\tilde{J}_0$. Then we generate a sequence $\{\tilde{J}_k\}$ where $\tilde{J}_{k+1}$ is equal to the exact value iterate $T\tilde{J}_k$ plus some error [we are using here the shorthand notation for the DP operator T given in Eqs. (4.8) and (4.13)]. Assuming that values $(T\tilde{J}_k)(i)$ may be generated for sample states i, we may obtain $\tilde{J}_{k+1}$ by some form of least squares regression. We will now discuss how the error $(\tilde{J}_k - J^*)$ is affected by this type of approximation process.

Error Bounds and Pathologies of Approximate Value Iteration

We will focus on approximate VI for discounted problems. The analysis for SSP problems is qualitatively similar. We first consider estimates of the *cost function error*

$$\max_{i=1,\ldots,n} \big|\tilde J_k(i) - J^*(i)\big|, \tag{4.20}$$

and the *policy error*

$$\max_{i=1,\ldots,n} \big|J_{\tilde\mu^k}(i) - J^*(i)\big|, \tag{4.21}$$

where the policy $\tilde\mu^k$ is obtained from the minimization

$$\tilde\mu^k(i) \in \arg\min_{u\in U(i)} \sum_{j=1}^n p_{ij}(u)\big(g(i,u,j) + \alpha \tilde J_k(j)\big).$$

It turns out that such estimates are possible, but under assumptions whose validity may be hard to guarantee. In particular, it is natural to assume that the error in generating the value iterates $(T\tilde J_k)(i)$ is within some $\delta > 0$ for every state i and iteration k, i.e., that

$$\max_{i=1,\ldots,n} \left| \tilde J_{k+1}(i) - \min_{u\in U(i)} \sum_{j=1}^n p_{ij}(u)\big(g(i,u,j) + \alpha \tilde J_k(j)\big) \right| \le \delta. \tag{4.22}$$

It is then possible to show that asymptotically, as $k \to \infty$, the cost error (4.20) becomes less or equal to $\delta/(1-\alpha)$, while the policy error (4.21) becomes less or equal to $2\delta/(1-\alpha)^2$.

Such error bounds are given in Section 6.5.3 of the book [BeT96] (see also Prop. 2.5.3 of [Ber12]), but it is important to note that the condition (4.22) may not be satisfied by the natural least squares regression/fitted VI scheme of Section 3.3. This is illustrated by the following simple example from [TsV96] (see also [BeT96], Section 6.5.3), which shows that the errors from successive approximate value iterations can accumulate to the point where the condition (4.22) cannot be maintained, and the approximate value iterates $\tilde J_k$ can grow unbounded.

Example 4.4.1 (Error Amplification in Approximate Value Iteration)

Consider a two-state discounted problem with states 1 and 2, and a single policy. The transitions are deterministic: from state 1 to state 2, and from state 2 to state 2. The transitions are also cost-free; see Fig. 4.4.1. Thus the Bellman equation is

$$J(1) = \alpha J(2), \qquad J(2) = \alpha J(2),$$

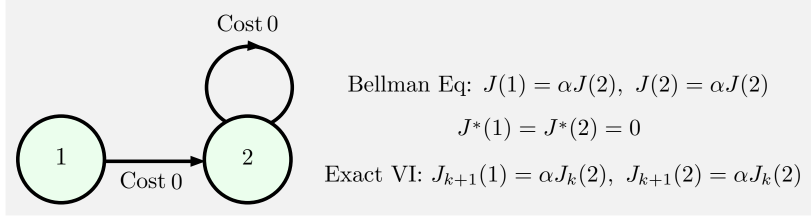

Figure 4.4.1 Illustration of the discounted problem of Example 4.4.1. There are two states, 1 and 2, and a single policy. The transitions are deterministic: from state 1 to state 2, and from state 2 to state 2. These transitions are also cost-free.

and its unique solution is $J^*(1) = J^*(2) = 0$. Moreover, exact VI has the form

$$J_{k+1}(1) = \alpha J_k(2), \qquad J_{k+1}(2) = \alpha J_k(2).$$

We consider a VI approach that approximates cost functions within the one-dimensional subspace of linear functions $S = \big\{(r, 2r) \mid r \in \Re\big\}$; this is a favorable choice since the optimal cost function $J^* = (0, 0)$ belongs to S. We use a weighted least squares regression scheme. In particular, given $\tilde J_k = (r_k, 2r_k)$, we find $\tilde J_{k+1} = (r_{k+1}, 2r_{k+1})$ as follows; see Fig. 4.4.2:

(a) We compute the exact VI iterate from $\tilde J_k$:

$$T\tilde J_k = \big(\alpha \tilde J_k(2), \alpha \tilde J_k(2)\big) = (2\alpha r_k, 2\alpha r_k).$$

(b) For some weights $\xi_1, \xi_2 > 0$, we obtain the scalar r_{k+1} as

$$r_{k+1} \in \arg\min_r \Big[\xi_1\big(r - (T\tilde J_k)(1)\big)^2 + \xi_2\big(2r - (T\tilde J_k)(2)\big)^2\Big],$$

or

$$r_{k+1} \in \arg\min_r \big[\xi_1(r - 2\alpha r_k)^2 + \xi_2(2r - 2\alpha r_k)^2\big].$$

To perform the preceding minimization, we write the corresponding optimality condition (set to zero the derivative with respect to r), and obtain after some calculation

$$r_{k+1} = \alpha\zeta r_k \quad \text{where} \quad \zeta = \frac{2(\xi_1 + 2\xi_2)}{\xi_1 + 4\xi_2} > 1. \tag{4.23}$$

Thus if ξ_1 and ξ_2 are chosen so that $\alpha > 1/\zeta$, the sequence $\{r_k\}$ diverges and so does $\{\tilde J_k\}$. In particular, for the natural choice $\xi_1 = \xi_2 = 1$, we have $\zeta = 6/5$, so the approximate VI scheme diverges for α in the range $(5/6, 1)$; see Fig. 4.4.2.

The difficulty here is that the approximate VI mapping that generates $\tilde J_{k+1}$ by a weighted least squares-based approximation of $T\tilde J_k$ is not a contraction (even though T itself is a contraction). At the same time there is

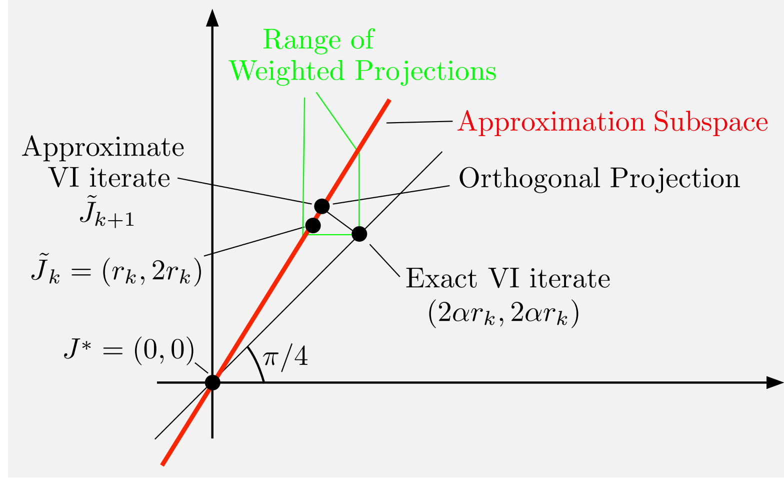

Figure 4.4.2 Illustration of Example 4.4.1. Iterates of approximate VI lie on the line $\big\{(r, 2r) \mid r \in \Re\big\}$. Given an iterate $\tilde{J}_k = (r_k, 2r_k)$, the next exact VI iterate is

$$\big(\alpha \tilde{J}_k(2), \alpha \tilde{J}_k(2)\big) = (2\alpha r_k, 2\alpha r_k).$$

The approximation of this iterate on the line $\{(r, 2r) \mid r \in \Re\}$ by least squares regression can be viewed as weighted projection onto the line, and depends on the weights (ξ_1, ξ_2). The range of weighted projections as the weights vary is shown in the figure. For the natural choice $\xi_1 = \xi_2 = 1$ and α sufficiently close to 1, the new approximate VI iterate $\tilde{J}_{k+1}$ is further away from $J^* = (0,0)$ than $\tilde{J}_k$. The difficulty here is that the mapping that consists of a VI followed by weighted projection onto the line $\{(r, 2r) \mid r \in \Re\}$ need not be a contraction.

no δ such that the condition (4.22) is satisfied for all k, because of error amplification in each approximate VI.

The preceding example indicates that the choice of the least squares weights is important in determining the success of least squares-based approximate VI schemes. Generally, in regression-based parametric architecture training schemes of the type discussed is Section 3.1.2, the weights are related to the way samples are collected: the weight ξ_i for state i is the proportion of the number of samples in the least squares summation that correspond to state i. Thus $\xi_1 = \xi_2 = 1$ in the preceding example means that we use an equal number of samples for each of the two states 1 and 2.

Now let us consider an approximation architecture $\tilde{J}(i, \cdot)$ and a sampling process for approximating the value iterates. In particular, let

$$\tilde{J}_k(i) = \tilde{J}(i, r_k), \qquad i = 1, \ldots, n,$$

where r_k is the parameter vector corresponding to iteration k. Then the

parameter r_{k+1} used to represent the next value iterate as

$$\tilde{J}_{k+1}(i) = \tilde{J}(i, r_{k+1}), \qquad i = 1, \ldots, n,$$

is obtained by the minimization

$$r_{k+1} \in \arg\min_r \sum_{s=1}^{q} \big(\tilde{J}(i^s, r) - \beta^s\big)^2, \tag{4.24}$$

where (i^s, β^s), $s = 1, \ldots, q$, is a training set with each β^s being the value iterate at the state i^s:

$$\beta^s = (T\tilde{J}_k)(i^s).$$

The critical question now is how to select the sample states i^s, $s = 1, \ldots, q$, to guarantee that the iterates r_k remain bounded, so that a condition of the form (4.22) is satisfied and the instability illustrated with Example 4.4.1 is avoided. It turns out that there is no known general method to guarantee this in infinite horizon problems. However, some practical methods have been developed. One such method is to weigh each state according to its "long-term importance," i.e., proportionally to the number of its occurrences over a long trajectory under a "good" heuristic policy.† To implement this, we may run the system with the heuristic policy starting from a number of representative states, wait for some time for the system to approach steady-state, and record the generated states i^s, $s = 1, \ldots, q$, to be used in the regression scheme (4.24). There is no theoretical guarantee for the stability of this scheme in the absence of additional conditions: it has been used with success in several reported case studies, although its rationale has only a tenuous basis in analysis. For a discussion of this issue, we refer to [Ber12], Section 6.3, and other end-of-chapter references.

4.5 POLICY ITERATION

The major alternative to value iteration is *policy iteration* (PI for short). This algorithm starts with a stationary policy μ^0, and generates iteratively a sequence of new policies $\mu^1, \mu^2, \ldots$. The algorithm has solid convergence guarantees when implemented in its exact form, as we will show shortly. When implemented in approximate form, as it is necessary when

† In the preceding Example 4.4.1, weighing the two states according to their "long-term importance" would choose ξ_2 to be much larger than ξ_1, since state 2 is "much more important," in the sense that it occurs almost exclusively in system trajectories. Indeed, from Eq. (4.23) it can be seen that when the ratio ξ_1/ξ_2 is close enough to 0, the scalar ζ is close enough to 1, making the scalar $\alpha\zeta$ strictly less than 1, and guaranteeing convergence of $\tilde{J}_k$ to J^*.

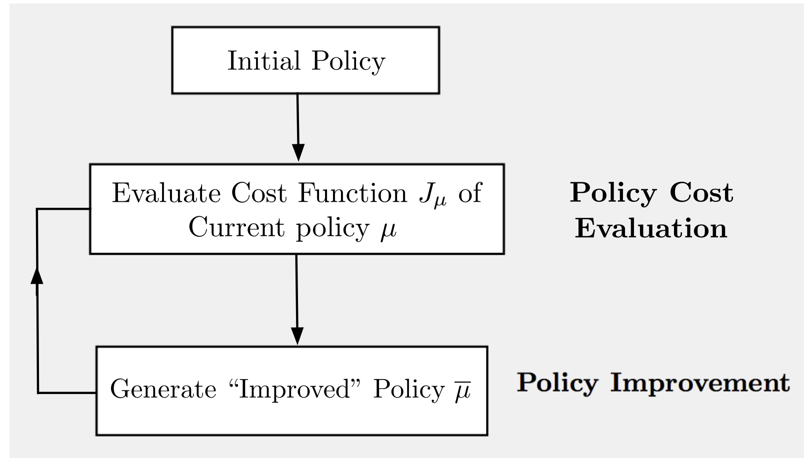

Figure 4.5.1 Illustration of exact PI. Each iteration consists of a policy evaluation using the current policy μ, followed by generation of an improved policy $\tilde{\mu}$.

the number of states is large, its performance guarantees are somewhat more favorable than those of the approximate VI of the preceding section.

The closest analog of PI that we have encountered so far is the rollout algorithm of Chapter 2. There we have started with some policy and produced an improved policy though a process of cost function evaluation and one-step or multistep minimization. This idea is extended in the context of PI, which consists of multiple successive policy evaluations and policy improvements.

4.5.1 Exact Policy Iteration

Consider first the SSP problem. Here, each policy iteration consists of two phases: *policy evaluation* and *policy improvement*; see Fig. 4.5.1.

Exact Policy Iteration: SSP Problems

Given the typical policy μ^k:

Policy evaluation computes $J_{\mu^k}(i)$, $i = 1, \ldots, n$, as the solution of the (linear) system of Bellman equations

$$J_{\mu^k}(i) = \sum_{j=1}^{n} p_{ij}\big(\mu^k(i)\big)\Big(g\big(i, \mu^k(i), j\big) + J_{\mu^k}(j)\Big), \quad i = 1, \ldots, n,$$

(cf. Prop. 4.2.3).

Policy improvement then computes a new policy μ^{k+1} as

$$\mu^{k+1}(i) \in \arg\min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + J_{\mu^k}(j)\big), \quad i = 1, \ldots, n.$$

The process is repeated with μ^{k+1} used in place of μ^k, unless we have

$$J_{\mu^{k+1}}(i) = J_{\mu^k}(i)$$

for all i, in which case the algorithm terminates with the policy μ^k.

The counterpart for discounted problems is as follows.

Exact Policy Iteration: Discounted Problems

Given the typical policy μ^k:

Policy evaluation computes $J_{\mu^k}(i)$, $i = 1, \ldots, n$, as the solution of the (linear) system of Bellman equations

$$J_{\mu^k}(i) = \sum_{j=1}^{n} p_{ij}\big(\mu^k(i)\big)\Big(g\big(i, \mu^k(i), j\big) + \alpha J_{\mu^k}(j)\Big), \quad i = 1, \ldots, n, \tag{4.25}$$

(cf. Prop. 4.2.3).

Policy improvement then computes a new policy μ^{k+1} as

$$\mu^{k+1}(i) \in \arg\min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha J_{\mu^k}(j)\big), \quad i = 1, \ldots, n. \tag{4.26}$$

The process is repeated with μ^{k+1} used in place of μ^k, unless we have $J_{\mu^{k+1}}(i) = J_{\mu^k}(i)$ for all i, in which case the algorithm terminates with the policy μ^k.

The following proposition, shown in the appendix, establishes the validity of PI, including finite termination with an optimal policy.

Proposition 4.5.1: (Convergence of Exact PI) For both the SSP and the discounted problems, the exact PI algorithm generates an improving sequence of policies, i.e.,

$$J_{\mu^{k+1}}(i) \leq J_{\mu^k}(i), \qquad \text{for all } i \text{ and } k, \tag{4.27}$$

and terminates with an optimal policy.

The proof of the policy improvement property (4.27) is quite intuitive and is worth summarizing for the discounted problem. Let μ be a policy and $\tilde{\mu}$ be the policy obtained from μ via a policy iteration. We want to show that $J_{\tilde{\mu}} \leq J_\mu$. To this end, let us *denote by J_N the cost function of a policy that applies $\tilde{\mu}$ for the first N stages and applies μ at every subsequent stage*. We have the Bellman equation

$$J_\mu(i) = \sum_{j=1}^n p_{ij}\big(\mu(i)\big)\Big(g\big(i,\mu(i),j\big) + \alpha J_\mu(j)\Big),$$

which together with the policy improvement equation (4.26) imply that

$$J_1(i) = \sum_{j=1}^n p_{ij}\big(\tilde{\mu}(i)\big)\Big(g\big(i,\tilde{\mu}(i),j\big) + \alpha J_\mu(j)\Big) \leq J_\mu(i). \tag{4.28}$$

From the definition of J_2 and J_1 we have

$$J_2(i) = \sum_{j=1}^n p_{ij}\big(\tilde{\mu}(i)\big)\Big(g\big(i,\tilde{\mu}(i),j\big) + \alpha J_1(j)\Big), \tag{4.29}$$

so the preceding two relations imply that

$$J_2(i) \leq J_1(i) \leq J_\mu(i), \qquad \text{for all } i. \tag{4.30}$$

Continuing similarly, we obtain

$$J_{N+1}(i) \leq J_N(i) \leq J_\mu(i), \qquad \text{for all } i \text{ and } N. \tag{4.31}$$

Since $J_N \to J_{\tilde{\mu}}$ (cf. Prop. 4.3.2), it follows that $J_{\tilde{\mu}} \leq J_\mu$.

In practice, a lot of cost improvement is often obtained with the PI algorithm after the first few policies are generated. This may happen even if the number of iterations needed for termination is quite large. The following is an example where termination occurs after just two iterations.

Example 4.5.1 (Treasure Hunting)

A treasure hunter has obtained a lease to search a site that contains n treasures, and wants to find a searching policy that maximizes his expected gain over an infinite number of days. At each day, knowing the current number of

treasures not yet found, he may decide to continue searching for more treasures at a cost c per day, or to permanently stop searching. If he searches on a day when there are i treasures on the site, he finds $m \in [0, i]$ treasures with given probability $p(m \mid i)$, where we assume that $p(0 \mid i) < 1$ for all $i \geq 1$, and that the expected number of treasures found,

$$r(i) = \sum_{m=0}^{i} m\, p(m \mid i),$$

is monotonically increasing with i. Each found treasure is worth 1 unit.

We formulate the problem as an SSP problem, with state equal to the number of treasures not yet found. The termination state is state 0, where the hunter stops searching. When the hunter decides to search at a state $i \geq 1$, the state moves to $i - m$ with probability $p(m \mid i)$. Here the inevitable termination Assumption 4.2.1 is satisfied, in view of our condition $p(0 \mid i) < 1$ for all i. Bellman's equation is

$$J^*(i) = \max\left[0, r(i) - c + \sum_{m=0}^{i} p(m \mid i) J^*(i-m)\right], \qquad i = 1, \ldots, n,$$

with $J^*(0) = 0$.

Let us apply PI starting with the policy μ^0 that never searches. This policy has value function

$$J_{\mu^0}(i) = 0, \qquad \text{for all } i.$$

The policy μ^1 subsequently produced by PI is the one that searches at a state i if and only if $r(i) > c$, and has value function satisfying the Bellman equation

$$J_{\mu^1}(i) = \begin{cases} 0 & \text{if } r(i) \leq c, \\ r(i) - c + \sum_{m=0}^{i} p(m \mid i) J_{\mu^1}(i-m) & \text{if } r(i) > c. \end{cases} \tag{4.32}$$

Note that the values $J_{\mu^1}(i)$ are nonnegative for all i, since by Prop. 4.5.1, we have

$$J_{\mu^1}(i) \geq J_{\mu^0}(i) = 0.$$

The next policy generated by PI is obtained from the minimization

$$\mu^2(i) = \arg\max\left[0, r(i) - c + \sum_{m=0}^{i} p(m \mid i) J_{\mu^1}(i-m)\right], \qquad i = 1, \ldots, n.$$

For i such that $r(i) \leq c$, we have $r(j) \leq c$ for all $j < i$ because $r(i)$ is monotonically nondecreasing in i. Moreover, using Eq. (4.32), we have $J_{\mu^1}(i - m) = 0$ for all $m \geq 0$. It follows that for i such that $r(i) \leq c$,

$$0 \geq r(i) - c + \sum_{m=0}^{i} p(m \mid i) J_{\mu^1}(i-m),$$

and $\mu^2(i) =$ *stop searching*.

For i such that $r(i) > c$, we have $\mu^2(i) =$ *search*, since $J_{\mu^1}(i) \geq 0$ for all i, so that

$$0 < r(i) - c + \sum_{m=0}^{i} p(m \mid i) J_{\mu^1}(i - m).$$

Thus, μ^2 is the same as μ^1 and the PI algorithm terminates. By Prop. 4.5.1, it follows that μ^2 is optimal.

4.5.2 Optimistic and Multistep Lookahead Policy Iteration

The PI algorithm that we have discussed so far uses exact policy evaluation of the current policy μ^k and one-step lookahead policy improvement, i.e., it computes exactly J_{μ^k}, and it obtains the next policy μ^{k+1} by a one-step lookahead minimization using J_{μ^k} as an approximation to J^*. It is possible to use a more flexible algorithm whereby J_{μ^k} is approximated by any number of value iterations corresponding to μ^k (cf. Prop. 4.3.3) and the policy improvement is done using multistep lookahead.

A PI algorithm that uses a finite number m_k of VI steps for policy evaluation of policy μ^k (in place of the infinite number required by exact PI) is referred to as *optimistic*. It can be viewed as a combination of VI and PI. The optimistic PI algorithm starts with a function J_0, an initial guess of J^*. It generates a sequence $\{J_k\}$ and an associated sequence of policies $\{\mu^k\}$, which asymptotically converge to J^* and an optimal policy, respectively. The kth iteration starts with a function J_k, and first generates μ^k. It then generates J_{k+1} using m_k iterations of the VI algorithm that corresponds to μ^k, starting with J_k as follows.

Optimistic Policy Iteration: Discounted Problems

Given the typical function J_k:

Policy improvement computes a policy μ^k such that

$$\mu^k(i) \in \arg \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i, u, j) + \alpha J_k(j)\big), \quad i = 1, \ldots, n. \tag{4.33}$$

Optimistic policy evaluation starts with $\hat{J}_{k,0} = J_k$, and uses m_k VI iterations for policy μ^k to compute $\hat{J}_{k,1}, \ldots, \hat{J}_{k,m_k}$ according to

$$\hat{J}_{k,m+1}(i) = \sum_{j=1}^{n} p_{ij}\big(\mu^k(i)\big)\Big(g\big(i, \mu^k(i), j\big) + \alpha \hat{J}_{k,m}(j)\Big), \tag{4.34}$$

for all $i = 1, \ldots, n$, $m = 0, \ldots, m_k - 1$, and sets $J_{k+1} = \hat{J}_{k,m_k}$.

From Eq. (4.34), it can be seen that one way to interpret optimistic PI is that *we approximate J_{μ^k} by using μ^k for m_k stages, and adding a terminal cost function equal to the current cost estimate* J_k instead of using μ^k for an additional infinite number of stages. Accordingly, simulation-based approximations of optimistic PI, evaluate the cost function J_{μ^k} by using m_k-stage trajectories, with the cost of future stages accounted for with some cost function approximation at the end of the m_k stages.

The convergence properties of optimistic PI are solid, although it may require an infinite number of iterations to converge to J^*. To see why this is so, suppose that we evaluate each policy with a *single* VI. Then the method is essentially identical to the VI method, which requires an infinite number of iterations to converge. For the same reason, optimistic PI, when implemented with approximations similar to VI, as in Section 4.4, is subject to the instability phenomenon illustrated in Example 4.4.1. Generally, most practical approximate policy evaluation schemes are optimistic in nature.

The following proposition, shown in the appendix, establishes the validity of optimistic PI. There is a corresponding convergence property for SSP problems, but its currently available proof is fairly complicated. It is given in Section 3.5.1 of the book [Ber12]. Asynchronous versions of optimistic PI also involve theoretical convergence difficulties, which are discussed in Section 2.6.2 of [Ber12] and Section 2.6.3 of [Ber18a].

Proposition 4.5.2: (Convergence of Optimistic PI) For the discounted problem, the sequences $\{J_k\}$ and $\{\mu^k\}$ generated by the optimistic PI algorithm satisfy

$$J_k \to J^*, \qquad J_{\mu^k} \to J^*.$$

The proof of the proposition is based on the policy improvement line of proof we gave earlier. In particular, if J_0 satisfies $T_{\mu^0} J_0 \leq J_0$, the argument of Eqs. (4.28)-(4.31) can be used to show that $J^* \leq J_{k+1} \leq J_k$ for all k. Moreover, the proof of the appendix argues that we may assume that $T_{\mu^0} J_0 \leq J_0$ holds without loss of generality, since we may add a constant to J_0 without affecting the sequence $\{\mu^k\}$ generated by the algorithm. The proof of the appendix also shows that the generated policies μ^k are optimal after some k, but this fact cannot be exploited in practice because the verification that μ^k is optimal requires additional computations that essentially defeat the purpose of the method.

Multistep Policy Improvement

The motivation for multistep policy improvement is that it may yield a

better policy μ^{k+1} than with one-step lookahead. In fact this makes even more sense when the evaluation of μ^k is approximate, since then the longer lookahead may compensate for errors in the policy evaluation. The method in its exact nonoptimistic form is given below (in a different version it may be combined with optimistic PI, i.e., with policy evaluation done using a finite number of VI iterations).

Multistep Lookahead Exact Policy Iteration: Discounted Problems

Given the typical policy μ^k:

Policy evaluation computes $J_{\mu^k}(i)$, $i = 1, \ldots, n$, as the solution of the (linear) system of Bellman equations

$$J_{\mu^k}(i) = \sum_{j=1}^{n} p_{ij}\big(\mu^k(i)\big)\Big(g\big(i, \mu^k(i), j\big) + \alpha J_{\mu^k}(j)\Big), \quad i = 1, \ldots, n,$$

(cf. Prop. 4.2.3).

Policy improvement with ℓ-step lookahead then solves the ℓ-stage problem with terminal cost function J_{μ^k}. If $\{\hat{\mu}_0, \ldots, \hat{\mu}_{\ell-1}\}$ is the optimal policy of this problem, then the new policy μ^{k+1} is $\hat{\mu}_0$.

The process is repeated with μ^{k+1} used in place of μ^k, unless we have $J_{\mu^{k+1}}(i) = J_{\mu^k}(i)$ for all i, in which case the algorithm terminates with the policy μ^k.

Exact PI with multistep lookahead has the same solid convergence properties as its one-step lookahead counterpart: it terminates with an optimal policy, and the generated sequence of policies is monotonically improving. The proof is based on a cost improvement property that will be shown as a special case of the subsequent Prop. 4.6.1.

4.5.3 Policy Iteration for Q-factors

Similar to VI, we may also equivalently implement PI through the use of Q-factors. To see this, first note that the policy improvement step may be implemented by minimizing over $u \in U(i)$ the expression

$$Q_\mu(i, u) = \sum_{j=1}^{n} p_{ij}(u)\big(g(i, u, j) + \alpha J_\mu(j)\big), \qquad i = 1, \ldots, n, \ u \in U(i),$$

which we view as the *Q-factor of the pair* (i, u) *corresponding to* μ. Note that we have

$$J_\mu(j) = Q_\mu\big(j, \mu(j)\big),$$

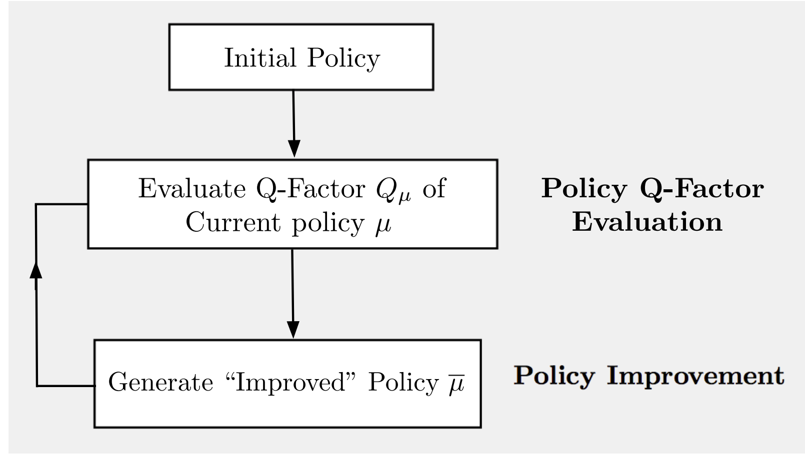

Figure 4.5.2 Block diagram of exact PI for Q-factors. Each iteration consists of a policy evaluation using the current policy μ, followed by generation of an improved policy $\tilde{\mu}$.

(cf. Prop. 4.2.3).

The following algorithm is thus obtained; see Fig. 4.5.2.

Exact Policy Iteration for Q-Factors: Discounted Problems

Given the typical policy μ^k:

Policy evaluation computes $Q_{\mu^k}(i,u)$, for all $i = 1,\dots,n$, and $u \in U(i)$, as the solution of the (linear) system of equations

$$Q_{\mu^k}(i,u) = \sum_{j=1}^{n} p_{ij}(u)\Big(g(i,u,j) + \alpha Q_{\mu^k}\big(j,\mu^k(j)\big)\Big). \tag{4.35}$$

Policy improvement then computes a new policy μ^{k+1} as

$$\mu^{k+1}(i) \in \arg\min_{u\in U(i)} Q_{\mu^k}(i,u), \qquad i = 1,\dots,n. \tag{4.36}$$

The process is repeated with μ^{k+1} used in place of μ^k, unless we have $J_{\mu^{k+1}}(i) = J_{\mu^k}(i)$ for all i, in which case the algorithm terminates with the policy μ^k.

Note that the system (4.35) has a unique solution, since from the uniqueness of solution of Bellman's equation, any solution must satisfy

$$Q_{\mu^k}\big(j,\mu^k(j)\big) = J_{\mu^k}(j).$$

Hence the Q-factors $Q_{\mu^k}\big(j,\mu^k(j)\big)$ are uniquely determined, and then the remaining Q-factors $Q_{\mu^k}(i,u)$ are also uniquely determined from Eq. (4.35).

The PI algorithm for Q-factors is mathematically equivalent to PI for costs, as given in the preceding subsection. The only difference is that we calculate *all* the Q-factors $Q_{\mu^k}(i,u)$, rather than just the costs $J_{\mu^k}(j) = Q_{\mu^k}\big(j,\mu^k(j)\big)$, i.e., just the Q-factors corresponding to the controls chosen by the current policy. However, the remaining Q-factors $Q_{\mu^k}(i,u)$ are needed for the policy improvement step (4.36), so no extra computation is required. It can be verified also that the PI algorithm (4.35)-(4.36) can be viewed as the PI algorithm for the discounted version of the modified problem of Fig. 4.2.2. Asynchronous and optimistic PI algorithms for Q-factors involve substantial theoretical convergence complications, as shown by Williams and Baird [WiB93], which have been resolved in papers by Bertsekas and Yu for discounted problems in [BeY12] and for SSP problems in [YuB13a].

4.6 APPROXIMATION IN VALUE SPACE - PERFORMANCE BOUNDS

We will focus on infinite horizon DP approximations, beginning with discounted problems. Consistently with the finite horizon approximation in value space schemes of Chapter 2, the general idea is to compute some approximation $\tilde{J}$ of the optimal cost function J^*, and then use one-step or multistep lookahead to implement a suboptimal policy $\tilde{\mu}$. Thus, a *one-step lookahead policy* applies at state i the control $\tilde{\mu}(i)$ that attains the minimum in the expression

$$\min_{u\in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha\tilde{J}(j)\big), \tag{4.37}$$

see Fig. 4.6.1.

Similarly, at state i, a *two-step lookahead policy* applies the control $\tilde{\mu}(i)$ attaining the minimum in the preceding equation, where now $\tilde{J}$ is obtained itself on the basis of a one-step lookahead approximation. In other words, for all states j that can be reached from i, we have

$$\tilde{J}(j) = \min_{u\in U(j)} \sum_{m=1}^{n} p_{jm}(u)\big(g(j,u,m) + \alpha\hat{J}(m)\big),$$

where $\hat{J}$ is some approximation of J^*. Thus $\tilde{J}$ is the result of a single value iteration starting from $\hat{J}$. Policies with lookahead of more than two stages are similarly defined. In particular, *the "effective one-step" cost approximation $\tilde{J}$ in ℓ-step lookahead is the result of $\ell - 1$ successive value iterations starting from some initial approximation $\hat{J}$.* Otherwise expressed,

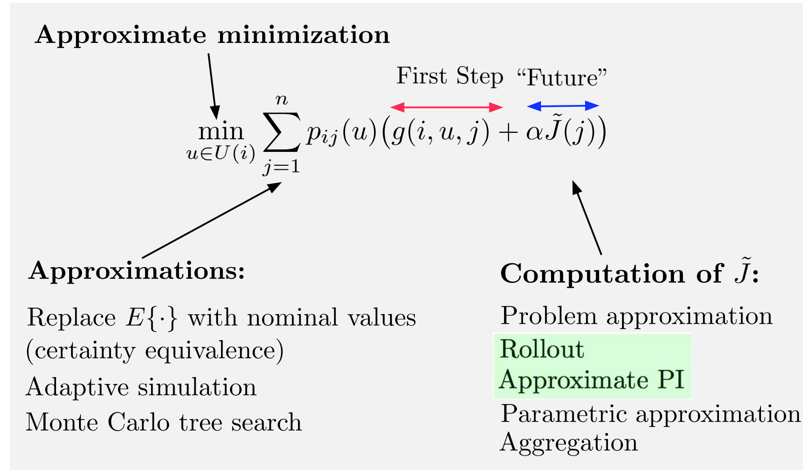

Figure 4.6.1 Schematic illustration of various options for approximation in value space with one-step lookahead in infinite horizon problems. The lookahead function values $\tilde J(j)$ approximate the optimal cost-to-go values $J^*(j)$, and can be computed by a variety of methods. There may be additional approximations in the minimization over u_k and the computation of the expected value.

ℓ-step lookahead with $\hat J$ at the end is the same as one-step lookahead with $T^{\ell-1}\hat J$ at the end, where T is the DP operator (4.13).

In Chapter 2 we gave several types of limited lookahead schemes, where $\tilde J$ is obtained in different ways, such as problem approximation, rollout, and others. Several of these schemes can be fruitfully adapted to infinite horizon problems; see Fig. 4.6.1.

In this chapter, we will focus on rollout, and particularly on approximate PI schemes, which operate as follows:

(a) Several policies $\mu^0, \mu^1, \ldots, \mu^m$ are generated, starting with an initial policy μ^0.

(b) Each policy μ^k is evaluated approximately, with a cost function $\tilde J_{\mu^k}$, often with the use of a parametric approximation/neural network approach.

(c) The next policy μ^{k+1} is generated by one-step or multistep policy improvement based on $\tilde J_{\mu^k}$.

(d) The approximate evaluation $\tilde J_{\mu^m}$ of the last policy in the sequence is used as the lookahead approximation $\tilde J$ in the one-step lookahead minimization (4.37), or its multistep counterpart.

Performance bounds for this type of approximate PI scheme will be discussed in Section 4.6.3, following a discussion of general performance bounds and rollout in the next two subsections. Note that rollout can be viewed as

the extreme special case of the preceding approximate PI procedure, where $m = 0$, and only the policy μ^0 is evaluated and used for a single policy improvement.

4.6.1 Limited Lookahead Performance Bounds

We will now consider performance bounds for ℓ-step lookahead. In particular, if $\hat{\mu}_0, \ldots, \hat{\mu}_{\ell-1}$ attain the minimum in the ℓ-step lookahead minimization below:

$$\min_{\mu_0,\ldots,\mu_{\ell-1}} E\left\{\sum_{k=0}^{\ell-1} \alpha^k g\big(i_k, \mu_k(i_k), j_k\big) + \alpha^\ell \tilde{J}(i_\ell)\right\},$$

we consider the suboptimal policy $\tilde{\mu} = \hat{\mu}_0$. We will refer to $\tilde{\mu}$ as the *ℓ-step lookahead policy corresponding to $\tilde{J}$*. Equivalently, in the shorthand notation of the DP operators T and $T_{\tilde{\mu}}$ of Eqs. (4.13) and (4.14), the ℓ-step lookahead policy $\tilde{\mu}$ is defined by

$$T_{\tilde{\mu}}(T^{\ell-1}\tilde{J}) = T^\ell \tilde{J}.$$

In part (a) of the following proposition, we will derive a bound for the performance of $\tilde{\mu}$.

We will also derive a bound for the case of a useful generalized one-step lookahead scheme [part (b) of the following proposition]. This scheme aims to reduce the computation to obtain $\tilde{\mu}(i)$, by performing the lookahead minimization over a subset $\overline{U}(i) \subset U(i)$. Thus, the control $\tilde{\mu}(i)$ used in this scheme is one that attains the minimum in the expression

$$\min_{u\in\overline{U}(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha\tilde{J}(j)\big).$$

This is attractive when by using some heuristic or approximate optimization, we can identify a subset $\overline{U}(i)$ of promising controls, and to save computation, we restrict attention to this subset in the one-step lookahead minimization.

Proposition 4.6.1: (Limited Lookahead Performance Bounds)

(a) Let $\tilde{\mu}$ be the ℓ-step lookahead policy corresponding to $\tilde{J}$. Then

$$\|J_{\tilde{\mu}} - J^*\| \leq \frac{2\alpha^\ell}{1-\alpha}\|\tilde{J} - J^*\|, \tag{4.38}$$

where $\|\cdot\|$ denotes the maximum norm (4.15).

(b) Let $\tilde{\mu}$ be the one-step lookahead policy obtained by minimization in the equation

$$\hat{J}(i) = \min_{u \in \overline{U}(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha \tilde{J}(j)\big), \qquad i = 1, \ldots, n, \tag{4.39}$$

where $\overline{U}(i) \subset U(i)$ for all $i = 1, \ldots, n$. Assume that for some constant c, we have

$$\hat{J}(i) \leq \tilde{J}(i) + c, \qquad i = 1, \ldots, n. \tag{4.40}$$

Then

$$J_{\tilde{\mu}}(i) \leq \hat{J}(i) + \frac{c}{1-\alpha}, \qquad i = 1, \ldots, n. \tag{4.41}$$

An important point regarding the bound (4.38) is that $J_{\tilde{\mu}}$ is unaffected by a constant shift in $\tilde{J}$ [an addition of a constant to all values $\tilde{J}(i)$]. Thus $\|\tilde{J} - J^*\|$ in Eq. (4.38) can be replaced by the potentially much smaller number

$$\min_{\beta \in \Re} \|\tilde{J} + \beta e - J^*\| = \min_{\beta \in \Re} \max_{i=1,\ldots,n} \big|\tilde{J}(i) + \beta - J^*(i)\big|.$$

We thus obtain the following performance bound,

$$\|J_{\tilde{\mu}} - J^*\| \leq \frac{2\alpha^\ell}{1-\alpha} \min_{\beta \in \Re} \max_{i=1,\ldots,n} \big|\tilde{J}(i) + \beta - J^*(i)\big|,$$

which is stronger than the one of Eq. (4.38) that corresponds to $\beta = 0$.

The preceding bound shows that performance is improved when the length ℓ of the lookahead in increased, and also when the lookahead cost approximation $\tilde{J}$ is closer to the optimal cost J^* (when modified with an optimal constant shift β). Both of these conclusions are intuitive and also consistent with practical experience. Note that we are not asserting that multistep lookahead will lead to better performance than one-step lookahead; we know that this not is necessarily true (cf. Example 2.2.1). It is the performance *bound* that is improved when multistep lookahead is used.

Regarding the condition (4.40), we note that it guarantees that when $c \leq 0$, the cost $J_{\tilde{\mu}}$ of the one-step lookahead policy is no larger than $\tilde{J}$. When $c = 0$, this condition bears resemblance with the consistent improvement condition for deterministic rollout methods (cf. Section 2.4.1). If $\tilde{J} = J_\mu$ for some policy μ (as in the case of the pure form of rollout to

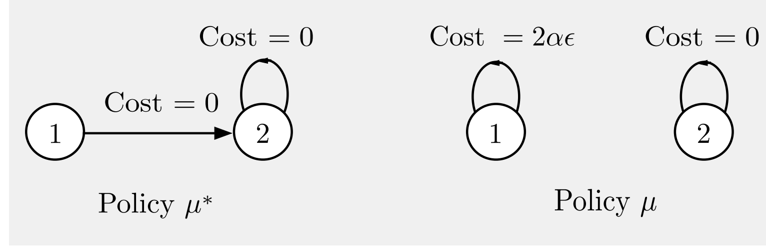

Figure 4.6.2 A two-state problem for proving the tightness of the performance bound of Prop. 4.6.1(b) (cf. Example 4.6.1). All transitions are deterministic as shown, but at state 1 there are two possible decisions: move to state 2 (policy μ^*) or stay at state 1 (policy μ). The cost of each transition is shown next to the corresponding arc.

be discussed in Section 4.6.2), then Eq. (4.40) holds as an equation with $c = 0$, and from Eq. (4.41), it follows that $J_{\tilde{\mu}} \leq J_\mu$.

Unfortunately, the bound (4.38) is not very reassuring when α is close to 1. Nonetheless, the following example shows that the bound is tight in very simple problems with just two states. What is happening here is that an $O(\epsilon)$ difference in single stage cost between two controls can generate an $O\big(\epsilon/(1-\alpha)\big)$ difference in policy costs, yet it can be "nullified" in Bellman's equation by an $O(\epsilon)$ difference between J^* and $\tilde{J}$.

Example 4.6.1

Consider the two-state discounted problem shown in Fig. 4.6.2, where ϵ is a positive scalar and $\alpha \in [0, 1)$ is the discount factor. The optimal policy μ^* is to move from state 1 to state 2, and the optimal cost-to-go function is $J^*(1) = J^*(2) = 0$. Consider the cost function approximation $\tilde{J}$ with

$$\tilde{J}(1) = -\epsilon, \qquad \tilde{J}(2) = \epsilon,$$

so that

$$\|\tilde{J} - J^*\| = \epsilon,$$

as assumed in Eq. (4.38) [cf. Prop. 4.6.1(b)]. The policy μ that decides to stay at state 1 is a one-step lookahead policy based on $\tilde{J}$, because

$$2\alpha\epsilon + \alpha\tilde{J}(1) = \alpha\epsilon = 0 + \alpha\tilde{J}(2).$$

Moreover, we have

$$J_\mu(1) = \frac{2\alpha\epsilon}{1-\alpha} = \frac{2\alpha}{1-\alpha}\|\tilde{J} - J^*\|,$$

so the bound of Eq. (4.38) holds with equality.

4.6.2 Rollout

Let us first consider rollout in its pure form, where $\tilde{J}$ in Eq. (4.37) is the cost function of some stationary policy μ (also called the *base policy* or *base heuristic*), i.e., $\tilde{J} = J_\mu$. Thus, *the rollout policy is the result of a single policy iteration starting from* μ. The policy evaluation that yields the costs $J_\mu(j)$ needed for policy improvement may be done in any suitable way. Monte-Carlo simulation (averaging the costs of many trajectories starting from j) is one major possibility. Of course if the problem is deterministic, a single simulation trajectory starting from j is sufficient, in which case the rollout policy is much less computationally demanding. Note also that in discounted problems the simulated trajectories must be truncated after a number of transitions, which is sufficiently large to make the cost of the remaining transitions insignificant in view of the discount factor.

An important fact is that in the pure form of rollout, *the rollout policy improves over the base policy*, as the following proposition shows. This is to be expected since rollout is one-step PI, so Prop. 4.5.1 applies.

Proposition 4.6.2: (Cost Improvement by Rollout) Let $\tilde{\mu}$ be the rollout policy obtained by the one-step lookahead minimization

$$\min_{u\in\overline{U}(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha J_\mu(j)\big),$$

where μ is a base policy [cf. Eq. (4.39) with $\tilde{J} = J_\mu$] and we assume that $\mu(i) \in \overline{U}(i) \subset U(i)$ for all $i = 1,\ldots,n$. Then $J_{\tilde{\mu}} \le J_\mu$.

Let us also mention the variation of rollout that uses multiple base heuristics, and simultaneously improves on all of them. This variant, also called *parallel rollout* because of its evident parallelization potential, extends its finite horizon counterpart; cf. Section 2.4.1.

Example 4.6.2 (Rollout with Multiple Heuristics)

Let $\mu_1,\ldots,\mu_M$ be stationary policies, let

$$\tilde{J}(i) = \min\big\{J_{\mu_1}(i),\ldots,J_{\mu_M}(i)\big\}, \qquad i = 1,\ldots,n,$$

let $\overline{U}(i) \subset U(i)$, and assume that $\mu_1(i),\ldots,\mu_M(i) \in \overline{U}(i)$ for all $i = 1,\ldots,n$. Then, for all i and $m = 1,\ldots,M$, we have

$$\hat{J}(i) = \min_{u\in\overline{U}(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha\tilde{J}(j)\big)$$

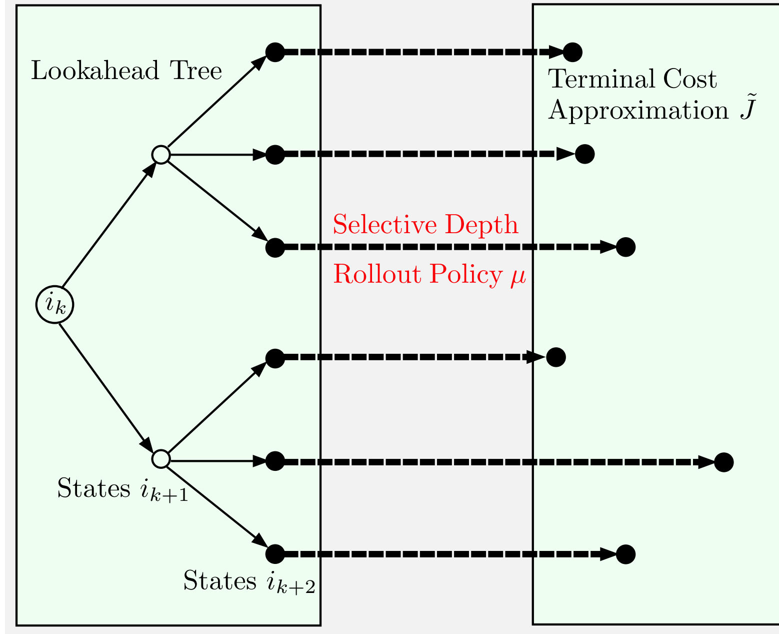

Figure 4.6.3 Illustration of two-step lookahead, rollout with a policy μ for a limited and state-dependent number of steps, and a terminal cost function approximation $\tilde{J}$. A Monte Carlo tree search scheme may also be used for multistep lookahead; cf. Section 2.4.2. Note that the three components of this scheme (multistep lookahead, rollout with μ, and cost approximation $\tilde{J}$) can be designed independently of each other. Moreover, while the multistep lookahead is implemented on-line, μ and $\tilde{J}$ must be available from an earlier off-line computation.

$$
\begin{aligned}
&\le \min_{u\in\overline{U}(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j)+\alpha J_{\mu_m}(j)\big)\\
&\le \sum_{j=1}^{n} p_{ij}\big(\mu_m(i)\big)\Big(g\big(i,\mu_m(i),j\big)+\alpha J_{\mu_m}(j)\Big)\\
&= J_{\mu_m}(i),
\end{aligned}
$$

from which, by taking minimum of the right-hand side over m, it follows that

$$
\hat{J}(i) \le \tilde{J}(i), \qquad i=1,\dots,n.
$$

Using Prop. 4.6.1(a), we see that the rollout policy $\tilde{\mu}$, obtained by using $\tilde{J}$ as one-step lookahead approximation satisfies

$$
J_{\tilde{\mu}}(i) \le \min\big\{J_{\mu_1}(i),\dots,J_{\mu_M}(i)\big\}, \qquad i=1,\dots,n,
$$

i.e., it improves over each of the policies $\mu_1,\dots,\mu_M$.

Combined Multistep Lookahead, Rollout, and Terminal Cost Approximation

Let us next discuss a variant of the rollout approach, whereby we use ℓ-step lookahead, we then apply rollout with policy μ for a limited number of steps, and finally we approximate the cost of the remaining steps using some terminal cost approximation $\tilde{J}$; see Fig. 4.6.3. We can view this form of rollout as *a single optimistic policy iteration combined with multistep lookahead*; cf. Eqs. (4.33)-(4.34). This type of algorithm was used in Tesauro's rollout-based backgammon player [TeG96] (it was also used in AlphaGo in a modified form, with Monte Carlo tree search in place of ordinary limited lookahead). We will give more details later.

The following result generalizes the performance bounds given for limited lookahead and rollout of the preceding two subsections. In particular, part (a) of the proposition follows by applying Prop. 4.6.1(a), since the truncated rollout scheme of this section can be viewed as ℓ-step approximation in value space with terminal cost function $T_\mu^m \tilde{J}$ at the end of the lookahead, where T_μ is the DP operator of Eq. (4.14).

Proposition 4.6.3: (Performance Bound of Rollout with Terminal Cost Function Approximation) Let ℓ and m be positive integers, let μ be a policy, and let $\tilde{J}$ be a function of the state. Consider a truncated rollout scheme consisting of ℓ-step lookahead, followed by rollout with a policy μ for m steps, and a terminal cost function approximation $\tilde{J}$ at the end of the m steps. Let $\tilde{\mu}$ be the policy generated by this scheme.

(a) We have

$$\|J_{\tilde{\mu}} - J^*\| \le \frac{2\alpha^\ell}{1-\alpha}\|T_\mu^m \tilde{J} - J^*\|,$$

where T_μ is the DP operator of Eq. (4.14), and $\|\cdot\|$ denotes the maximum norm (4.15).

(b) Assume that for some constant c, $\tilde{J}$ and μ satisfy the condition

$$\hat{J}(i) = \sum_{j=1}^{n} p_{ij}\big(\mu(i)\big)\Big(g\big(i,\mu(i),j\big) + \alpha\tilde{J}(j)\Big) \le \tilde{J}(i) + c, \tag{4.42}$$

for all $i = 1,\ldots,n$. Then

$$J_{\tilde{\mu}}(i) \le \tilde{J}(i) + \frac{c}{1-\alpha}, \qquad i = 1,\ldots,n. \tag{4.43}$$

As a special case of part (b) of the preceding proposition, suppose that the terminal cost function $\tilde{J}$ approximates within $c/(1+\alpha)$ the cost function of μ,

$$\big|\tilde{J}(i) - J_\mu(i)\big| \leq \frac{c}{1+\alpha}, \qquad i = 1, \ldots, n.$$

Then Eq. (4.42) is satisfied since we have

$$\begin{aligned}
\hat{J}(i) &= \sum_{j=1}^{n} p_{ij}\big(\mu(i)\big)\Big(g\big(i,\mu(i),j\big) + \alpha\tilde{J}(j)\Big) \\
&\leq \sum_{j=1}^{n} p_{ij}\big(\mu(i)\big)\Big(g\big(i,\mu(i),j\big) + J_\mu(j)\Big) + \frac{\alpha c}{1+\alpha} \\
&= J_\mu(i) + \frac{\alpha c}{1+\alpha} \\
&\leq \tilde{J}(i) + \frac{c}{1+\alpha} + \frac{\alpha c}{1+\alpha} \\
&= \tilde{J}(i) + c.
\end{aligned}$$

The proposition then shows that multistep lookahead followed by infinite step rollout with μ produces a rollout policy $\tilde{\mu}$ with

$$J_{\tilde{\mu}}(i) \leq \tilde{J}(i) + \frac{c}{1-\alpha} \leq J_\mu(i) + \frac{c}{1+\alpha} + \frac{c}{1-\alpha} = J_\mu(i) + \frac{2c}{1-\alpha^2}$$

for all i. Thus, *if $\tilde{J}$ is nearly equal to J_μ, then $\tilde{\mu}$ nearly improves over μ* [within $2c/(1-\alpha^2)$].

There is also an extension for the case where $m = 0$, i.e., when there is no rollout with a policy μ. It states that under the condition

$$\min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha\tilde{J}(j)\big) \leq \tilde{J}(i) + c, \qquad i = 1, \ldots, n,$$

the multistep lookahead policy $\tilde{\mu}$ satisfies

$$J_{\tilde{\mu}}(i) \leq \tilde{J}(i) + \frac{c}{1-\alpha}$$

for all i. This performance bound is similar to fairly old bounds that date to the mid-90s; see Prop. 6.1.1 in the author's book [Ber17] (and its earlier editions). It extends Prop. 4.6.1(b) from one-step to multistep lookahead approximation in value space schemes.

Regarding the nature of the terminal cost approximation $\tilde{J}$ in truncated rollout schemes, it may be heuristic, based on problem approximation, or based on a more systematic simulation methodology. For example,

the values $J_\mu(i)$ may be computed by simulation for all i in a subset of representative states, and $\tilde{J}$ may be selected from a parametric class of vectors by a least squares regression of the computed values. This approximation may be performed off-line, outside the time-sensitive restrictions of a real-time implementation, and the result $\tilde{J}$ may be used on-line in place of J_μ as a terminal cost function approximation. Note, however, that once cost function approximation is introduced at the end of the rollout, the cost improvement property of the rollout policy over the base policy may be lost.

The truncated rollout scheme of Fig. 4.6.3 has been adopted in the rollout backgammon algorithm of Tesauro and Galperin [TeG96], with μ and the terminal cost function approximation $\tilde{J}$ provided by the TD-Gammon algorithm of Tesauro [Tes94], which was based on a neural network, trained using a form of optimistic policy iteration and TD(λ). A similar type of algorithm was used in the AlphaGo program (Silver et al. [SHM16]), with the policy and the terminal cost function obtained with a deep neural network, trained using a form of approximate policy iteration. Also the multistep lookahead in the AlphaGo algorithm was implemented using Monte Carlo tree search (cf. Section 2.4.2).

4.6.3 Approximate Policy Iteration

When the number of states is very large, the policy evaluation step and/or the policy improvement step of the PI method may be implementable only through approximations. In an approximate PI scheme, each policy μ^k is evaluated approximately, with a cost function $\tilde{J}_{\mu^k}$, often with the use of a feature-based architecture or a neural network, and the next policy μ^{k+1} is generated by (perhaps approximate) policy improvement based on $\tilde{J}_{\mu^k}$.

To formalize this type of procedure, we assume an approximate policy evaluation error satisfying

$$\max_{i=1,\ldots,n} \big|\tilde{J}_{\mu^k}(i) - J_{\mu^k}(i)\big| \le \delta, \tag{4.44}$$

and an approximate policy improvement error satisfying

$$\max_{i=1,\ldots,n} \left| \sum_{j=1}^n p_{ij}\big(\mu^{k+1}(i)\big)\big(g(i,\mu^{k+1}(i),j) + \alpha \tilde{J}_{\mu^k}(j)\big) - \min_{u\in U(i)} \sum_{j=1}^n p_{ij}(u)\big(g(i,u,j) + \alpha \tilde{J}_{\mu^k}(j)\big) \right| \le \epsilon, \tag{4.45}$$

where δ and ϵ are some nonnegative scalars. The following proposition, proved in the appendix (and also in the original source [BeT96], Section 6.2.2), provides a performance bound for discounted problems (a similar result is available for SSP problems; see [BeT96], Section 6.2.2).

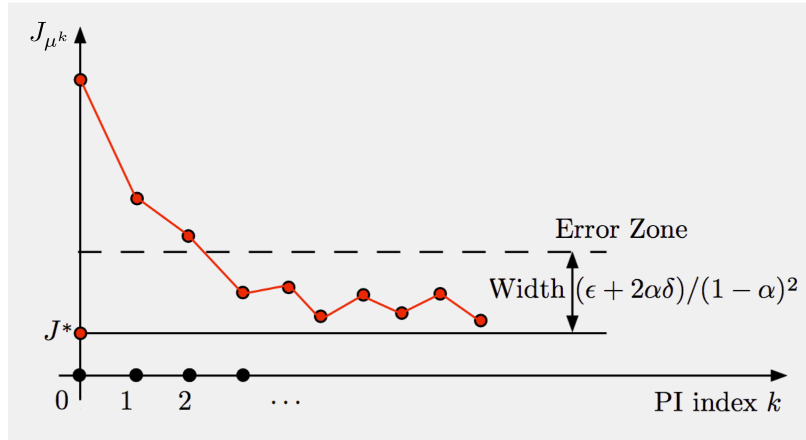

Figure 4.6.4 Illustration of typical behavior of approximate PI. In the early iterations, the method tends to make rapid and fairly monotonic progress, until J_{μ^k} gets within an error zone of size less than $(\epsilon + 2\alpha\delta)/(1-\alpha)^2$. After that J_{μ^k} oscillates randomly within that zone.

Proposition 4.6.4: (Performance Bound for Approximate PI) Consider the discounted problem, and let $\{\mu^k\}$ be the sequence generated by the approximate PI algorithm defined by the approximate policy evaluation (4.44) and the approximate policy improvement (4.45). Then the policy error

$$\max_{i=1,\ldots,n} \big| J_{\mu^k}(i) - J^*(i) \big|,$$

becomes less or equal to

$$\frac{\epsilon + 2\alpha\delta}{(1-\alpha)^2},$$

asymptotically as $k \to \infty$.

The preceding performance bound is not particularly useful in practical terms. Significantly, however, it is in qualitative agreement with the empirical behavior of approximate PI. In the beginning, the method tends to make rapid and fairly monotonic progress, but eventually it gets into an oscillatory pattern. This happens after J_{μ^k} gets within an error zone of size $(\delta + 2\epsilon)/(1-\alpha)^2$ or smaller, and then J_{μ^k} oscillates fairly randomly within that zone; see Fig. 4.6.4. In practice, the error bound of Prop. 4.6.4 tends to be pessimistic, so the zone of oscillation is usually much narrower than what is suggested by the bound. However, the bound itself can be

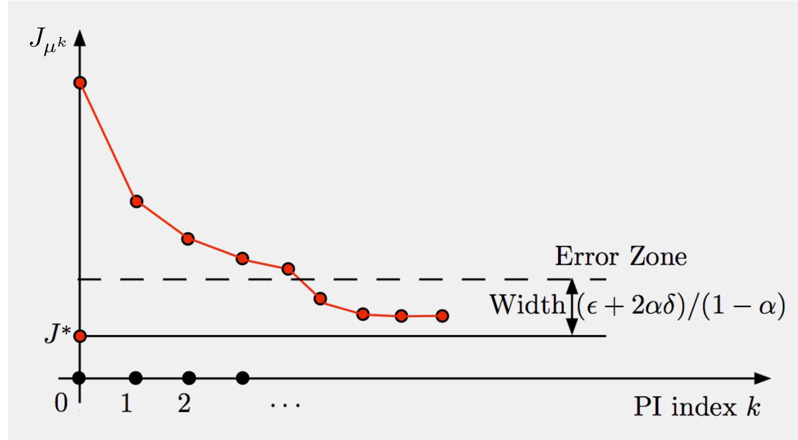

Figure 4.6.5 Illustration of typical behavior of approximate PI when policies converge. The method tends to make monotonic progress, and J_{μ^k} converges within an error zone of size less than

$$\frac{\epsilon + 2\alpha\delta}{1-\alpha}.$$

proved to be tight, in worst case. This is shown with an example in the book [BeT96], Section 6.2.3. Note also that the bound of Prop. 4.6.4 holds in the case of infinite state and control spaces discounted problems, when there are infinitely many policies (see [Ber18a], Prop. 2.4.3).

We finally note that since the set of policies is finite, the sequence $\{J_{\mu^k}\}$ is guaranteed to be bounded, so approximate PI is not hampered by the instability that was highlighted by Example 4.4.1 for approximate VI.

Performance Bound for the Case Where Policies Converge

Generally, the policy sequence $\{\mu^k\}$ generated by approximate PI may oscillate between several policies, as noted earlier. However, under some circumstances the sequence will converge to some policy $\tilde{\mu}$, in the sense that

$$\mu^{\overline{k}+1} = \mu^{\overline{k}} = \tilde{\mu} \qquad \text{for some } \overline{k}. \tag{4.46}$$

An important case where this happens is aggregation methods, which will be discussed in Chapter 5. In this case the behavior of the method is more regular, and we can show a more favorable bound than the one of Prop. 4.6.4, by a factor

$$\frac{1}{1-\alpha};$$

see Fig. 4.6.5.

Proposition 4.6.5: (Performance Bound for Approximate PI when Policies Converge) Let $\tilde{\mu}$ be a policy generated by the approximate PI algorithm under conditions (4.44), (4.45), and (4.46). Then we have

$$\max_{i=1,\ldots,n} \big|J_{\tilde{\mu}}(i) - J^*(i)\big| \leq \frac{\epsilon + 2\alpha\delta}{1-\alpha}.$$

We finally note that similar performance bounds can be obtained for optimistic PI methods, where the policy evaluation is performed with just a few approximate value iterations, and policy improvement is approximate (cf. Section 4.5.2). These bounds are similar to the ones of the nonoptimistic PI case given in this section, but their derivation is quite complicated; see [Ber12], Chapter 2, or [Ber18a], Section 2.5.2, and the end-of-chapter references. It should be noted, however, that in the absence of special modifications, optimistic PI with approximations is subject to the error amplification phenomenon illustrated in Example 4.4.1. Indeed approximate VI, as described in Section 4.4, can be viewed as a special case of an optimistic PI method, where each policy evaluation is done with a single VI, and then approximated by least squares/regression.

4.7 SIMULATION-BASED POLICY ITERATION WITH PARAMETRIC APPROXIMATION

In this section we will discuss PI methods where the policy evaluation step is carried out with the use of a parametric approximation method and Monte-Carlo simulation. We will focus on the discounted problem, but similar methods can be used for SSP problems.

4.7.1 Self-Learning and Actor-Critic Systems

The name "self-learning" in RL usually refers to some form of PI method that involves the use of simulation for approximate policy evaluation, and/or approximate Q-factor evaluation. A parametric architecture is used for this, and the algorithm that performs the policy evaluation is usually called a *critic*. If a neural network is used as the parametric architecture, it is called a *critic network*. The PI algorithm generates a sequence of stationary policies $\{\mu^k\}$ and a corresponding sequence of approximate cost function evaluations $\{\tilde{J}_{\mu^k}\}$ using a simulator of the system.

As in all PI methods, the policy evaluation $\tilde{J}_{\mu^k}$ is used for policy improvement, to generate the next policy μ^{k+1}. The algorithm that performs

the policy improvement is usually called an *actor*, and if a neural network is involved, it is called an *actor network*.

The two operations needed at each policy iteration are as follows:

(a) *Evaluate the current policy μ^k (critic)*: Here algorithm, system, and simulator are merged in one, and the system "observes itself" by generating simulation cost samples under the policy μ^k. It then combines these samples to "learn" a policy evaluation $\tilde J_{\mu^k}$. Usually this is done through some kind of incremental method that involves a least squares minimization using cost samples, and either a linear architecture or a neural network.

(b) *Improve the current policy μ^k (actor)*: Given the approximate policy evaluation $\tilde J_{\mu^k}$, the system can generate or "learn" the new policy μ^{k+1} through the minimization

$$\mu^{k+1}(i) \in \arg\min_{u\in U(i)} \sum_{j=1}^n p_{ij}(u)\big(g(i,u,j)+\alpha \tilde J_{\mu^k}(j)\big), \qquad i=1,\ldots,n.$$

Alternatively the system can compute the minimizing control u^s at a set of sample states i^s, $s=1,\ldots,q$, through

$$u^s \in \arg\min_{u\in U(i^s)} \sum_{j=1}^n p_{i^sj}(u)\big(g(i^s,u,j)+\alpha \tilde J_{\mu^k}(j)\big).$$

These are the sample values of the improved policy μ^{k+1} at the sample states i^s. They are generalized to "learn" a complete policy μ^{k+1} by using some approximation in policy space scheme (cf. Section 2.1.3).

We can thus describe simulation-based PI as *a process where the system learns better and better policies by observing its behavior*. This is true up to the point where either policy oscillations occur (cf. Fig. 4.6.4) or the algorithm terminates (cf. Fig. 4.6.5), at which time learning essentially stops.

It is worth noting that the system learns by itself, but it does not learn itself, in the sense that *it does not construct a mathematical model for itself*. It only learns to behave better, i.e., construct improved policies, through experience gained by simulating state and control trajectories generated with these policies. We may adopt instead an alternative two-phase approach: first use system identification and simulation to construct a mathematical model of the system, and then use a model-based PI method. However, we will not discuss this approach in this book.

4.7.2 A Model-Based Variant

We will first provide an example of a model-based PI method that is conceptually simple, and then discuss its model-free version. In particular, we

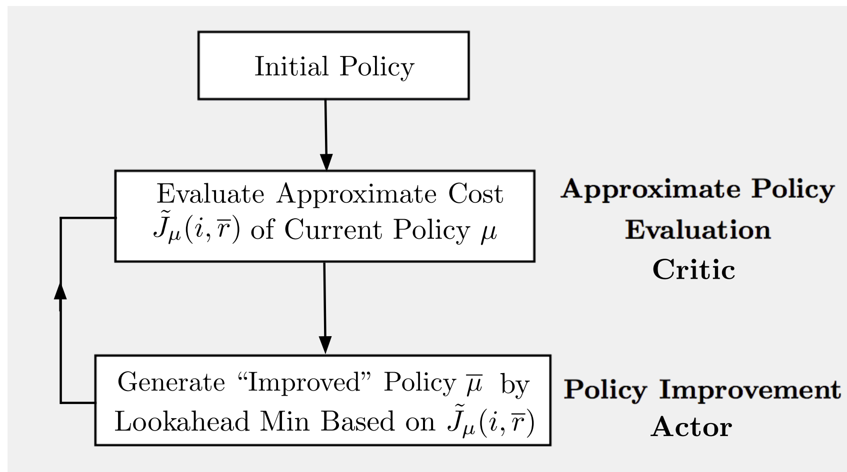

Figure 4.7.1 Block diagram of model-based approximate PI for cost functions.

assume that the transition probabilities $p_{ij}(u)$ are available, and that the cost function J_μ of any given policy μ is approximated using a parametric architecture $\tilde{J}_\mu(i,r)$.

We recall that given any policy μ, the exact PI algorithm for costs [cf. Eqs. (4.34)-(4.33)] generates the new policy $\tilde{\mu}$ with a policy evaluation/policy improvement process. We approximate this process as follows; see Fig. 4.7.1.

(a) *Approximate policy evaluation*: To evaluate μ, we determine the value of the parameter vector r by generating a large number of training pairs (i^s,β^s), $s=1,\ldots,q$, and by using least squares training:

$$\overline{r} \in \arg\min_r \sum_{s=1}^{q} \big(\tilde{J}_\mu(i^s,r)-\beta^s\big)^2. \tag{4.47}$$

For a given state i^s, the scalar β^s is a sample cost corresponding to i^s and μ.

In particular β^s is generated by starting at i^s, simulating a trajectory of states and controls using μ and the known transition probabilities for some number N of stages, accumulating the corresponding discounted costs, and adding a terminal cost approximation

$$\alpha^N \hat{J}(i_N),$$

where i_N is the terminal state of the N-stage trajectory and $\hat{J}$ is some initial guess of J_μ. The guess $\hat{J}$ may be obtained with additional training or some other means, such as using the result of the policy

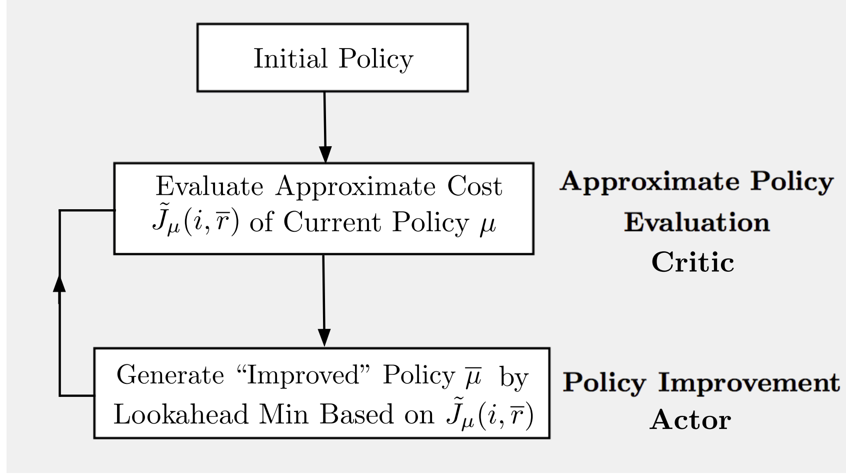

Figure 4.7.1 Block diagram of model-based approximate PI for cost functions.

evaluation of the preceding policy μ^{k-1}; this is similar to the cost function approximation implicitly used in optimistic policy iteration, cf. Section 4.5.2. It is also possible to simplify the method by using $\hat{J}(i_N) = 0$, or obtaining $\hat{J}$ via a problem approximation process.

The approximate policy evaluation problem of Eq. (4.47) can be solved with the incremental methods discussed in Section 3.1.3. In particular the incremental gradient method is given by

$$r^{k+1} = r^k - \gamma^k \nabla \tilde{J}(i^{s_k}, r^k)\big(\tilde{J}(i^{s_k}, r^k) - \beta^{s_k}\big),$$

where (i^{s_k}, β^{s_k}) is the state-cost sample pair that is used at the kth iteration, and r^0 is an initial parameter guess. Here the approximation architecture $\tilde{J}(i, r)$ may be linear or may be nonlinear and differentiable. In the case of a linear architecture it is also possible to solve the problem (4.47) using the exact linear least squares formula.

(b) *Approximate policy improvement*: Having solved the approximate policy evaluation problem (4.47), the new "improved" policy $\tilde{\mu}$ is obtained by the approximate policy improvement operation

$$\tilde{\mu}(i) \in \arg \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i, u, j) + \alpha \tilde{J}(j, \overline{r})\big), \qquad i = 1, \ldots, n, \tag{4.48}$$

where $\overline{r}$ is the parameter vector obtained from the policy evaluation operation (4.47).

Trajectory Reuse and Bias-Variance Tradeoff

As we have noted, to each training pair (i^s, β^s) there corresponds an N-stage trajectory over which the sample cost β^s is accumulated, but the length of the trajectory may depend on s. This allows sampling effort economies based on *trajectory reuse*. In particular, suppose that starting at some state i_0 we generate a long trajectory $(i_0, i_1, \ldots, i_N)$ using the policy μ. Then we can obtain the state-cost sample that corresponds to i_0, as discussed above, but we can also obtain additional cost samples for the subsequent states i_1, i_2, etc, by using the tail portions of the trajectory $(i_0, i_1, \ldots, i_N)$ that start at these states.

Clearly, it is necessary to truncate the sample trajectories to some number of stages N, since we cannot simulate an infinite length trajectory in practice. If N is large, then because of the discount factor, the error for neglecting the stage costs beyond stage N will be small. However, there are other important concerns when choosing the trajectory lengths N.

In particular, a short length reduces the sampling effort, but is also a source of inaccuracy. The reason is that the cost of the tail portion of the trajectory (from stage N to infinity) is approximated by $\alpha^N \hat{J}(i_N)$, where i_N is the terminal state of the N-stage trajectory and $\hat{J}$ is the initial guess of J_μ. This terminal cost compensates for the costs of the neglected stages in the spirit of optimistic PI, but adds an error to the cost samples β^s, which becomes larger as the trajectory length N becomes smaller.

We note two additional benefits of using many training trajectories, each with a relatively short trajectory length:

(1) The cost samples β^s are less noisy, as they correspond to summation of fewer random stage costs. This leads to the so-called *bias-variance tradeoff*: short trajectories lead to larger bias but smaller variance of the cost samples.

(2) With more starting states i_0, there is better opportunity for *exploration of the state space*. By this we mean adequate representation of all possible initial trajectory states in the sample set. This is a major issue in approximate PI, as we will discuss in Section 4.7.4.

Let us also note that the bias-variance tradeoff underlies the motivation for a number of alternative policy evaluation methods such as TD(λ), LSTD(λ), and LSPE(λ), which we will summarize in Section 4.9; see Section 6.3 of the book [Ber12] and other approximate DP/RL books referenced earlier. The papers [Ber11b], [YuB12], and the book [Ber12], Section 6.4, discuss a broad range of short trajectory sampling methods.

4.7.3 A Model-Free Variant

We will now provide an example model-free PI method. Let us restate the PI method in terms of Q-factors, and in a form that involves approx-

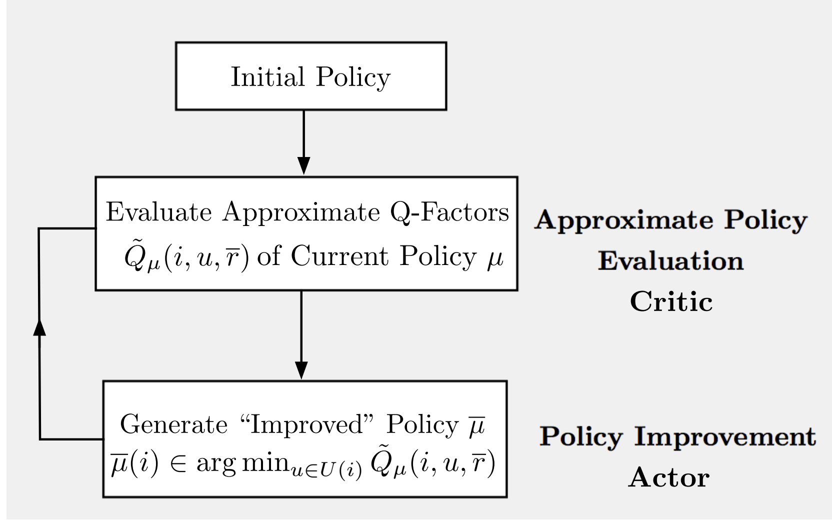

Figure 4.7.2 Block diagram of model-free approximate PI for Q-factors.

imations and simulation-based implementations. We recall that given any policy μ, the exact PI algorithm for Q-factors [cf. Eqs. (4.35)-(4.36)] generates the new policy $\tilde{\mu}$ with a policy evaluation-policy improvement process. We approximate this process as follows; see Fig. 4.7.2.

(a) *Approximate policy evaluation*: Here we introduce a parametric architecture $\tilde{Q}_\mu(i,u,r)$ for the Q-factors of μ. We determine the value of the parameter vector r by generating (using a simulator of the system) a large number of training triplets (i^s,u^s,β^s), $s=1,\ldots,q$, and by using a least squares fit:

$$\overline{r} \in \arg\min_r \sum_{s=1}^{q} \big(\tilde{Q}_\mu(i^s,u^s,r)-\beta^s\big)^2. \tag{4.49}$$

In particular, for a given pair (i^s,u^s), the scalar β^s is a sample Q-factor corresponding to (i,u). It is generated by starting at i^s, using u^s at the first stage, and simulating a trajectory of states and controls using μ for a total of N stages, and accumulating the corresponding discounted costs. Thus, β^s is a sample of $Q_\mu^N(i^s,u^s)$, the N-stage Q-factor of μ, given by

$$Q_\mu^N(i,u) = \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j)+\alpha J_\mu^{N-1}(j)\big),$$

where $J_\mu^{N-1}(j)$ is the $(N-1)$-stages cost of μ starting at j. The number of stages N in the sample trajectories may be different for

different samples, and can be either large, or fairly small, and a terminal cost $\alpha^N \hat{J}(i_N)$ may be added as in the model-based case of Section 4.7.2. Again an incremental method may be used to solve the training problem (4.49).

(b) *Approximate policy improvement*: Here we compute the new policy $\tilde{\mu}$ according to

$$\tilde{\mu}(i) \in \arg\min_{u\in U(i)} \tilde{Q}_\mu(i,u,\overline{r}), \qquad i=1,\ldots,n, \tag{4.50}$$

where $\overline{r}$ is the parameter vector obtained from the policy evaluation operation (4.49).

Unfortunately, trajectory reuse is more problematic in Q-factor evaluation than in cost evaluation, because each trajectory generates state-control pairs of the special form $\big(i,\mu(i)\big)$ at every stage after the first, so *pairs* (i,u) *with* $u \neq \mu(i)$ *are not adequately explored*; cf. the discussion in Section 4.7.2. For this reason, it is necessary to make an effort to include in the samples a rich enough set of trajectories that start at pairs (i,u) with $u\neq\mu(i)$.

An important alternative to the preceding procedure is a two-stage process for policy evaluation: first compute in model-free fashion a cost function approximation $\tilde{J}_\mu(j,\overline{r})$, using the regression (4.47), and then use a *second sampling process and regression* to approximate further the (already approximate) Q-factor

$$\sum_{j=1}^n p_{ij}(u)\big(g(i,u,j)+\alpha\tilde{J}_\mu(j,\overline{r})\big),$$

with some $\tilde{Q}_\mu(i,u,\overline{r})$ possibly obtained with a policy approximation architecture (see the discussion of Section 2.1.3 on model-free approximation in policy space). Finally, once $\tilde{Q}_\mu(i,u,\overline{r})$ is obtained with this approximation in policy space, the "improved" policy $\tilde{\mu}$ is obtained from the minimization (4.50). The overall scheme can be viewed as *model-free approximate PI that is based on approximation in both value and policy space*. In view of the two-fold approximation needed to obtain $\tilde{Q}_\mu(i,u,\overline{r})$, this scheme is more complex, but allows trajectory reuse and thus deals better with the exploration issue.

4.7.4 Implementation Issues of Parametric Policy Iteration

Approximate PI in its various forms has been the subject of extensive research, both theoretical and applied. Let us provide a few comments, focusing on the preceding parametric PI schemes.

Architectural Issues and Cost Shaping

The choice of architectures for costs $\tilde{J}_\mu(i, r)$ and Q-factors $\tilde{Q}_\mu(i, u, r)$ is critical for the success of parametric approximation schemes. These architectures may involve the use of features, and they could be linear, or they could be nonlinear such as a neural network. A major advantage of a linear feature-based architecture is that the policy evaluations (4.47) and (4.49) involve linear least squares problems, which admit a closed-form solution. Moreover, when linear architectures are used, there is a broader variety of approximate policy evaluation methods with solid theoretical performance guarantees, such as TD(λ), LSTD(λ), and LSPE(λ), which will be summarized in Section 4.9, and are described in detail in several textbook sources.

Another interesting possibility for architecture choice has to do with *cost shaping*, which we discussed in Section 4.2. This possibility involves a modified cost per stage

$$\hat{g}(i, u, j) = g(i, u, j) + V(j) - V(i), \qquad i = 1, \dots, n,$$

[cf. Eq. (4.11)] for SSP problems, where V can be any approximation to J^*. The corresponding formula for discounted problems is

$$\hat{g}(i, u, j) = g(i, u, j) + \alpha V(j) - V(i), \qquad i = 1, \dots, n.$$

As noted in Section 4.2, cost shaping may change significantly the suboptimal policies produced by approximate DP methods and approximate PI in particular. Generally, V should be chosen close (at least in terms of "shape") to J^* or to the current policy cost function J_{μ^k}, so that the difference $J^* - V$ or $J_{\mu^k} - V$, respectively, can be approximated by an architecture that matches well the characteristics of the problem. It is possible to approximate either V or $\hat{J}$ with a parametric architecture or with a different approximation method, depending on the problem at hand. Moreover, in the context of approximate PI, the choice of V may change from one policy evaluation to the next.

The literature referenced at the end of the chapter provide some applications of cost shaping. An interesting possibility is to use complementary approximations for V and for J^* or J_{μ^k}. For example V may be approximated by a neural network-based approach that aims to discover the general form of J^* or J_{μ^k}, and then a different method may be applied to provide a local correction to V in order to refine the approximation. The next chapter will also illustrate this idea within the context of aggregation.

Exploration Issues

Generating an appropriate set of training pairs (i^s, β^s) or triplets (i^s, u^s, β^s) at the policy evaluation step of approximate PI poses considerable challenges, and the literature contains several related proposals. A generic

difficulty has to do with *inadequate exploration*, which we noted in Section 4.7.2.

In particular, when evaluating a policy μ with trajectory reuse, we will be generating many cost or Q-factor samples that start from states frequently visited by μ, but this may bias the simulation by underrepresenting states that are unlikely to occur under μ. As a result, the cost or Q-factor estimates of these underrepresented states may be highly inaccurate, causing potentially serious errors in the calculation of the improved policy $\overline{\mu}$ via the policy improvement operation.

One possibility to improve the exploration of the state space is to use a large number of initial states to form a rich and representative subset, thereby limiting trajectory reuse. It may then be necessary to use relatively short trajectories to keep the cost of the simulation low. However, when using short trajectories it will be important to introduce a terminal cost function approximation in the policy evaluation step in order to make the cost sample β^s more accurate, as noted earlier.

There have been other related approaches to improve exploration, particularly in connection with the temporal difference methods to be discussed in Section 4.9. In some of these approaches, trajectories are generated through a mix of two policies: the policy being evaluated, sometimes called the *target policy*, to distinguish from the other policy, used with some probability at each stage, which is called *behavior policy* and is introduced to enhance exploration; see the end-of-chapter references. Also, methods that use a behavior policy are called *off-policy* methods, while methods that do not are called *on-policy* methods. Note, however, that it may still be difficult to ensure that the mixed on-and-off policy will induce sufficient exploration. The area of efficient sampling, and the attendant issue of balancing exploration and the choice of promising controls (the so-called exploration-exploitation tradeoff) is a subject continuing research; for some recent work, see the paper by Russo and Van Roy [RuV16], and the monograph [RVK18].

Oscillation Issues

Contrary to exact PI, which is guaranteed to yield an optimal policy, approximate PI produces a sequence of policies, which are only guaranteed to lie asymptotically within a certain error bound from the optimal; cf. Prop. 4.6.4. Moreover, the generated policies may oscillate. By this we mean that after a few iterations, policies tend to repeat in cycles.

This oscillation phenomenon, first described by the author in a 1996 conference [Ber96], occurs systematically in the absence of special conditions, for both optimistic and nonoptimistic PI methods. It can be observed even in very simple examples, and it is geometrically explained in the books [BeT96] (Section 6.4.2) and [Ber12] (Section 6.4.3).

Oscillations can in principle be particularly damaging, because there is no guarantee that the oscillating policies are "good" policies, and there is often no way to verify how well they perform relative to the optimal. Section 6.4.2 of the book [BeT96] provides an argument suggesting that oscillations may not degrade significantly the approximate PI performance for many types of problems. Moreover, we note that oscillations can be avoided and approximate PI can be shown to converge under special conditions, which arise in particular when an aggregation approach is used; see Chapter 5 and the approximate policy iteration survey [Ber11a]. Also, when policies converge, there is a more favorable error bound, cf. Prop. 4.6.5.

4.8 Q-LEARNING

In this section we will discuss various Q-learning algorithms for discounted problems, which can be implemented in model-free fashion. The original method of this type is related to VI and can be used directly in the case of multiple policies. Instead of approximating the cost functions of successive policies as in the PI method, it updates the Q-factors associated with an *optimal* policy, thereby avoiding the multiple policy evaluation steps of PI. We will consider Q-learning as well as a variety of related methods with the shared characteristic that they involve exact or approximate Q-factors.

We first discuss the original form of Q-learning for discounted problems; the books [BeT96] and [Ber12] contain discussions of Q-learning for SSP problems. Then we discuss PI algorithms for Q-factors, including optimistic asynchronous versions, which lead to algorithms with reduced overhead per iteration. Finally we focus on Q-learning algorithms with Q-factor approximation.

Q-Learning: A Stochastic VI Algorithm

In the discounted problem, the optimal Q-factors are defined for all pairs (i,u) with $u \in U(i)$, by

$$Q^*(i,u) = \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha J^*(j)\big).$$

As discussed in Section 4.3, these Q-factors satisfy for all (i,u),

$$Q^*(i,u) = \sum_{j=1}^{n} p_{ij}(u)\left(g(i,u,j) + \alpha \min_{v \in U(j)} Q^*(j,v)\right),$$

and are the unique solution of this set of equations. Moreover the optimal Q-factors can be obtained by the VI algorithm $Q_{k+1} = FQ_k$, where F is

the operator defined by

$$(FQ)(i,u) = \sum_{j=1}^{n} p_{ij}(u) \left(g(i,u,j) + \alpha \min_{v \in U(j)} Q(j,v) \right), \quad \text{for all } (i,u). \tag{4.51}$$

It is straightforward to show that F is a contraction with modulus α, similar to the DP operator T. Thus the algorithm $Q_{k+1} = FQ_k$ converges to Q^* from every starting point Q_0.

The original and most widely known Q-learning algorithm ([Wat89]) is a stochastic version of VI, whereby the expected value in Eq. (4.51) is suitably approximated by sampling and simulation. In particular, an infinitely long sequence of state-control pairs $\{(i_k, u_k)\}$ is generated according to some probabilistic mechanism. For each pair (i_k, u_k), a state j_k is generated according to the probabilities $p_{i_k j}(u_k)$. Then the Q-factor of (i_k, u_k) is updated using a stepsize $\gamma^k \in (0, 1]$ while all other Q-factors are left unchanged:

$$Q_{k+1}(i,u) = (1 - \gamma^k) Q_k(i,u) + \gamma^k (F_k Q_k)(i,u), \qquad \text{for all } (i,u), \tag{4.52}$$

where

$$(F_k Q_k)(i,u) = \begin{cases} g(i_k, u_k, j_k) + \alpha \min_{v \in U(j_k)} Q_k(j_k, v) & \text{if } (i,u) = (i_k, u_k), \\ Q_k(i,u) & \text{if } (i,u) \neq (i_k, u_k). \end{cases} \tag{4.53}$$

Note that $(F_k Q_k)(i_k, u_k)$ is a single sample approximation of the expected value defining $(FQ_k)(i_k, u_k)$ in Eq. (4.51).

To guarantee the convergence of the algorithm (4.52)-(4.53) to the optimal Q-factors, some conditions must be satisfied. Chief among these are that all state-control pairs (i, u) must be generated infinitely often within the infinitely long sequence $\{(i_k, u_k)\}$, and that the successor states j must be independently sampled at each occurrence of a given state-control pair. Furthermore, the stepsize γ^k should satisfy

$$\gamma^k > 0, \quad \text{for all } k, \qquad \sum_{k=0}^{\infty} \gamma^k = \infty, \quad \sum_{k=0}^{\infty} (\gamma^k)^2 < \infty,$$

which are typical of stochastic approximation methods (see e.g, the books [BeT96], [Ber12], Section 6.1.4), as for example when $\gamma^k = c_1/(k + c_2)$, where c_1 and c_2 are some positive constants. In addition some other technical conditions should hold. A mathematically rigorous convergence proof was given in the paper [Tsi94], which embeds Q-learning within a broad class of asynchronous stochastic approximation algorithms. This proof (also reproduced in [BeT96]) combines the theory of stochastic approximation algorithms with the convergence theory of asynchronous DP

and asynchronous iterative methods; cf. the paper [Ber82], and the books [BeT89] and [Ber16].

In practice, Q-learning has some drawbacks, the most important of which is that the number of Q-factors/state-control pairs (i, u) may be excessive. To alleviate this difficulty, we may introduce a Q-factor approximation architecture, which could be linear or nonlinear based for example on a neural network. One of these possibilities will be discussed next.

Optimistic Policy Iteration Methods with Q-Factor Approximation - SARSA

We have discussed so far Q-learning algorithms with an exact representation of Q-factors. We will now consider Q-learning with linear feature-based Q-factor approximation. As we noted earlier, we may view Q-factors as optimal costs of a certain discounted DP problem, whose states are the state-control pairs (i, u) in addition to the original states; cf. Fig. 4.2.2. We may thus apply the approximate PI methods discussed earlier. For this, we need to introduce a linear parametric architecture $\tilde{Q}(i, u, r)$,

$$\tilde{Q}(i, u, r) = \phi(i, u)'r,$$

where $\phi(i, u)$ is a feature vector that depends on both state and control.

We have already discussed in Section 4.7.3 a model-free approximate PI method that is based on Q-factors and least squares training/regression. There are also optimistic approximate PI methods, which use a policy for a limited number of stages with cost function approximation for the remaining states, and/or a few samples in between policy updates. As an example, let us consider a Q-learning algorithm that uses a single sample between policy updates. At the start of iteration k, we have the current parameter vector r^k, we are at some state i^k, and we have chosen a control u^k. Then:

(1) We simulate the next transition (i^k, i^{k+1}) using the transition probabilities $p_{i^k j}(u^k)$.

(2) We generate the control u^{k+1} with the minimization

$$u^{k+1} \in \arg \min_{u \in U(i^{k+1})} \tilde{Q}(i^{k+1}, u, r^k).$$

[In some schemes, u^{k+1} is chosen with a small probability to be a different or random element of $U(i^{k+1})$ in order to enhance exploration.]

(3) We update the parameter vector via

$$r^{k+1} = r^k - \gamma^k \phi(i^k, u^k) q_k,$$

where γ^k is a positive stepsize, and q_k is given by

$$q_k = \phi(i^k, u^k)'r^k - \alpha\phi(i^{k+1}, u^{k+1})'r^k - g(i^k, u^k, i^{k+1}).$$

The vector $\phi(i^k, u^k)q_k$ can be interpreted as an approximate gradient direction based on an underlying regression procedure, and q_k is referred to as a temporal difference (cf. Section 4.9).

The process is now repeated with r^{k+1}, i^{k+1}, and u^{k+1} replacing r^k, i^k, and u^k, respectively.

Extreme optimistic schemes of the type just described, including variants that involve nonlinear architectures, have been used in practice, and are often referred to as SARSA (State-Action-Reward-State-Action); see e.g., the books [BeT96], [BBD10], [SuB18]. When Q-factor approximation is used, their behavior is very complex, their theoretical convergence properties are unclear, and there are no associated performance bounds in the literature.

We finally note that in simulation-based PI methods for Q-factors, a major concern is the issue of exploration in the approximate evaluation step of the current policy μ, to ensure that state-control pairs $(i,u) \neq \big(i, \mu(i)\big)$ are generated sufficiently often in the simulation.

4.9 ADDITIONAL METHODS - TEMPORAL DIFFERENCES

In this section, we summarize a few additional methods for approximation in value space in infinite horizon problems. These include the simulation-based temporal difference methods for policy evaluation with a linear parametric architecture, whose primary aim is to address a bias-variance tradeoff similar to the one discussed in Section 4.7.2. Our presentation is brief, somewhat abstract, and makes use of linear algebra mathematics. It may be skipped without loss of continuity. This is only a summary; it is meant to provide a connection to other material in this chapter, and orientation for further reading into both the optimization and artificial intelligence literature on the subject.

Approximate Policy Evaluation Using Projections

Our main concern in policy evaluation is to solve approximately the Bellman equation corresponding to a given policy μ. Thus, for discounted problems, we are interested in solving the linear system of equations

$$J_\mu(i) = \sum_{i=1}^{n} p_{ij}\big(\mu(i)\big)\Big(g\big(i,\mu(i),j\big) + \alpha J_\mu(j)\Big), \qquad i = 1,\ldots,n,$$

or in shorthand,

$$J_\mu = T_\mu J_\mu, \tag{4.54}$$

where T_μ is the DP operator for μ, given by

$$(T_\mu J)(i) = \sum_{i=1}^{n} p_{ij}\big(\mu(i)\big)\Big(g\big(i,\mu(i),j\big) + \alpha J(j)\Big), \qquad i = 1,\ldots,n. \tag{4.55}$$

Let us consider the approximate solution of this equation by parametric approximation (cf. Section 4.7). This amounts to replacing J_μ with some vector that lies within the manifold represented by the approximation architecture

$$\mathcal{M} = \Big\{ \big(\tilde{J}(1,r),\ldots,\tilde{J}(n,r)\big) \mid \text{all parameter vectors } r \Big\}. \tag{4.56}$$

The approximate solution of systems of equations within an approximation manifold of the form (4.56) has a long history in scientific computation, particularly when the manifold is linear. A central approach involves the use of projections with respect to a weighted quadratic norm

$$\|J\|^2 = \sum_{i=1}^{n} \xi_i \big(J(i)\big)^2, \tag{4.57}$$

where $J(i)$ are the components of the vector J and ξ_i are some positive weights. The projection of a vector J onto the manifold $\mathcal{M}$ is denoted by $\Pi(J)$. Thus

$$\Pi(J) \in \arg\min_{V\in\mathcal{M}} \|J - V\|^2. \tag{4.58}$$

Note that for a nonlinear parametric architecture, such as a neural network, the projection may not exist and may not be unique. However, in the case of a linear architecture, where the approximation manifold $\mathcal{M}$ is a subspace, the projection does exist and is unique; this is a consequence of the fundamental orthogonal projection theorem of calculus and real analysis.

Let us consider three general approaches for approximation of J_μ.

(a) Project J_μ onto $\mathcal{M}$ to obtain $\Pi(J_\mu)$, which is used as an approximation of J_μ.

(b) Start with some approximation $\hat{J}$ of J_μ, perform N value iterations to obtain $T_\mu^N \hat{J}$, and project onto $\mathcal{M}$ to obtain $\Pi(T_\mu^N \hat{J})$. We then use $\Pi(T_\mu^N \hat{J})$ as an approximation to J_μ.

(c) Solve a projected version $J_\mu = \Pi(T_\mu J_\mu)$ of the Bellman Eq. (4.54), and use the solution of this projected equation as an approximation to J_μ. We will also discuss related projected versions that involve other operators in place of T_μ.

The preceding three approaches cannot be implemented exactly; for example, (a) is impossible since we do not know the values of J_μ. However, it turns out that it is possible to implement these approaches by using a Monte Carlo simulation methodology that is suitable for large problems. To explain this methodology we first discuss the implementation of the projection operation through sampling for the case where the parametric architecture is linear and $\mathcal{M}$ is a subspace.

Projection by Monte Carlo Simulation

We will focus on the case where the manifold $\mathcal{M}$ is a subspace of the form

$$\mathcal{M} = \{\Phi r \mid r \in \Re^m\}, \tag{4.59}$$

where $\Re^m$ denote the space of m-dimensional vectors, and Φ is an $n \times m$ matrix with rows denoted by $\phi(i)'$, $i = 1, \ldots, n$. Here we use the notational convention that all vectors are column vectors, and prime denotes transposition, so $\phi(i)'$ is an m-dimensional row vector, and the subspace $\mathcal{M}$ may be viewed as the space spanned by the n-dimensional columns of Φ.

We consider projection with respect to the weighted Euclidean norm of Eq. (4.57), so $\Pi(J)$ is of the form Φr^*, where

$$r^* \in \arg\min_{r\in\Re^m} \|\Phi r - J\|_\xi^2 = \arg\min_{r\in\Re^m} \sum_{i=1}^n \xi_i \big(\phi(i)'r - J(i)\big)^2. \tag{4.60}$$

By setting to 0 the gradient at r^* of the minimized expression above,

$$2\sum_{i=1}^n \xi_i \phi(i)\big(\phi(i)'r^* - J(i)\big) = 0,$$

we obtain the solution in closed form,

$$r^* = \left(\sum_{i=1}^n \xi_i \phi(i)\phi(i)'\right)^{-1} \sum_{i=1}^n \xi_i \phi(i) J(i), \tag{4.61}$$

assuming that the inverse above exists. The difficulty here is that when n is very large, the matrix-vector calculations in this formula can be very time-consuming.

On the other hand, assuming (by normalizing ξ if necessary) that $\xi = (\xi_1, \ldots, \xi_n)$ is a probability distribution, we may view the two terms in Eq. (4.61) as expected values with respect to ξ, and approximate them by Monte Carlo simulation. In particular, suppose that we generate a set of index samples i^s, $s = 1, \ldots, q$, according to the distribution ξ, and form the Monte Carlo estimates

$$\frac{1}{q}\sum_{s=1}^q \phi(i^s)\phi(i^s)' \approx \sum_{i=1}^n \xi_i \phi(i)\phi(i)', \qquad \frac{1}{q}\sum_{s=1}^q \phi(i^s)\beta^s \approx \sum_{i=1}^n \xi_i \phi(i) J(i), \tag{4.62}$$

where β^s is a "noisy" sample of the exact value $J(i^s)$

$$\beta^s = J(i^s) + n(i^s).$$

For the Monte Carlo estimates (4.62) to be asymptotically correct, we must have

$$\frac{1}{q}\sum_{s=1}^{q}\phi(i^s)n(i^s)\approx 0, \tag{4.63}$$

which is implied by a zero sample mean condition for the noise.†

Given the Monte Carlo approximation of the two terms in Eq. (4.61), we can estimate r^* with

$$\overline{r}=\left(\sum_{s=1}^{q}\phi(i^s)\phi(i^s)'\right)^{-1}\sum_{s=1}^{q}\phi(i^s)\beta^s, \tag{4.65}$$

(assuming sufficiently many samples are obtained to ensure the existence of the inverse above).‡ This is also equivalent to estimating r^* by approximating the least squares minimization (4.60) with the following least squares training problem

$$\overline{r}\in\arg\min_{r\in\Re^m}\sum_{s=1}^{q}\big(\phi(i^s)'r-\beta^s\big)^2. \tag{4.66}$$

Thus simulation-based projection can be implemented in two equivalent ways:

---

† A suitable zero mean condition for the noise $n(i^s)$ has the form

$$\lim_{q\to\infty}\frac{\sum_{s=1}^{q}\delta(i^s=i)n(i^s)}{\sum_{s=1}^{q}\delta(i^s=i)}=0, \qquad \text{for all } i=1,\ldots,n, \tag{4.64}$$

where $\delta(i^s=i)=1$ if $i^s=i$ and $\delta(i^s=i)=0$ if $i^s\neq i$. It states that the Monte Carlo averages of the noise terms corresponding to every state i are zero. Then the expression in Eq. (4.63) has the form

$$\begin{aligned}\frac{1}{q}\sum_{s=1}^{q}\phi(i^s)n(i^s)&=\frac{1}{q}\sum_{i=1}^{n}\phi(i)\sum_{s=1}^{q}\delta(i^s=i)n(i^s)\\&=\frac{1}{q}\sum_{i=1}^{n}\phi(i)\sum_{s'=1}^{q}\delta(i^{s'}=i)\frac{\sum_{s=1}^{q}\delta(i^s=i)n(i^s)}{\sum_{s=1}^{q}\delta(i^s=i)},\end{aligned}$$

and converges to 0 as $q\to\infty$, assuming that each index i is sampled infinitely often so that Eq. (4.64) can be used.

‡ The preceding derivation and the formula (4.65) actually make sense even if $\xi=(\xi_1,\ldots,\xi_n)$ has some zero components, as long as the inverses in Eqs. (4.61) and (4.65) exist. This is related to the concept of *seminorm projection*; see [YuB12] for an approximate DP-related discussion.

(a) Replacing expected values in the exact projection formula (4.61) by simulation-based estimates [cf. Eq. (4.65)].

(b) Replacing the exact least squares problem (4.60) with a simulation-based least squares approximation [cf. Eq. (4.66)].

These dual possibilities of implementing projection by simulation can be used interchangeably. In particular, *the least squares training problems considered in this book may be viewed as simulation-based approximate projection calculations*.

Generally, we wish that the estimate $\overline{r}$ converges to r^* as the number of samples q increases. An important point is that it is not necessary that the simulation produces independent samples. Instead it is sufficient that the long term empirical frequencies by which the indices i appear in the simulation sequence are consistent with the probabilities ξ_i of the projection norm, i.e.,

$$\xi_i = \lim_{k\to\infty} \frac{1}{q}\sum_{s=1}^{q} \delta(i^s = i), \qquad i = 1,\ldots,n, \tag{4.67}$$

where $\delta(i^s = i) = 1$ if $i^s = i$ and $\delta(i^s = i) = 0$ if $i^s \neq i$.

Another important point is that the probabilities ξ_i need not be predetermined. In fact, often the exact values of ξ_i do not matter much, and *one may wish to first specify a reasonable and convenient sampling scheme, and let ξ_i be implicitly specified via Eq. (4.67).*

Projected Equation View of Approximate Policy Evaluation

Let us now discuss the approximate policy evaluation method for costs of Section 4.7.2 [cf. Eq. (4.47)]. It can be interpreted in terms of a projected equation, written abstractly as

$$\tilde J_\mu \approx \Pi(T_\mu^N \hat J), \tag{4.68}$$

where:†

(a) $\hat J$ is some initial guess of J_μ (the terminal cost function approximation discussed in Section 4.7.2), and $\tilde J_\mu$ is the vector

$$\tilde J_\mu = \big(\tilde J(1,\overline{r}),\ldots,\tilde J(n,\overline{r})\big),$$

which is the approximate policy evaluation of μ, used in the policy improvement operation (4.48). Here $\overline{r}$ is the solution of the training problem (4.47).

† The equation (4.68) assumes that all trajectories have equal length N, and thus does not allow trajectory reuse. If trajectories of different lengths are allowed, the term T_μ^N in the equation should be replaced by a more complicated weighted sum of powers of T_μ; see the paper [YuB12] for related ideas.

(b) T_μ is the DP operator corresponding to μ, which maps a vector $J = \big(J(1), \ldots, J(n)\big)$ into the vector $T_\mu J$ of Eq. (4.55).

(c) T_μ^N denotes the N-fold application of the operator T_μ, where N is the length of the sample trajectories used in the least squares regression problem (4.47). In particular, $(T_\mu^N \hat{J})(i)$ is the cost associated with starting at i, using μ for N stages, and incurring a terminal cost specified by the terminal cost function $\hat{J}$. The sample state-cost pairs (i^s, β^s) are obtained from trajectories corresponding to this N-stage problem.

(d) $\Pi(T_\mu^N \hat{J})$ denotes projection of the vector $T_\mu^N \hat{J}$ on the manifold of possible approximating vectors $\mathcal{M}$ with respect to a weighted norm, where each weight ξ_i represents the relative frequency of the state i as initial state of a training trajectory. This projection is approximated by the least squares regression (4.47). In particular, the cost samples β^s of the training set are noisy samples of the values $(T_\mu^N \hat{J})(i^s)$, and the projection is approximated with a least squares minimization, to yield the function $\tilde{J}_\mu$ of Eq. (4.68).

Suppose now that $T_\mu^N \hat{J}$ is close to J_μ (which happens if either N is large or $\hat{J}$ is close to J_μ, or both) and the number of samples q is large (so that the simulation-based regression approximates well the projection operation Π). Then from Eq. (4.68), the approximate evaluation $\tilde{J}_\mu$ of μ approaches the projection of J_μ on the approximation manifold (4.56), which can be viewed as the best possible approximation of J_μ (at least relative to the distance metric defined by the weighted projection norm). This provides an abstract formal rationale for the parametric PI method of Section 4.7.2, which is based on Eq. (4.68).

TD(λ), LSTD(λ), and LSPE(λ)

Projected equations also fundamentally underlie *temporal difference methods* (TD for short), a prominent class of simulation-based methods for approximate evaluation of a policy. Examples of such methods are TD(λ), LSTD(λ), and LSPE(λ), where λ is a scalar with $0 \leq \lambda < 1$.†

These three methods require a linear parametric approximation architecture $\tilde{J}_\mu = \Phi r$, and all aim at the same problem. This is the problem of solving a projected equation of the form

$$\Phi r = \Pi\big(T_\mu^{(\lambda)} \Phi r\big), \tag{4.69}$$

† TD stands for "temporal difference," LSTD stands for "least squares temporal difference," and LSPE stands for "least squares policy evaluation."

where T_μ is the operator (4.55), $T_\mu^{(\lambda)}J$ is defined by

$$(T_\mu^{(\lambda)}J)(i) = (1-\lambda)\sum_{\ell=0}^{\infty}\lambda^\ell (T_\mu^{\ell+1}J)(i), \qquad i = 1,\ldots,n,$$

and Π is projection on the approximation subspace

$$\mathcal{M} = \{\Phi r \mid r \in \Re^m\},$$

with respect to some weighted projection norm. One interpretation of the equation $J = T_\mu^{(\lambda)}J$ is as a *multistep version of Bellman's equation*. It has the same solution, J_μ, as the "one-step" Bellman equation $J = T_\mu J$, which corresponds to $\lambda = 0$.

Of course the projected equation (4.69) cannot be solved exactly when the number of states n is large, since the projection is a high dimensional operation that requires computations of order n. Instead the key idea is to *replace the projection by a simulation-based approximate projection*, of the type discussed earlier. This yields the equation,

$$\Phi r = \tilde{\Pi}\big(T_\mu^{(\lambda)}\Phi r\big), \tag{4.70}$$

where $\tilde{\Pi}$ is the approximate projection obtained by sampling.

For a more concrete description, let the ith row of the matrix Φ be the m-dimensional row vector $\phi(i)'$, so that the cost $J_\mu(i)$ is approximated as the inner product $\phi(i)'r$:

$$J_\mu(i) \approx \phi(i)'r.$$

Suppose that we collect q samples of initial states i^s, $s = 1,\ldots,q$, together with the corresponding transition costs $g(i^s, i^{s+1})$, $s = 1,\ldots,q$. Then the parameter vector $\overline{r}$ that solves Eq. (4.70) satisfies

$$\overline{r} \in \arg\min_r \sum_{s=1}^{q}\big(\phi(i^s)'r - \text{sample of } (T_\mu^{(\lambda)}\Phi\overline{r})(i^s)\big)^2, \tag{4.71}$$

[cf. Eq. (4.66)], and defines the approximate evaluation $\Phi\overline{r}$ of J_μ. This relation can be expressed as a linear equation, which in principle can be solved in closed form [cf. Eq. (4.65)], and indeed LSTD(λ) does exactly that. By contrast LSPE(λ) and TD(λ) solve this relation iteratively.

We will first give a high level description of the three methods, and then provide a more concrete description for the simpler case where $\lambda = 0$.

(a) The LSTD(λ) method, after the q samples have been collected, solves the relation (4.71) by matrix inversion, taking advantage of the fact

that this relation can be written as a linear equation. In particular, it can be written as

$$C\overline{r} = d, \tag{4.72}$$

where C is some $m \times m$ square matrix, and d is an m-dimensional vector. The components of C and d are explicitly computed, and LSTD(λ) produces the approximate cost function $\tilde{J}_\mu(i) = \Phi\overline{r}$ where $\overline{r} = C^{-1}d$ is the solution of Eq. (4.72).

(b) The LSPE(λ) method solves the projected equation (4.69) by using a simulation-based *projected value iteration*,

$$J_{k+1} = \tilde{\Pi}\big(T_\mu^{(\lambda)} J_k\big). \tag{4.73}$$

Here the projection is implemented iteratively, with sampling-based least squares regression, in a manner that resembles the incremental aggregated method of Section 3.1.3.

(c) The TD(λ) method is a simpler iterative stochastic approximation method for solving the linear equation (4.72). It can also be viewed as a stochastic gradient method, or as a stochastic version of the proximal algorithm for solving this linear equation (see the author's papers [Ber16c] and [Ber18d]).

An interesting question is how to select λ and what is its role. There is a bias-variance tradeoff here, similar to the one we discussed in Section 4.7.2. We will address this issue later in this section.

TD(0), LSTD(0), and LSPE(0)

Let us describe in more detail LSTD(0) for evaluation of a given policy μ. We assume that the simulation generates a sample sequence of q transitions using μ:

$$(i^1, j^1), (i^2, j^2), \ldots, (i^q, j^q),$$

with corresponding transition costs

$$g(i^1, j^1), g(i^2, j^2), \ldots, g(i^q, j^q).$$

Here, to simplify notation, we do not show the dependence of the transition costs on the control applied by μ. Let the ith row of the matrix Φ be the m-dimensional row vector $\phi(i)'$, so that the cost $J_\mu(i)$ is approximated as the inner product $\phi(i)'r$:

$$J_\mu(i) \approx \phi(i)'r.$$

Since $\lambda = 0$, we have $T^{(\lambda)} = T$, the samples of $T_\mu\Phi\overline{r}$ in Eq. (4.71) are

$$g(i^s, i^{s+1}) + \alpha\phi(i^{s+1})'\overline{r},$$

and the least squares problem in Eq. (4.71) has the form

$$\min_r \sum_{s=1}^{q} \big(\phi(i^s)'r - g(i^s, i^{s+1}) - \alpha\phi(i^{s+1})'\overline{r}\big)^2. \tag{4.74}$$

By setting the gradient of the minimized expression to zero, we obtain the condition for $\overline{r}$ to attain the above minimum:

$$\sum_{s=1}^{q} \phi(i^s)\big(\phi(i^s)'\overline{r} - g(i^s, i^{s+1}) - \alpha\phi(i^{s+1})'\overline{r}\big) = 0. \tag{4.75}$$

Solving this equation for $\overline{r}$ yields the LSTD(0) solution:

$$\overline{r} = \left(\sum_{s=1}^{q} \phi(i^s)\big(\phi(i^s) - \alpha\phi(i^{s+1})\big)'\right)^{-1} \sum_{s=1}^{q} \phi(i^s) g(i^s, i^{s+1}). \tag{4.76}$$

Note that the inverse in the preceding equation must exist for the method to be well-defined; otherwise the iteration has to be modified. A modification may also be needed when the matrix inverted is nearly singular; in this case the simulation noise may introduce serious numerical problems. Various methods have been developed to deal with the near singularity issue; see Wang and Bertsekas [WaB13a], [WaB13b], and the DP textbook [Ber12], Section 7.3.

The expression

$$d^s(\overline{r}) = \phi(i^s)'\overline{r} - g(i^s, i^{s+1}) - \alpha\phi(i^s)'\overline{r} \tag{4.77}$$

that appears in the least squares sum minimization (4.74) and Eq. (4.75) is referred to as the *temporal difference associated with the sth transition and parameter vector* $\overline{r}$. In the artificial intelligence literature, temporal differences are viewed as fundamental to learning and are accordingly interpreted, but we will not go further in this direction; see the RL textbooks that we have cited.

The LSPE(0) method is similarly derived. It consists of a simulation-based approximation of the projected value iteration method

$$J_{k+1} = \tilde{\Pi}\big(T_\mu J_k\big),$$

[cf. Eq. (4.73)]. At the kth iteration, it uses only the samples $s = 1, \ldots, k$, and updates the parameter vector according to

$$r^{k+1} = r^k - \left(\sum_{s=1}^{k} \phi(i^s)\phi(i^s)'\right)^{-1} \sum_{s=1}^{k} \phi(i^s) d^s(r^s), \qquad k = 1, 2, \ldots, \tag{4.78}$$

where $d^s(r^s)$ is the temporal difference of Eq. (4.77), evaluated at the iterate of iteration s; the form of this iteration is derived similar to the case of LSTD(0). After q iterations, when all the samples have been processed, the vector r^q obtained is the one used for the approximate evaluation of J_μ. Note that the inverse in Eq. (4.78) can be updated economically from one iteration to the next, using fast linear algebra operations (cf. the discussion of the incremental Newton method in Section 3.1.3).

Overall, it can be shown that LSTD(0) and LSPE(0) [with efficient matrix inversion in Eq. (4.78)] require essentially identical amount of work to process the q samples associated with the current policy μ [this is also true for the LSTD(λ) and LSPE(λ) methods; see [Ber12], Section 6.3]. An advantage offered by LSPE(0) is that because it is iterative, it allows carrying over the final parameter vector r^q, as a "hot start" when passing from one policy evaluation to the next, in the context of an approximate PI scheme.

The TD(0) method has the form

$$r^{k+1} = r^k - \gamma^k \phi(i^k) d^k(r^k), \qquad k = 1, 2, \ldots, \tag{4.79}$$

where γ^k is a diminishing stepsize parameter. It can be seen that TD(0) resembles an *incremental gradient* iteration for solving the least squares training problem (4.74), but with $\overline{r}$ replaced by the current iterate r^k. The reason is that the gradient of the typical kth term in the least squares sum of Eq. (4.74) is the vector $\phi(i^k)d^k(r^k)$ that appears in the TD(0) iteration (4.79) (cf. Section 3.1.3). Thus at each iteration, TD(0) uses only one sample, and changes r^k in the opposite direction to the corresponding incremental gradient using a stepsize γ^k that must be carefully controlled.

By contrast the LSPE(0) iteration (4.78) uses the full sum

$$\sum_{s=1}^{k} \phi(i^s) d^s(r^s),$$

which may be viewed as an *aggregated incremental* method, with scaling provided by the matrix $\left(\sum_{s=1}^{k} \phi(i^s)\phi(i^s)'\right)^{-1}$. This explains why TD(0) is generally much slower and more fragile than LSPE(0). On the other hand TD(0) is simpler than both LSTD(0) and LSPE(0), and does not require a matrix inversion, which may be inconvenient when the column dimension m of Φ is large.

The properties, the analysis, and the implementation of TD methods in the context of approximate PI are quite complicated. In particular, the issue of exploration is important and must be addressed. Moreover there are convergence, oscillation, and reliability issues to contend with. LSTD(λ) relies on matrix inversion and not on iteration, so it does not have a serious convergence issue, but the system (4.72) may be singular or near

singular, in which case very accurate simulation is needed to approximate C well enough for its inversion to be reliable; remedies for the case of a singular or near singular system are discussed in the papers [WaB13a], [WaB13b] (see also [Ber12], Section 7.3). LSPE(λ) has a convergence issue because the mapping $\Pi T_\mu^{(\lambda)}$ may not be a contraction mapping (even though T_μ is) and the projected value iteration (4.73) may not be convergent (it turns out that the mapping $\Pi T_\mu^{(\lambda)}$ is guaranteed to be a contraction for λ sufficiently close to 1).

Direct and Indirect Policy Evaluation Methods

In trying to compare the approximate policy evaluation methods discussed in this section, we may draw a distinction between *direct methods*, which aim to compute approximately the projection $\Pi(J_\mu)$, and *indirect methods*, which try to solve the projected equation (4.69).

The method of Section 4.7.2 is direct and is based on Eq. (4.68). In particular, as $N \to \infty$ and $q \to \infty$, it yields the approximate evaluation $\Pi(J_\mu)$. The TD methods are indirect, and aim at computing the solution of the projected equation (4.69). The solution of this equation is of the form Φr_λ^*, where the parameter vector r_λ^* depends on λ. In particular the projected equation solution Φr_λ^* is different from $\Pi(J_\mu)$. It can be shown that it satisfies the error bound

$$\|J_\mu - \Phi r_\lambda^*\|_\xi \leq \frac{1}{\sqrt{1-\alpha_\lambda^2}}\, \|J_\mu - \Pi J_\mu\|_\xi, \tag{4.80}$$

where

$$\alpha_\lambda = \frac{\alpha(1-\lambda)}{1-\alpha\lambda}$$

and $\|\cdot\|_\xi$ is a special projection norm of the form (4.57), where ξ is the steady-state probability distribution of the controlled system Markov chain under policy μ. Moreover as $\lambda \to 1$ the projected equation solution Φr_λ^* approaches $\Pi(J_\mu)$. Based on this fact, methods which aim to compute $\Pi(J_\mu)$, such as the direct method of Section 4.7.2 are sometimes called TD(1). We refer to [Ber12], Section 6.3, for an account of this analysis, which is beyond the scope of this book.

The difference $\Pi(J_\mu) - \Phi r_\lambda^*$ is commonly referred to as the *bias* and is illustrated in Figure 4.9.1. As indicated in this figure and as the estimate (4.80) suggests, there is a *bias-variance tradeoff*. As λ is decreased, the solution of the projected equation (4.69) changes and more bias is introduced relative to the "ideal" approximation ΠJ_μ (this bias can be embarassingly large as shown by examples in the paper [Ber95]). At the same time, however, the simulation samples of $T_\mu^{(\lambda)} J$ contain less noise as λ is decreased. This provides another view of the bias-variance tradeoff, which we discussed in Section 4.7.2 in connection with the use of short trajectories.

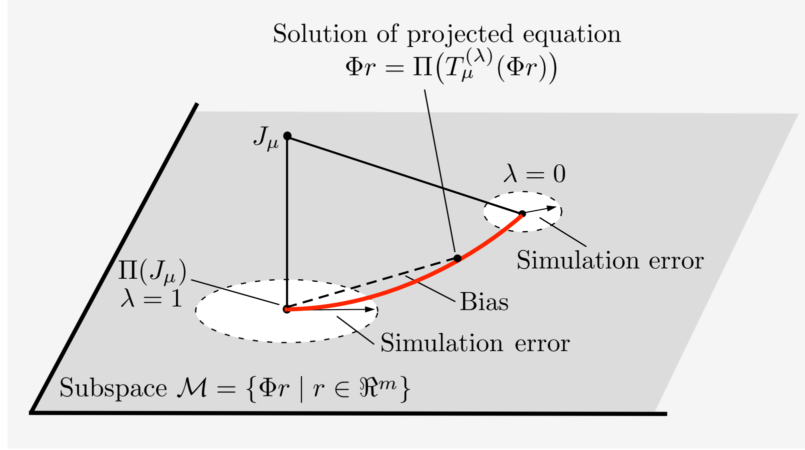

Figure 4.9.1 Illustration of the bias-variance tradeoff in estimating the solution of the projected equation for different values of λ. As λ increases from $\lambda = 0$ towards $\lambda = 1$, the solution Φr_λ^* of the projected equation $\Phi r = \Pi T^{(\lambda)}(\Phi r)$ approaches the projection ΠJ_μ. The difference $\Phi r_\lambda^* - \Pi J_\mu$ is the bias, and it decreases to 0 as λ approaches 1, while the simulation error variance increases.

4.10 EXACT AND APPROXIMATE LINEAR PROGRAMMING

Another method for exact solution of infinite horizon DP problems is based on linear programming ideas. In particular, J^* can be shown to be the unique optimal solution of a certain linear program. Focusing on α-discounted problems, the key idea is that J^* is the "largest" (on a component-by-component basis) vector J that satisfies the constraint

$$J(i) \le \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha J(j)\big), \qquad \text{for all } i = 1,\ldots,n \text{ and } u \in U(i), \tag{4.81}$$

so that $J^*(1),\ldots,J^*(n)$ solve the linear program

$$\begin{aligned} &\text{maximize} \quad \sum_{i=1}^{n} J(i) \\ &\text{subject to the constraint (4.81),} \end{aligned} \tag{4.82}$$

(see Fig. 4.10.1).

To verify this, let us use the VI algorithm to generate a sequence of vectors $J_k = \big(J_k(1),\ldots,J_k(n)\big)$ starting with an initial condition vector $J_0 = \big(J(1),\ldots,J(n)\big)$ such that

$$J_0(i) \le \min_{u\in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha J_0(j)\big) = J_1(i), \qquad \text{for all } i.$$

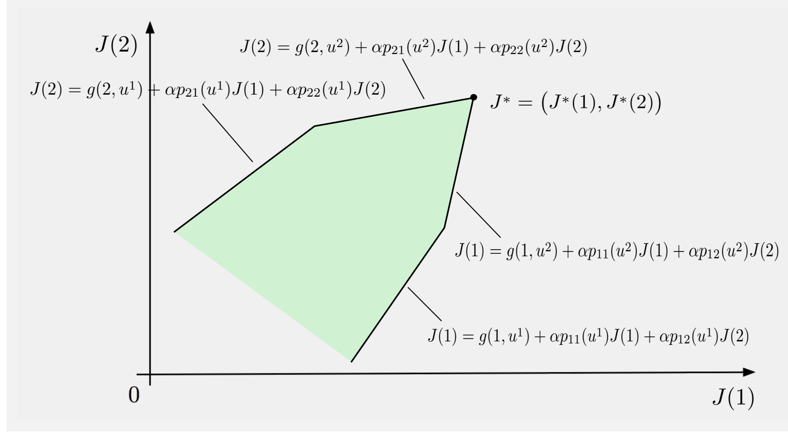

Figure 4.10.1 A linear program associated with a two-state SSP problem. The constraint set is shaded, and the objective to maximize is $J(1) + J(2)$. Note that because we have $J(i) \leq J^*(i)$ for all i and vectors J in the constraint set, the vector J^* maximizes any linear cost function of the form $\sum_{i=1}^n \beta_i J(i)$, where $\beta_i \geq 0$ for all i. If $\beta_i > 0$ for all i, then J^* is the unique optimal solution of the corresponding linear program.

This inequality can be used to show that

$$J_0(i) \leq J_1(i) \leq \min_{u \in U(i)} \sum_{j=1}^n p_{ij}(u)\big(g(i,u,j) + \alpha J_1(j)\big) = J_2(i), \qquad \text{for all } i,$$

and similarly

$$J(i) = J_0(i) \leq J_k(i) \leq J_{k+1}(i), \qquad \text{for all } i.$$

Since $J_k(i)$ converges to $J^*(i)$ as $k \to \infty$, it follows that we will also have

$$J(i) = J_0(i) \leq J^*(i), \qquad \text{for all } i.$$

Thus out of all J satisfying the constraint (4.81), J^* is the largest on a component-by-component basis.

Unfortunately, for large n the dimension of the linear program (4.82) can be very large and its solution can be impractical, particularly in the absence of special structure. In this case, we may consider finding an approximation to J^*, which can be used in turn to obtain a (suboptimal) policy through approximation in value space.

One possibility is to approximate $J^*(i)$ with a linear feature-based architecture

$$\tilde{J}(i,r) = \sum_{\ell=1}^m r_\ell \phi_\ell(i),$$

where $r = (r_1, \ldots, r_m)$ is a vector of parameters, and for each state i, $\phi_\ell(i)$ are some features. It is then possible to determine r by using $\tilde{J}(i, r)$ in place of J^* in the preceding linear programming approach. In particular, we compute r as the solution of the program

$$\begin{aligned} &\text{maximize} \quad \sum_{i \in \tilde{I}} \tilde{J}(i, r) \\ &\text{subject to} \quad \tilde{J}(i, r) \leq \sum_{i=1}^{n} p_{ij}(u)\big(g(i, u, j) + \alpha \tilde{J}(j, r)\big), \quad i \in \tilde{I},\ u \in \tilde{U}(i), \end{aligned}$$

where $\tilde{I}$ is either the state space $I = \{1, \ldots, n\}$ or a suitably chosen subset of I, and $\tilde{U}(i)$ is either $U(i)$ or a suitably chosen subset of $U(i)$. This is a linear program because $\tilde{J}(i, r)$ is assumed linear in the parameter vector r.

The major difficulty with this approximation approach is that while the dimension of r may be moderate, the number of constraints can be extremely large. It can be as large as nm, where n is the number of states and m is the maximum number of elements of the control constraint sets $U(i)$. Thus for a large problem it is essential to reduce drastically the number of constraints. Random sampling methods may be used to select a suitable subset of the constraints to enforce (perhaps using some known suboptimal policies), and progressively enrich the subset as necessary. With such constraint sampling schemes, the linear programming approach may be practical even for problems with a very large number of states. Its application, however, may require considerable sophistication, and a substantial amount of computation (see de Farias and Van Roy [DFV03], [DFV04], [DeF04]).

We finally mention the possibility of using linear programming to evaluate approximately the cost function J_μ of a stationary policy μ in the context of approximate PI. The motivation for this is that the linear program to evaluate a given policy involves fewer constraints.

4.11 APPROXIMATION IN POLICY SPACE

We will now consider briefly an alternative to approximation in value space: approximation within the space of policies. In particular, we parametrize stationary policies with a parameter vector r and denote them by $\tilde{\mu}(r)$, with components $\tilde{\mu}(i, r)$, $i = 1, \ldots, n$. The parametrization may be feature-based and/or may involve a neural network. The idea is then to optimize some measure of performance with respect to the parameter r.

Note that it is possible for a suboptimal control scheme to employ both types of approximation: in policy space and in value space, with a distinct architecture for each case (examples of such schemes have been discussed briefly in Sections 2.1.5 and 4.7.3). When neural networks are

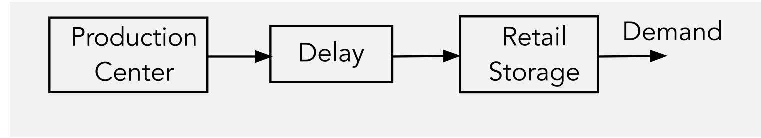

Figure 4.11.1. Illustration of a simple supply chain system.

used, this is known as the simultaneous use of a "policy network" (or "actor network") and a "value network" (or "critic network"), each with its own set of parameters (see the following discussion on expert training).

Let us provide two examples where policy parametrizations are natural.

Example 4.11.1: (Supply chain parametrization)

There are many problems where the general structure of an optimal or near-optimal policy is known through analysis or insight into the problem's structure. An important case are supply chain systems involving production, inventory, and retail centers that are connected with transportation links. A simple example is illustrated in Fig. 4.11.1. Here a retail center places orders to the production center, depending on current stock. There may be orders in transit, and demand and delays can be stochastic. Such a problem can be formulated by DP but can be very difficult to solve exactly. However, intuitively, a near-optimal policy has a simple form: When the retail inventory goes below some critical level r_1, order an amount to bring the inventory to a target level r_2. Here a policy is specified by the parameter vector $r = (r_1, r_2)$, and can be trained by one of the methods of this section. This type of approach readily extends to the case of a complex network of production/retail centers, multiple products, etc.

Example 4.11.2: (Policy parametrization through cost parametrization)

In an important approach for parametrization of policies we start with a parametric cost function approximation $\tilde J(j, r)$. We then define a parametrization in policy policy through the one-step lookahead minimization

$$\tilde\mu(i, r) \in \arg\min_{u\in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i, u, j) + \tilde J(j, r)\big),$$

where $\tilde J$ is a function of a given form that depends on r. For example, $\tilde J$ may be a linear feature-based architecture, with features possibly obtained through a separately trained neural network. The policies $\tilde\mu(r)$ thus defined form a class of one-step lookahead policies parametrized by r. We may then determine r through some form of policy training method.

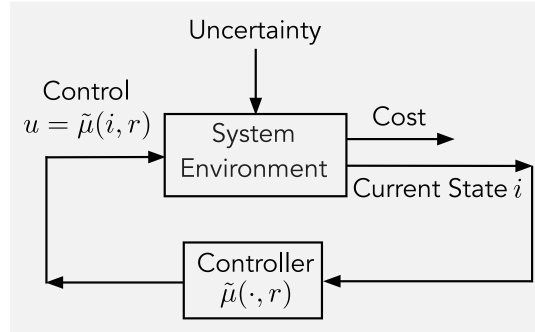

Figure 4.11.2 The optimization framework for approximation in policy space. Here policies are parametrized with a parameter vector r and denoted by $\tilde{\mu}(r)$, with components $\tilde{\mu}(i, r)$, $i = 1, \ldots, n$. Each parameter value r determines a policy $\mu(r)$, and a cost $J_{\tilde{\mu}(r)}(i_0)$ for each initial state i_0, as indicated in the figure. The optimization approach determines r through the minimization

$$\min_r E\big\{J_{\tilde{\mu}(r)}(i_0)\big\},$$

where the expected value above is taken with respect to a suitable probability distribution of i_0.

In what follows we will discuss briefly two training approaches for approximation in policy space.

4.11.1 Training by Cost Optimization - Policy Gradient and Random Search Methods

According to the first approach, we parametrize the policies by the parameter vector r, and we optimize the corresponding expected cost over r. In particular, we determine r through the minimization

$$\min_r E\big\{J_{\tilde{\mu}(r)}(i_0)\big\}, \tag{4.83}$$

where $J_{\tilde{\mu}(r)}(i_0)$ is the cost of the policy $\tilde{\mu}(r)$ starting from the initial state i_0, and the expected value above is taken with respect to a suitable probability distribution of the initial state i_0 (cf. Fig. 4.11.2). In the case where the initial state i_0 is known and fixed, the method involves just minimization of $J_{\tilde{\mu}(r)}(i_0)$ over r, This simplifies a great deal the minimization, particularly when the problem is deterministic.

Gradient Methods for Cost Optimization

Let us first consider methods that perform the minimization (4.83) by using a gradient method, and for simplicity let us assume that the initial

condition i_0 is known. Thus the problem is to minimize $J_{\tilde{\mu}(r)}(i_0)$ over r by using the gradient method

$$r^{k+1} = r^k - \gamma^k \nabla J_{\tilde{\mu}(r^k)}(i_0), \qquad k = 0, 1, \ldots, \tag{4.84}$$

assuming that $J_{\tilde{\mu}(r)}(i_0)$ is differentiable with respect to r. Here γ^k is a positive stepsize parameter, and $\nabla(\cdot)$ denotes gradient with respect to r evaluated at the current iterate r^k.

The difficulty with this method is that the gradient $\nabla J_{\tilde{\mu}(r^k)}(i_0)$ may not be explicitly available. In this case, the gradient must be approximated by finite differences of cost function values $J_{\tilde{\mu}(r^k)}(i_0)$. Unfortunately, in the case of a stochastic problem, the cost function values may be computable only through Monte Carlo simulation. This may introduce a large amount of noise, so it is likely that many samples will need to be averaged in order to obtain sufficiently accurate gradients, thereby making the method inefficient. On the other hand, when the problem is deterministic, this difficulty does not appear, and the use of the gradient method (4.84) or other methods that do not rely on the use of gradients (such as coordinate descent) is facilitated.

There is extensive literature on alternative and more efficient policy gradient methods for stochastic problems, which are based on gradient approximations through sampling. A popular type of method is based on the use of randomized policies [i.e., policies that map a state i to a probability distribution over the set of controls $U(i)$, rather than mapping onto a single control].† The method also uses a convenient gradient formula that involves the natural logarithm of the sampling distribution, and is known as the *log-likelihood trick*. We will next provide an outline of the ideas underlying this method.

Policy Gradient Methods for Randomized Policies

The detailed description and analysis of randomized policies and the associated policy gradient methods are beyond our scope. To get a sense of the general principle underlying this gradient-based approach, let us digress from the DP context of this chapter, and consider the generic optimization problem

$$\min_{z \in Z} F(z),$$

---

† The AlphaGo and AlphaZero programs (Silver et al. [SHM16], [SHS17]) also use randomized policies, and a policy adjustment scheme that involves incremental changes along "directions of improvement." However, these changes are implemented through the MCTS algorithm used by these programs, without the explicit use of a gradient (see the discussion in Section 2.4.2). Thus it may be said that the AlphaGo and AlphaZero programs involve a form of approximation in policy space (as well as approximation in value space), which bears resemblance but cannot be classified as a policy gradient method.

where Z is a subset of the m-dimensional space $\Re^m$, and F is some real-valued function over $\Re^m$.

We will take the unusual step of converting this problem to the *stochastic* optimization problem

$$\min_{p\in\mathcal{P}_Z} E_p\big\{F(z)\big\}, \tag{4.85}$$

where z is viewed as a random variable, $\mathcal{P}_Z$ is the set of probability distributions over Z, p denotes the generic distribution in $\mathcal{P}_Z$, and $E_p\{\cdot\}$ denotes expected value with respect to p. Of course this enlarges the search space from Z to $\mathcal{P}_Z$, but it allows the use of randomization schemes and simulation-based methods, even if the original problem is deterministic.

In a stochastic DP context, such as the SSP and discounted problems that we focused on in this chapter, the cost function is already stochastic, but to obtain a problem of the form (4.85), we must enlarge the set of policies to include *randomized policies*, mapping a state i into a probability distribution over the set of controls $U(i)$.

Suppose now that we restrict attention to a subset $\tilde{\mathcal{P}}_Z \subset \mathcal{P}_Z$ of probability distributions $p(z;r)$ that are parametrized by some continuous parameter r, e.g., a vector in some m-dimensional space. In other words, we approximate the stochastic optimization problem (4.85) with the restricted problem

$$\min_{r} E_{p(z;r)}\big\{F(z)\big\}.$$

Then we may use a gradient method for solving this problem, such as

$$r^{k+1} = r^k - \gamma^k \nabla\Big(E_{p(z;r^k)}\big\{F(z)\big\}\Big), \qquad k = 0,1,\ldots, \tag{4.86}$$

where $\nabla(\cdot)$ denotes gradient with respect to r of the function in parentheses, evaluated at the current iterate r^k.

A key fact here is that there is a useful formula for the gradient in Eq. (4.86), which involves the gradient with respect to r of the natural logarithm $\log\big(p(z;r^k)\big)$. Indeed, assuming for notational convenience that $p(z;r^k)$ is a discrete distribution, we have

$$\begin{aligned}
\nabla\Big(E_{p(z;r^k)}\big\{F(z)\big\}\Big) &= \nabla\left(\sum_{z\in Z} p(z;r^k)F(z)\right)\\
&= \sum_{z\in Z} \nabla p(z;r^k)F(z)\\
&= \sum_{z\in Z} p(z;r^k)\frac{\nabla p(z;r^k)}{p(z;r^k)}F(z)\\
&= \sum_{z\in Z} p(z;r^k)\nabla\Big(\log\big(p(z;r^k)\big)\Big)F(z),
\end{aligned}$$

and finally

$$\nabla\Big(E_{p(z;r^k)}\big\{F(z)\big\}\Big) = E_{p(z;r^k)}\bigg\{\nabla\Big(\log\big(p(z;r^k)\big)\Big)F(z)\bigg\}.$$

The preceding formula suggests an incremental implementation of the gradient iteration (4.86) that approximates the expected value in the right side above with a single sample. This is in the spirit of the incremental/stochastic gradient training methods that we have discussed in Sections 3.1.3 and 3.2.1. The typical iteration of this method is as follows.

Sample-Based Gradient Method for Parametric Approximation of $\min_{z\in Z} F(z)$

Let r_k be the current parameter vector.

(a) Obtain a sample z^k according to the distribution $p(z;r^k)$.

(b) Compute the gradient $\nabla\Big(\log\big(p(z^k;r^k)\big)\Big)$.

(c) Iterate according to

$$r^{k+1} = r^k - \gamma^k\nabla\Big(\log\big(p(z^k;r^k)\big)\Big)F(z^k). \tag{4.87}$$

The advantage of the preceding sample-based method is its simplicity and generality. It allows the use of parametric approximation for any minimization problem, as long as the logarithm of the sampling distribution $p(z;r)$ can be differentiated with respect to r, and samples of z can be obtained using the distribution $p(z;r)$.

Another major advantage is that the iteration (4.87) requires the sample cost values $F(z^k)$ but not the gradient of F. As a result the iteration has a *model-free character*: we don't need to know the form of the function F as long as we have a simulator that produces the cost function value $F(z)$ for any given z. There are, however, some challenging issues to consider.

The first of these is that the problem solved is a randomized version of the original. If the gradient iteration (4.87) produces a parameter $\overline{r}$ in the limit and the distribution $p(z;\overline{r})$ is not atomic (i.e., it is not concentrated a single point), then a solution $\overline{z}\in Z$ must be extracted from $p(z;\overline{r})$. In the SSP and discounted problems of this chapter, the subset $\tilde{\mathcal{P}}_Z$ of parametric distributions typically contains the atomic distributions, while it can be shown that minimization over the set of all distributions $\mathcal{P}_Z$ produces the same optimal value as minimization over Z (the use of randomized policies does not improve the optimal cost of the problem), so this difficulty does not arise.

Another issue is how to design the approximation architecture and how to collect the samples z^k. Different methods must strike a balance of convenient implementation, and a reasonable guarantee that the search space Z is sufficiently well explored.

Finally, we must deal with the issue of efficient computation of the sampled gradient

$$\nabla\Big(\log\big(p(z^k;r^k)\big)\Big).$$

In the context of DP, including the SSP and discounted problems that we have been dealing with, there are some specialized procedures and corresponding parametrizations to approximate this gradient conveniently. The following is an example.

Example 4.11.3 (Policy Gradient Method - Discounted Cost)

Consider the α-discounted problem and denote by z the infinite horizon state-control trajectory:

$$z = \{i_0, u_0, i_1, u_1, \ldots\}.$$

We consider a parametrization of randomized policies with parameter r, so the control at state i is generated according to a distribution $p(u \mid i; r)$ over $U(i)$. Then for a given r, the state-control trajectory z is a random vector with probability distribution denoted $p(z;r)$. The cost corresponding to the trajectory z is

$$F(z) = \sum_{m=0}^{\infty} \alpha^m g(i_m, u_m, i_{m+1}),$$

and the problem is to minimize

$$E_{p(z;r)}\big\{F(z)\big\},$$

over r.

To apply the sample-based gradient method (4.87), given the current iterate r^k, we must generate the sample state-control trajectory z^k, according to the distribution $p(z;r^k)$, compute the corresponding cost $F(z^k)$, and also calculate the gradient

$$\nabla\Big(\log\big(p(z^k;r^k)\big)\Big). \tag{4.88}$$

Let us assume a model-based context where the transition probabilities $p_{ij}(u)$ are known, and let us also assume that the logarithm of the randomized policy distribution $p(u \mid i; r)$ is differentiable with respect to r. Then the logarithm that is differentiated in Eq. (4.88) can be written as

$$\begin{aligned}\log\big(p(z^k;r^k)\big) &= \log \prod_{m=0}^{\infty} p_{i_m i_{m+1}}(u_m) p(u_m \mid i_m; r^k) \\ &= \sum_{m=0}^{\infty} \log\big(p_{i_m i_{m+1}}(u_m)\big) + \sum_{m=0}^{\infty} \log\big(p(u_m \mid i_m; r^k)\big),\end{aligned}$$

and its gradient (4.88), which is needed in the iteration (4.87), is given by

$$\nabla\Big(\log\big(p(z^k;r^k)\big)\Big) = \sum_{m=0}^{\infty} \log\big(p_{i_m i_{m+1}}(u_m)\big) + \sum_{m=0}^{\infty} \nabla\Big(\log\big(p(u_m \mid i_m;r^k)\big)\Big). \tag{4.89}$$

Thus the policy gradient method (4.87) is very simple to implement: for the given parameter vector r^k, we generate a sample trajectory z^k using the corresponding randomized policy $p(u \mid i; r^k)$, we calculate the corresponding sample cost $F(z^k)$, and the gradient (4.88) using the expression (4.89), and we update r^k using Eq. (4.87).

Policy gradient methods for other types of DP problems can be similarly developed, including for model-free contexts. A further discussion is beyond our scope, and we refer to the end-of-chapter literature for a variety of specific methods.

The main drawback of policy gradient methods is potential unreliability due to the stochastic uncertainty corrupting the calculation of the gradients, the slow convergence that is typical of gradient methods in many settings, and the presence of local minima. For this reason, methods based on random search have been considered as potentially more reliable alternatives to policy gradient methods. Viewed from a high level, random search methods are similar to policy gradient methods in that they aim at iterative cost improvement through sampling. However, they need not involve randomized policies, they are not subject to cost differentiability restrictions, and they offer some global convergence guarantees, so in principle they are not affected much by local minima.

Random Search and Cross-Entropy Methods

Random search methods explore the space of the parameter vector r in some randomized but intelligent fashion. There are several types of random search methods for general optimization, and some of them have been suggested for approximate DP. We will briefly describe the *cross-entropy method*, which has gained considerable attention.

The method bears resemblance to policy gradient methods, in that it generates a parameter sequence $\{r^k\}$ by changing r^k to r^{k+1} along a direction of "improvement." This direction is obtained by using the policy $\tilde{\mu}(r^k)$ to generate randomly cost samples corresponding to a set of sample parameter values that are concentrated around r^k. The current set of sample parameters are then screened: some are accepted and the rest are rejected, based on a cost improvement criterion. Then r^{k+1} is determined as a "central point" or as the "sample mean" in the set of accepted sample parameters, some more samples are generated randomly around r^{k+1}, and the process is repeated; see Fig. 4.11.3. Thus successive iterates r^k are "central points" of successively better groups of samples, so in some broad

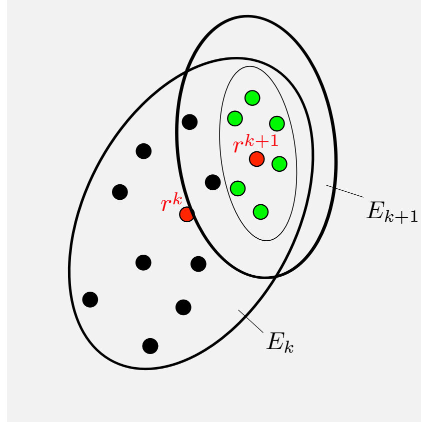

Figure 4.11.3 Schematic illustration of the cross entropy method. At the current iterate r^k, we construct an ellipsoid E_k centered at r^k. We generate a number of random samples within E_k, and we "accept" a subset of the samples that have "low" cost. We then choose r^{k+1} to be the sample mean of the accepted samples, and construct a sample "covariance" matrix of the accepted samples. We then form the new ellipsoid E_{k+1} using this matrix and a suitable radius parameter, and continue. Notice the resemblance with a policy gradient method: we move from r^k to r^{k+1} in a direction of cost improvement.

sense, the random sample generation process is guided by cost improvement. This idea is shared with evolutionary programming; see e.g., the books [Bac96], [DeJ06], and the paper by Salimans et al. [SHC17], which provides references to many other works.

The cross-entropy method is very simple to implement, does not suffer from the fragility of gradient-based optimization, does not involve randomized policies, and relies on some supportive theory; we refer to the literature for details. Like all random search methods, its convergence rate guarantees are limited, and its success depends on domain-specific insight and the skilled use of heuristics. However, the method is well-suited for the use of parallel computation, and has gained a favorable reputation through some impressive successes. In particular, it was used for learning a high-scoring strategy in the game of tetris; see Szita and Lorinz [SzL06], and Thiery and Scherrer [ThS09]. There have also been reports of domain-specific successes with related random search methods; see Salimans et al. [SHC17].

4.11.2 Expert Supervised Training

According to the second approximation in policy space approach, we choose the parameter r by "training" on a large number of sample state-control pairs (i^s, u^s), $s = 1, \ldots, q$, such that for each s, u^s is a "good" control at state i^s. This can be done for example by solving the least squares problem

$$\min_r \sum_{s=1}^{q} \big\| u^s - \tilde{\mu}(i^s, r) \big\|^2 \tag{4.90}$$

(possibly with added regularization). In particular, we may determine u^s by a human or a software "expert" that can choose "near-optimal" controls at given states, so $\tilde{\mu}$ is trained to match the behavior of the expert. We have also discussed this approach in Section 2.1.3, in the context of finite horizon problems. In the context of artificial intelligence, it comes within the framework of *supervised learning* methods.†

Another possibility is to suitably select a large number of sample states i^s, $s = 1, \ldots, q$, and generate the controls u^s, $s = 1, \ldots, q$, through a one-step lookahead minimization of the form

$$u^s = \arg \min_{u \in U(i^s)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i^s, u, j) + \tilde{J}_{k+1}(j)\big), \tag{4.91}$$

where $\tilde{J}$ is a suitable one-step lookahead function (multistep lookahead can also be used). Similarly, once a parametric Q-factor approximation architecture $\tilde{Q}(i, u, r)$ is chosen, we can select a large number of sample states i^s, $s = 1, \ldots, q$, and then compute the controls u^s, $s = 1, \ldots, q$, through the one-step lookahead minimization

$$u^s = \arg \min_{u \in U(i^s)} \tilde{Q}(i^s, u, r). \tag{4.92}$$

In this case, we will be collecting sample state-control pairs (i^s, u^s), $s = 1, \ldots, q$, using approximation in value space through Eq. (4.91) or Eq. (4.92), and then applying approximation in policy space through Eq. (4.90).

Of course in the expert training approach we cannot expect to obtain a policy that performs better than the expert with which it is trained, in the case of Eq. (4.90), or the one-step lookahead policy that is based on the approximation $\tilde{J}$ or $\tilde{Q}$, in the case of Eq. (4.91) or Eq. (4.92), respectively. However, a major advantage is that once the parametrized policy is obtained, the on-line implementation of the policy is fast and does not involve extensive calculations such as minimizations of the form (4.91). This advantage is generally shared by schemes that are based on approximation in policy space.

---

† Tesauro [Tes89a], [Tes89b] constructed a backgammon player, trained by a neural network and a supervised learning approach (called "comparison learning"), which used examples from human expert play (he was the expert who provided the training samples). However, his subsequent TD-based algorithm [Tes92], [Tes94], [Tes95], performed substantially better, and his rollout-based algorithm [TeG96] performed even better. The Deepchess program by David, Netanyahu, and Wolf [DNW16] provides another example of an expert supervised training approach.

4.12 NOTES AND SOURCES

In this chapter we have provided an introduction to infinite horizon DP with a view towards approximate solution methods that are suitable for large-scale problems. We have restricted ourselves to finite-state problems with perfect state information. Infinite-state problems as well as partial state information and average cost problems exhibit more complex behaviors and present many challenges for approximate DP methods. The theory of these problems is developed in several books, including the author's [Ber12] and [Ber18a]. The latter book contains much recent advanced research on infinite-state deterministic and stochastic shortest path problems. The book by Puterman [Put94] contains a detailed account of discounted and average cost finite-state Markovian decision problems.

The methods of VI and PI, and their optimistic variants are the cornerstones of infinite horizon DP, and they serve as the principal points of departure for approximations. In addition to the computational topics covered in this chapter, we should mention that both VI and PI can be implemented via distributed asynchronous computation; see the author's papers on DP and fixed point computations [Ber82], [Ber83], the paper by Williams and Baird on asynchronous PI [WiB93], and the series of papers by Bertsekas and Yu [BeY10], [BeY12], [YuB13a] on asynchronous optimistic PI and Q-learning. Generally, asynchronous and distributed algorithms are natural in computational contexts involving simulation, which by its nature is well suited to both multiprocessing and asynchronous implementation.

The variational form of Bellman's equation (Section 4.2) has been used in various contexts, involving error bounds for value iteration, since the early days of DP theory, see e.g., [Ber12], Section 2.1.1. The variational form of Bellman's equation is also implicit in the adaptive aggregation framework of Bertsekas and Castanon [BeC89]. In the context of RL, the variational equation has been used in various algorithmic contexts under the name *reward shaping* or *potential-based shaping* (we have used the term "cost shaping" here as we are focusing on cost minimization); see e.g., the papers by Ng, Harada, and Russell [NHR99], Wiewiora [Wie03], Asmuth, Littman, and Zinkov [ALZ08], Devlin and Kudenko [DeK11], Grzes [Grz17] for some representative works. While reward shaping does not change the optimal policies of the original DP problem, it may change significantly the suboptimal policies produced by approximate DP methods that use linear feature-based approximation. Basically, with reward shaping and a linear approximation architecture, *V is used as an extra feature*. This is closely related with the idea of using approximate cost functions of policies as basis functions in approximation architectures; see the discussion in the neuro-dynamic programming book [BeT96], Section 3.1.4.

Fitted VI algorithms have been used for finite horizon problems since the early days of DP. They are conceptually simple and easily imple-

mentable, and they are in wide use for approximation of either optimal costs or Q-factors (see e.g., Gordon [Gor95], Longstaff and Schwartz [LoS01], Ormoneit and Sen [OrS02], Ernst, Geurts, and Wehenkel [EGW06], Antos, Munos, and Szepesvari [AMS07], and Munos and Szepesvari [MuS08]).

The performance bound of Props. 4.6.1 and 4.6.3 for multistep lookahead, rollout, and terminal cost function approximation are sharper versions of earlier results for one step lookahead, terminal cost function approximation, but no rollout; see Prop. 6.1.1 in the author's DP textbook [Ber17] (and earlier editions), as well as [Ber18a], Section 2.2. The approximate PI method of Section 4.7.3 has been proposed by Fern, Yoon, and Givan [FYG06], and variants have also been discussed and analyzed by several other authors. The method (with some variations) has been used to train a tetris playing computer program that performs impressively better than programs that are based on other variants of approximate policy iteration, and various other methods; see Scherrer [Sch13], Scherrer et al. [SGG15], and Gabillon, Ghavamzadeh, and Scherrer [GGS13], who also provide an analysis of the method.

Q-learning (Section 4.8) was first proposed by Watkins [Wat89], and had a major impact in the development of the field. A rigorous convergence proof of Q-learning was given by Tsitsiklis [Tsi94], in a more general framework that combined several ideas from stochastic approximation theory and the theory of distributed asynchronous computation. This proof covered discounted problems, and SSP problems where all policies are proper. It also covered SSP problems with improper policies, assuming that the Q-learning iterates are either nonnegative or bounded. Convergence without the nonnegativity or the boundedness assumption was shown by Yu and Bertsekas [YuB13b]. Optimistic asynchronous versions of PI based on Q-learning, which have solid convergence properties, are given by Bertsekas and Yu [BeY10], [BeY12], [YuB13a]. The distinctive feature of these methods is that the policy evaluation process aims towards the solution of an optimal stopping problem rather than towards to solution of the linear system of Bellman equations associated with the policy; this is needed for the convergence proof, to avoid the pathological behavior first identified by Williams and Baird [WiB93], and noted earlier.

The advantage updating idea, which was noted in the context of finite horizon problems in Section 3.3, can be readily extended to infinite horizon problems. In this context, it was proposed by Baird [Bai93], [Bai94]; see [BeT96], Section 6.6. A related variant of approximate policy iteration and Q-learning, called *differential training*, has been proposed by the author in [Ber97b] (see also Weaver and Baxter [WeB99]).

Projected equations (Section 4.9) underlie Galerkin methods, which have a long history in scientific computation. They are widely used for many types of problems, including the approximate solution of large linear systems arising from discretization of partial differential and integral equations. The connection of approximate policy evaluation based on projected

equations with Galerkin methods was first discussed by Yu and Bertsekas [YuB10], and Bertsekas [Ber11c], and is potentially important as it may lead to cross-fertilization of ideas. However, the Monte Carlo simulation ideas that are central in approximate DP differentiate the projected equation methods of the present chapter from the Galerkin methodology. On the other hand, Galerkin methods apply to a wide range of problems, far beyond DP, and the simulation-based ideas of approximate DP can consequently be extended to apply more broadly (see [Ber12], Section 7.3).

Temporal difference methods originated in RL, where they are viewed as a means to encode the error in predicting future costs of a given policy, which is associated and an approximation architecture. They were introduced in the works of Samuel [Sam59], [Sam67] on a checkers-playing program. The work by Sutton [Sut88], following earlier work by Barto, Sutton, and Anderson [BSA83], formalized temporal differences and proposed the TD(λ) method. This was a major development and motivated a lot of research in simulation-based DP, particularly following an impressive early success with the backgammon playing program of Tesauro [Tes92], [Tes94].

The three methods TD(λ), LSTD(λ), and LSPE(λ) are discussed in detail in the journal and textbook RL literature. For a discussion that extends our presentation of Section 4.9, see Chapters 6 and 7 of the book [Ber12].

The convergence of TD(λ) was proved by Tsitsiklis and Van Roy [TsV97], with extensions in [TsV99a] and [TsV99b]. The author's papers [Ber16b], [Ber18d] describe the connection of TD and proximal methods, a central methodology in convex optimization. In particular, TD(λ) is shown to be a stochastic version of the proximal algorithm for solving linear systems of equations, and extensions of TD(λ) for solving nonlinear systems of equations are described.

The LSTD(λ) algorithm was first proposed by Bradtke and Barto [BrB96] for $\lambda = 0$, and was extended for $\lambda > 0$ later by Boyan [Boy02]. Convergence analyses of LSTD(λ) under assumptions of increasing generality were given by Nedić and Bertsekas [NeB03], Bertsekas and Yu [BeY09], and Yu [Yu12].

The LSPE(λ) algorithm was first proposed by Bertsekas and Ioffe [BeI96] under the name λ-*policy iteration*, and it was used to train a tetris playing program using the feature-based linear architecture described in Example 3.1.3. The motivation for λ-policy iteration was to provide a better alternative to TD(λ)-based policy iteration, which failed within the tetris context. LSPE(λ) was also given in the book [BeT96], Section 2.3.1, with subsequent contributions by Nedic, Borkar, Yu, Scherrer, and the author [NeB03], [BBN04], [YuB07], [BeY09], [YuB09], [Ber11b], [Yu12], [Sch13], [Ber18a].

In our discussion here, we did not go much into the implementation details of TD(λ), LSTD(λ), and LSPE(λ); see the approximate DP/RL

textbooks cited earlier, and the paper by Bertsekas and Yu [BeY09], which adapts the TD methodology to the solution of large systems of linear equations.

Policy gradient methods have a long history. For a detailed discussion and references we refer to the book by Sutton and Barto [SuB18], the monograph by Deisenroth, Neumann, and Peters [DNP11], and the survey by Grondman et. al. [GBL12]. The use of the log-likelihood trick in the context of simulation-based DP is generally attributed to Williams [Wil92]. Early works on simulation-based policy gradient schemes for various DP problems have been given by Glynn [Gly87], L'Ecuyer [L'Ec91], Fu and Hu [FuH94], Jaakkola, Singh, and Jordan [JSJ95], Cao and Chen [CaC97], Cao and Wan [CaW98]. The more recent works of Marbach and Tsitsiklis [MaT01], [MaT03], Konda and Tsitsiklis [KoT99], [KoT03], and Sutton et. al. [SMS99] have been influential. For textbook discussions of the cross-entropy method, see Rubinstein and Kroese [RuK04], [RuK17], and Busoniu et. al. [BBD10], and for surveys see de Boer et. al. [BKM05], and Kroese et. al. [KRC13].

4.13 APPENDIX: MATHEMATICAL ANALYSIS

In this appendix we provide proofs of the mathematical results stated in this chapter. We also prove some supplementary results that are described in the chapter without formal statement.

We will make heavy use of the DP operators T and T_μ, particularly for the discounted problem:

$$(TJ)(i) = \min_{u\in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha J(j)\big), \qquad i = 1,\ldots,n,$$

$$(T_\mu J)(i) = \sum_{i=1}^{n} p_{ij}\big(\mu(i)\big)\Big(g\big(i,\mu(i),j\big) + \alpha J(j)\Big), \qquad i = 1,\ldots,n.$$

A key property is the monotonicity of these operators, i.e.,

$$TJ \geq TJ', \quad T_\mu J \geq T_\mu J', \qquad \text{for all } J \text{ and } J' \text{ with } J \geq J'.$$

Also for the discounted problem, we have the "constant shift" property, which states that if the functions J is increased uniformly by a constant c, then the functions TJ and $T_\mu J$ are also increased uniformly by the constant αc.

4.13.1 Proofs for Stochastic Shortest Path Problems

We provide the proofs of Props. 4.2.1-4.2.5 from Section 4.2. A key insight for the analysis is that the expected cost incurred within an m-stage block vanishes exponentially as the start of the block moves forward (here m is the integer specified by Assumption 4.2.1, i.e., the termination state can be reached within m steps with positive probability from every starting state). In particular, the cost in the m stages between km and $(k+1)m-1$ is bounded in absolute value by $\rho^k C$, where

$$C = m \max_{\substack{i=1,\ldots,n \\ j=1,\ldots,n,t \\ u\in U(i)}} \big|g(i,u,j)\big|. \tag{4.93}$$

Thus, we have

$$\big|J_\pi(i)\big| \leq \sum_{k=0}^{\infty} \rho^k C = \frac{1}{1-\rho} C. \tag{4.94}$$

This shows that the "tail" of the cost series,

$$\sum_{k=mK}^{\infty} E\Big\{g\big(x_k,\mu_k(x_k),w_k\big)\Big\},$$

vanishes as K increases to ∞, since the probability that $x_{mK}\neq t$ decreases like ρ^K [cf. Eq. (4.6)]. Intuitively, since the "tail" of the cost series can be neglected as $K\to\infty$, it is valid to take the limit in the finite horizon DP algorithm, and obtain the infinite horizon Bellman equation and VI convergence. Mathematically, this is the essence of the following proofs.

Proposition 4.2.1: (Convergence of VI) Given any initial conditions $J_0(1),\ldots,J_0(n)$, the sequence $\big\{J_k(i)\big\}$ generated by the VI algorithm

$$J_{k+1}(i) = \min_{u\in U(i)} \left[p_{it}(u)g(i,u,t) + \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j)+J_k(j)\big)\right], \tag{4.95}$$

converges to the optimal cost $J^*(i)$ for each $i=1,\ldots,n$.

Proof: For every positive integer K, initial state x_0, and policy $\pi = \{\mu_0,\mu_1,\ldots\}$, we break down the cost $J_\pi(x_0)$ into the portions incurred

over the first mK stages and over the remaining stages:

$$J_\pi(x_0) = \lim_{N\to\infty} E\left\{\sum_{k=0}^{N-1} g\big(x_k, \mu_k(x_k), w_k\big)\right\}$$
$$= E\left\{\sum_{k=0}^{mK-1} g\big(x_k, \mu_k(x_k), w_k\big)\right\} + \lim_{N\to\infty} E\left\{\sum_{k=mK}^{N-1} g\big(x_k, \mu_k(x_k), w_k\big)\right\}.$$

The expected cost during the Kth m-stage cycle [stages Km to $(K+1)m-1$] is upper bounded by $C\rho^K$ [cf. Eqs. (4.6) and (4.94)], so that

$$\left|\lim_{N\to\infty} E\left\{\sum_{k=mK}^{N-1} g\big(x_k, \mu_k(x_k), w_k\big)\right\}\right| \le C\sum_{k=K}^{\infty}\rho^k = \frac{\rho^K C}{1-\rho}.$$

Also, denoting $J_0(t) = 0$, let us view J_0 as a terminal cost function and bound its expected value under π after mK stages. We have

$$\begin{aligned}\Big|E\big\{J_0(x_{mK})\big\}\Big| &= \left|\sum_{i=1}^{n} P(x_{mK} = i \mid x_0, \pi) J_0(i)\right| \\ &\le \left(\sum_{i=1}^{n} P(x_{mK} = i \mid x_0, \pi)\right) \max_{i=1,\ldots,n}\big|J_0(i)\big| \\ &\le \rho^K \max_{i=1,\ldots,n}\big|J_0(i)\big|,\end{aligned}$$

since the probability that $x_{mK} \neq t$ is less or equal to ρ^K for any policy. Combining the preceding relations, we obtain

$$\begin{aligned}&-\rho^K \max_{i=1,\ldots,n}\big|J_0(i)\big| + J_\pi(x_0) - \frac{\rho^K C}{1-\rho} \\ &\qquad\le E\left\{J_0(x_{mK}) + \sum_{k=0}^{mK-1} g\big(x_k, \mu_k(x_k), w_k\big)\right\} \\ &\qquad\le \rho^K \max_{i=1,\ldots,n}\big|J_0(i)\big| + J_\pi(x_0) + \frac{\rho^K C}{1-\rho}.\end{aligned} \tag{4.96}$$

Note that the expected value in the middle term of the above inequalities is the mK-stage cost of policy π starting from state x_0, with a terminal cost $J_0(x_{mK})$; the minimum of this cost over all π is equal to the value $J_{mK}(x_0)$, which is generated by the DP recursion (4.95) after mK iterations. Thus, by taking the minimum over π in Eq. (4.96), we obtain for all x_0 and K,

$$\begin{aligned}&-\rho^K \max_{i=1,\ldots,n}\big|J_0(i)\big| + J^*(x_0) - \frac{\rho^K C}{1-\rho} \\ &\qquad\le J_{mK}(x_0) \\ &\qquad\le \rho^K \max_{i=1,\ldots,n}\big|J_0(i)\big| + J^*(x_0) + \frac{\rho^K C}{1-\rho},\end{aligned}$$

and by taking the limit as $K \to \infty$, we obtain

$$\lim_{K\to\infty} J_{mK}(x_0) = J^*(x_0)$$

for all x_0. Since

$$\big|J_{mK+\ell}(x_0) - J_{mK}(x_0)\big| \le \rho^K C, \qquad \ell = 1,\dots,m,$$

we see that $\lim_{K\to\infty} J_{mK+\ell}(x_0)$ is the same for all $\ell = 1,\dots,m$, so that

$$\lim_{k\to\infty} J_k(x_0) = J^*(x_0).$$

Q.E.D.

Proposition 4.2.2: (Bellman's Equation) The optimal cost function $J^* = \big(J^*(1),\dots,J^*(n)\big)$ satisfies for all $i = 1,\dots,n$, the equation

$$J^*(i) = \min_{u\in U(i)} \left[p_{it}(u)g(i,u,t) + \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + J^*(j)\big) \right],$$

and in fact it is the unique solution of this equation.

Proof: By taking the limit as $k \to \infty$ in the DP iteration (4.95) and using the result of Prop. 4.2.1, we see that $J^*(1),\dots,J^*(n)$ satisfy Bellman's equation (we are using here the fact that the limit and minimization operations commute when the minimization is over a finite number of alternatives). To show uniqueness, observe that if $J(1),\dots,J(n)$ satisfy Bellman's equation, then the DP iteration (4.95) starting from $J(1),\dots,J(n)$ just replicates $J(1),\dots,J(n)$. It follows from the convergence result of Prop. 4.2.1 that $J(i) = J^*(i)$ for all i. **Q.E.D.**

Proposition 4.2.3: (VI and Bellman's Equation for Policies) For any stationary policy μ, the corresponding cost function $J_\mu = \big(J_\mu(1),\dots,J_\mu(n)\big)$ satisfies for all $i = 1,\dots,n$ the equation

$$J_\mu(i) = p_{it}\big(\mu(i)\big)g\big(i,\mu(i),t\big) + \sum_{j=1}^{n} p_{ij}\big(\mu(i)\big)\Big(g\big(i,\mu(i),j\big) + J_\mu(j)\Big),$$

and is in fact the unique solution of this equation. Furthermore, given any initial conditions $J_0(1), \ldots, J_0(n)$, the sequence $\{J_k(i)\}$ generated by the VI algorithm that is specific to μ,

$$J_{k+1}(i) = p_{it}\big(\mu(i)\big)g\big(i,\mu(i),t\big) + \sum_{j=1}^{n} p_{ij}\big(\mu(i)\big)\Big(g\big(i,\mu(i),j\big) + J_k(j)\Big),$$

converges to the cost $J_\mu(i)$ for each i.

Proof: Given μ, consider a modified stochastic shortest path problem, which is the same as the original except that the control constraint set contains only one element for each state i, the control $\mu(i)$; i.e., the control constraint set is $\tilde{U}(i) = \{\mu(i)\}$ instead of $U(i)$. From Prop. 4.2.2, J_μ solves uniquely Bellman's equation for this modified problem, i.e., for all i,

$$J_\mu(i) = p_{it}\big(\mu(i)\big)g\big(i,\mu(i),t\big) + \sum_{j=1}^{n} p_{ij}\big(\mu(i)\big)\Big(g\big(i,\mu(i),j\big) + J_\mu(j)\Big).$$

From Prop. 4.2.1, the VI algorithm converges to $J_\mu(i)$. **Q.E.D.**

Proposition 4.2.4: (Optimality Condition) A stationary policy μ is optimal if and only if for every state i, $\mu(i)$ attains the minimum in Bellman's equation (4.7).

Proof: We have that $\mu(i)$ attains the minimum in Eq. (4.7) if and only if for all $i = 1, \ldots, n$, we have

$$\begin{aligned} J^*(i) &= p_{it}(u)g(i,u,t) + \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + J^*(j)\big) \\ &= p_{it}\big(\mu(i)\big)g\big(i,\mu(i),t\big) + \sum_{j=1}^{n} p_{ij}\big(\mu(i)\big)\Big(g\big(i,\mu(i)\big) + J^*(j)\Big). \end{aligned}$$

Proposition 4.2.3 and this equation imply that $J_\mu(i) = J^*(i)$ for all i. Conversely, if $J_\mu(i) = J^*(i)$ for all i, Props. 4.2.2 and 4.2.3 imply this equation. **Q.E.D.**

Proposition 4.2.5: (Contraction Property of the DP Operator) The DP operators T and T_μ of Eqs. (4.8) and (4.9) are contraction mappings with respect to some weighted norm

$$\|J\| = \max_{i=1,\ldots,n} \frac{\big|J(i)\big|}{v(i)},$$

defined by some vector $v = \big(v(1),\ldots,v(n)\big)$ with positive components. In other words, there exist positive scalar $\rho < 1$ and $\rho_\mu < 1$ such that for any two n-dimensional vectors J and J', we have

$$\|TJ - TJ'\| \leq \rho\,\|J - J'\|, \qquad \|T_\mu J - T_\mu J'\| \leq \rho_\mu\,\|J - J'\|.$$

Proof: We first define the vector v using the problem of Example 4.2.1. In particular, we let $v(i)$ be the maximal expected number of steps to termination starting from state i. From Bellman's equation in Example 4.2.1, we have for all $i = 1,\ldots,n$, and stationary policies μ,

$$v(i) = 1 + \max_{u\in U(i)} \sum_{j=1}^{n} p_{ij}(u)v(j) \geq 1 + \sum_{j=1}^{n} p_{ij}\big(\mu(i)\big)v(j), \qquad i = 1,\ldots,n.$$

Thus we obtain for all μ,

$$\sum_{j=1}^{n} p_{ij}\big(\mu(i)\big)v(j) \leq v(i) - 1 \leq \rho\, v(i), \qquad i = 1,\ldots,n, \tag{4.97}$$

where ρ is defined by

$$\rho = \max_{i=1,\ldots,n} \frac{v(i)-1}{v(i)}.$$

Since $v(i) \geq 1$ for all i, we have $\rho < 1$.

We will now show that Eq. (4.97) implies the desired contraction property. Indeed, consider the operator T_μ, which when applied to a vector $J = \big(J(1),\ldots,J(n)\big)$ produces the vector $T_\mu J = \big((T_\mu J)(1),\ldots,(T_\mu J)(n)\big)$ defined by

$$(T_\mu J)(i) = p_{it}\big(\mu(i)\big)g\big(i,\mu(i),t\big) + \sum_{j=1}^{n} p_{ij}\big(\mu(i)\big)\Big(g\big(i,\mu(i),j\big) + J(j)\Big),$$

for all $i = 1, \ldots, n$. We have for all J, J', and i

$$\begin{aligned}(T_\mu J)(i) &= (T_\mu J')(i) + \sum_{j=1}^{n} p_{ij}\big(\mu(i)\big)\big(J(j) - J'(j)\big) \\ &= (T_\mu J')(i) + \sum_{j=1}^{n} p_{ij}\big(\mu(i)\big)v(j)\frac{\big(J(j) - J'(j)\big)}{v(j)} \\ &\le (T_\mu J')(i) + \sum_{j=1}^{n} p_{ij}\big(\mu(i)\big)v(j)\,\|J - J'\| \\ &\le (T_\mu J')(i) + \rho\, v(i)\,\|J - J'\|,\end{aligned}$$

where the last inequality follows from Eq. (4.97). By minimizing both sides over all $\mu(i) \in U(i)$, we obtain

$$(TJ)(i) \le (TJ')(i) + \rho\, v(i)\,\|J - J'\|, \qquad i = 1, \ldots, n.$$

Thus we have

$$\frac{(TJ)(i) - (TJ')(i)}{v(i)} \le \rho\,\|J - J'\|, \qquad i = 1, \ldots, n.$$

Similarly, by reversing the roles of J and J', we obtain

$$\frac{(TJ')(i) - (TJ)(i)}{v(i)} \le \rho\,\|J - J'\|, \qquad i = 1, \ldots, n.$$

By combining the preceding two inequalities, we have

$$\frac{\big|(TJ)(i) - (TJ')(i)\big|}{v(i)} \le \rho\,\|J - J'\|, \qquad i = 1, \ldots, n,$$

and by maximizing the left-hand side over i, the contraction property $\|TJ - TJ'\| \le \rho\|J - J'\|$ follows. **Q.E.D.**

4.13.2 Proofs for Discounted Problems

Since we have shown that the discounted problem can be converted to the equivalent SSP problem of Fig. 4.3.1, we can apply Props. 4.2.1-4.2.4. Then Props. 4.3.1-4.3.4 are obtained from the construction of Fig. 4.3.1. The contraction property of Prop. 4.3.5 can also be proved in the same way, since in the SSP problem of Fig. 4.3.1, the expected number of steps to terminate starting from a state $i \neq t$ can be obtained as the mean of a geometrically distributed random variable with parameter $1 - \alpha$:

$$v(i) = 1 \cdot (1 - \alpha) + 2 \cdot \alpha(1 - \alpha) + 3 \cdot \alpha^2(1 - \alpha) + \cdots = \frac{1}{1 - \alpha}, \quad i = 1, \ldots, n.$$

so that the modulus of contraction is

$$\rho = \frac{v(i) - 1}{v(i)} = \alpha.$$

Thus by applying Prop. 4.2.5, we obtain Prop. 4.3.5. Note that there is similar contraction property for T_μ.

4.13.3 Convergence of Exact and Optimistic Policy Iteration

We provide a proof of the convergence of exact PI for the case of a discounted problem. The proof for the SSP problem is similar.

> **Proposition 4.5.1: (Convergence of Exact PI)** For both the SSP and the discounted problems, the exact PI algorithm generates an improving sequence of policies [i.e., $J_{\mu^{k+1}}(i) \leq J_{\mu^k}(i)$ for all i and k] and terminates with an optimal policy.

Proof: For any k, consider the sequence generated by the VI algorithm for policy μ^{k+1}:

$$J_{N+1}(i) = \sum_{j=1}^{n} p_{ij}\big(\mu^{k+1}(i)\big)\Big(g\big(i, \mu^{k+1}(i), j\big) + \alpha J_N(j)\Big), \qquad i = 1, \ldots, n,$$

where $N = 0, 1, \ldots$, and

$$J_0(i) = J_{\mu^k}(i), \qquad i = 1, \ldots, n.$$

From Eqs. (4.34) and (4.33), we have

$$\begin{aligned} J_0(i) &= \sum_{j=1}^{n} p_{ij}\big(\mu^{k}(i)\big)\Big(g\big(i, \mu^{k}(i), j\big) + \alpha J_0(j)\Big) \\ &\geq \sum_{j=1}^{n} p_{ij}\big(\mu^{k+1}(i)\big)\Big(g\big(i, \mu^{k+1}(i), j\big) + \alpha J_0(j)\Big) \\ &= J_1(i), \end{aligned}$$

for all i. By using the above inequality we obtain

$$\begin{aligned} J_1(i) &= \sum_{j=1}^{n} p_{ij}\big(\mu^{k+1}(i)\big)\Big(g\big(i, \mu^{k+1}(i), j\big) + \alpha J_0(j)\Big) \\ &\geq \sum_{j=1}^{n} p_{ij}\big(\mu^{k+1}(i)\big)\Big(g\big(i, \mu^{k+1}(i), j\big) + \alpha J_1(j)\Big) \\ &= J_2(i), \end{aligned}$$

for all i, and by continuing similarly we have

$$J_0(i) \geq J_1(i) \geq \cdots \geq J_N(i) \geq J_{N+1}(i) \geq \cdots, \qquad i = 1, \ldots, n. \tag{4.98}$$

Since by Prop. 4.3.3, $J_N(i) \to J_{\mu^{k+1}}(i)$, we obtain $J_0(i) \geq J_{\mu^{k+1}}(i)$ or

$$J_{\mu^k}(i) \geq J_{\mu^{k+1}}(i), \qquad i = 1, \ldots, n, \quad k = 0, 1, \ldots.$$

Thus the sequence of generated policies is improving, and since the number of stationary policies is finite, we must after a finite number of iterations, say $k+1$, obtain $J_{\mu^k}(i) = J_{\mu^{k+1}}(i)$ for all i. Then we will have equality throughout in Eq. (4.98), which means that

$$J_{\mu^k}(i) = \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha J_{\mu^k}(j)\big), \qquad i = 1, \ldots, n.$$

Thus the costs $J_{\mu^k}(1), \ldots, J_{\mu^k}(n)$ solve Bellman's equation, and by Prop. 4.3.2, it follows that $J_{\mu^k}(i) = J^*(i)$ and that μ^k is optimal. **Q.E.D.**

We provide a proof of convergence of optimistic PI for discounted problems.

Proposition 4.5.2: (Convergence of Optimistic PI) For the discounted problem, the sequences $\{J_k\}$ and $\{\mu^k\}$ generated by the optimistic PI algorithm satisfy

$$J_k \to J^*, \qquad J_{\mu^k} \to J^*.$$

Proof: First we choose a scalar r such that the vector $\bar{J}_0$ defined by $\bar{J}_0 = J_0 + r\,e$, satisfies $T\bar{J}_0 \leq \bar{J}_0$ [here and later, e is the unit vector, i.e., $e(i) = 1$ for all i]. This can be done since if r is such that $TJ_0 - J_0 \leq (1-\alpha)r\,e$, we have

$$T\bar{J}_0 = TJ_0 + \alpha r\,e \leq J_0 + r\,e = \bar{J}_0,$$

where $e = (1, 1, \ldots, 1)'$ is the unit vector.

With $\bar{J}_0$ so chosen, define for all k, $\bar{J}_{k+1} = T_{\mu^k}^{m_k}\bar{J}_k$. Then since we have

$$T(J + re) = TJ + \alpha r\,e, \qquad T_\mu(J + re) = T_\mu + \alpha r\,e$$

for any J and μ, it can be seen by induction that for all k and $m = 0, 1, \ldots, m_k$, the vectors $J_{k+1} = T_{\mu^k}^m J_k$ and $\bar{J}_{k+1} = T_{\mu^k}^m \bar{J}_k$ differ by a multiple of the unit vector, namely

$$r\alpha^{m_0 + \cdots + m_{k-1} + m} e.$$

It follows that if J_0 is replaced by $\bar{J}_0$ as the starting vector in the algorithm, the same sequence of policies $\{\mu^k\}$ will be obtained; i.e., for all k, we have $T_{\mu^k}\bar{J}_k = T\bar{J}_k$. Moreover, we have $\lim_{k\to\infty}(\bar{J}_k - J_k) = 0$.

Next we will show that $J^* \leq \bar{J}_k \leq T^k \bar{J}_0$ for all k, from which convergence will follow. Indeed, we have $T_{\mu^0} \bar{J}_0 = T\bar{J}_0 \leq \bar{J}_0$, from which we obtain

$$T^m_{\mu^0} \bar{J}_0 \leq T^{m-1}_{\mu^0} \bar{J}_0, \qquad m = 1, 2, \ldots,$$

so that

$$T_{\mu^1} \bar{J}_1 = T\bar{J}_1 \leq T_{\mu^0} \bar{J}_1 = T^{m_0+1}_{\mu^0} \bar{J}_0 \leq T^{m_0}_{\mu^0} \bar{J}_0 = \bar{J}_1 \leq T_{\mu^0} \bar{J}_0 = T\bar{J}_0.$$

This argument can be continued to show that for all k, we have $\bar{J}_k \leq T\bar{J}_{k-1}$, so that

$$\bar{J}_k \leq T^k \bar{J}_0, \qquad k = 0, 1, \ldots.$$

On the other hand, since $T\bar{J}_0 \leq \bar{J}_0$, we have $J^* \leq \bar{J}_0$, and it follows that successive application of any number of operators of the form T_μ to $\bar{J}_0$ produces functions that are bounded from below by J^*. Thus,

$$J^* \leq \bar{J}_k \leq T^k \bar{J}_0, \qquad k = 0, 1, \ldots.$$

By taking the limit as $k \to \infty$, we obtain $\lim_{k\to\infty} \bar{J}_k(i) = J^*(i)$ for all i, and since $\lim_{k\to\infty} (\bar{J}_k - J_k) = 0$, we obtain

$$\lim_{k\to\infty} J_k(i) = J^*(i), \qquad i = 1, \ldots, n.$$

Finally, from the finiteness of the state and control spaces, it follows that there exists $\epsilon > 0$ such that if $\max_i \big|J(i) - J^*(i)\big| \leq \epsilon$ and $T_\mu J = TJ$, so that μ is optimal. Since $J_k \to J^*$, this shows that μ^k is optimal for all sufficiently large k. **Q.E.D.**

4.13.4 Performance Bounds for One-Step Lookahead, Rollout, and Approximate Policy Iteration

We first prove the basic performance bounds for ℓ-step lookahead schemes and discounted problems.

Proposition 4.6.1: (Limited Lookahead Performance Bounds)

(a) Let $\tilde{\mu}$ be the ℓ-step lookahead policy corresponding to $\tilde{J}$. Then

$$\|J_{\tilde{\mu}} - J^*\| \leq \frac{2\alpha^\ell}{1-\alpha}\|\tilde{J} - J^*\|, \tag{4.99}$$

where $\|\cdot\|$ denotes the maximum norm (4.15).

(b) Let $\tilde{\mu}$ be the one-step lookahead policy obtained by minimization in the equation

$$\hat{J}(i) = \min_{u \in \overline{U}(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha\tilde{J}(j)\big), \qquad i = 1,\ldots,n, \tag{4.100}$$

where $\overline{U}(i) \subset U(i)$ for all $i = 1,\ldots,n$. Assume that for some constant c, we have

$$\hat{J}(i) \leq \tilde{J}(i) + c, \qquad i = 1,\ldots,n. \tag{4.101}$$

Then

$$J_{\tilde{\mu}}(i) \leq \tilde{J}(i) + \frac{c}{1-\alpha}, \qquad i = 1,\ldots,n. \tag{4.102}$$

Proof: (a) In the course of the proof, we will use the contraction property of T and T_μ (cf. Prop. 4.3.5). Using the triangle inequality, we write for every k,

$$\|T_{\tilde{\mu}}^k J^* - J^*\| \leq \sum_{m=1}^{k} \|T_{\tilde{\mu}}^m J^* - T_{\tilde{\mu}}^{m-1} J^*\| \leq \sum_{m=1}^{k} \alpha^{m-1}\|T_{\tilde{\mu}} J^* - J^*\|.$$

By taking the limit as $k \to \infty$ and using the fact $T_{\tilde{\mu}}^k J^* \to J_{\tilde{\mu}}$, we obtain

$$\|J_{\tilde{\mu}} - J^*\| \leq \frac{1}{1-\alpha}\|T_{\tilde{\mu}} J^* - J^*\|. \tag{4.103}$$

Denote $\hat{J} = T^{\ell-1}\tilde{J}$. The rightmost expression of Eq. (4.103) is estimated by using the triangle inequality and the fact $T_{\tilde{\mu}}\hat{J} = T\hat{J}$ as follows:

$$\begin{aligned}
\|T_{\tilde{\mu}} J^* - J^*\| &\leq \|T_{\tilde{\mu}} J^* - T_{\tilde{\mu}}\hat{J}\| + \|T_{\tilde{\mu}}\hat{J} - T\hat{J}\| + \|T\hat{J} - J^*\| \\
&= \|T_{\tilde{\mu}} J^* - T_{\tilde{\mu}}\hat{J}\| + \|T\hat{J} - TJ^*\| \\
&\leq 2\alpha\|\hat{J} - J^*\| \\
&= 2\alpha\|T^{\ell-1}\tilde{J} - T^{\ell-1}J^*\| \\
&\leq 2\alpha^\ell\|\tilde{J} - J^*\|.
\end{aligned}$$

By combining the preceding two relations, we obtain Eq. (4.99).

(b) Let us denote by e the unit vector whose components are all equal to 1. Then by assumption, we have

$$\tilde{J} + ce \geq \hat{J} = T_{\tilde{\mu}}\tilde{J}.$$

Applying $T_{\tilde{\mu}}$ to both sides of this relation, and using the monotonicity and constant shift property of $T_{\tilde{\mu}}$, we obtain

$$T_{\tilde{\mu}}\tilde{J} + \alpha ce \geq T_{\tilde{\mu}}^2\tilde{J}.$$

Continuing similarly, we have,

$$T_{\tilde{\mu}}^k\tilde{J} + \alpha^k ce \geq T_{\tilde{\mu}}^{k+1}\tilde{J}, \qquad k = 0, 1, \ldots.$$

Adding these relations, we obtain

$$\tilde{J} + (1 + \alpha + \cdots + \alpha^k)ce \geq T_{\tilde{\mu}}^{k+1}\tilde{J}, \qquad k = 0, 1, \ldots.$$

Taking the limit as $k \to \infty$, and using the fact $T_{\tilde{\mu}}^{k+1}\tilde{J} \to J_{\tilde{\mu}}$, we obtain the desired inequality (4.102). **Q.E.D.**

We next show the basic cost improvement property of rollout.

Proposition 4.6.2: (Cost Improvement by Rollout) Let $\tilde{\mu}$ be the rollout policy obtained by the one-step lookahead minimization

$$\min_{u \in \overline{U}(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha J_\mu(j)\big),$$

where μ is a base policy [cf. Eq. (4.100) with $\tilde{J} = J_\mu$] and we assume that $\mu(i) \in \overline{U}(i) \subset U(i)$ for all $i = 1, \ldots, n$. Then $J_{\tilde{\mu}} \leq J_\mu$.

Proof: Let us denote

$$\hat{J}(i) = \min_{u \in \overline{U}(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha J_\mu(j)\big).$$

We have for all $i = 1, \ldots, n$,

$$\hat{J}(i) \leq \sum_{j=1}^{n} p_{ij}(u)\big(g\big(i,\mu(i),j\big) + \alpha J_\mu(j)\big) = J_\mu(i),$$

where the equality on the right holds by Bellman's equation. Hence the hypothesis of Prop. 4.6.1(b) holds, and the result follows. **Q.E.D.**

We finally show the following performance bound for the rollout algorithm with cost function approximation.

Proposition 4.6.3: (Performance Bound of Rollout with Terminal Cost Function Approximation) Let ℓ and m be positive integers, let μ be a policy, and let $\tilde{J}$ be a function of the state. Consider a truncated rollout scheme consisting of ℓ-step lookahead, followed by rollout with a policy μ for m steps, and a terminal cost function approximation $\tilde{J}$ at the end of the m steps. Let $\tilde{\mu}$ be the policy generated by this scheme.

(a) We have

$$\|J_{\tilde{\mu}} - J^*\| \leq \frac{2\alpha^\ell}{1-\alpha}\|T_\mu^m \tilde{J} - J^*\|,$$

where T_μ is the DP operator of Eq. (4.14), and $\|\cdot\|$ denotes the maximum norm (4.15).

(b) Assume that for some constant c, $\tilde{J}$ and μ satisfy the condition

$$\hat{J}(i) = \sum_{j=1}^{n} p_{ij}\big(\mu(i)\big)\Big(g\big(i, \mu(i), j\big) + \alpha\tilde{J}(j)\Big) \leq \tilde{J}(i) + c, \tag{4.104}$$

for all $i = 1, \dots, n$. Then

$$J_{\tilde{\mu}}(i) \leq \tilde{J}(i) + \frac{c}{1-\alpha}, \qquad i = 1, \dots, n. \tag{4.105}$$

Proof: Part (a) is simply Prop. 4.6.1(a) adapted to the truncated rollout scheme, so we focus on the proof of part (b). We first prove the result for the case where $c = 0$. Then the condition (4.104) can be written as $\tilde{J} \geq T_\mu \tilde{J}$, from which by using the monotonicity of T and T_μ, we have

$$\tilde{J} \geq T_\mu^m \tilde{J} \geq T T_\mu^m \tilde{J} \geq T^{\ell-1} T_\mu^m \tilde{J} \geq T^\ell T_\mu^m \tilde{J} = T_{\tilde{\mu}} T^{\ell-1} T_\mu^m \tilde{J}, \tag{4.106}$$

so that

$$\tilde{J} \geq T^{\ell-1} T_\mu^m \tilde{J} \geq T_{\tilde{\mu}} T^{\ell-1} T_\mu^m \tilde{J}.$$

This relation and the monotonicity of T_μ, imply that $\{T_{\tilde{\mu}}^k T^{\ell-1} T_\mu^m \tilde{J}\}$ is monotonically nonincreasing as k increases, and is bounded above by $\tilde{J}$. Since by Prop. 4.3.3 (VI convergence), the sequence converges to $J_{\tilde{\mu}}$ as $k \to \infty$, the result follows.

To prove the result for general c, we introduce the function J' given by

$$J'(i) = \tilde{J}(i) + \frac{c}{1-\alpha}, \qquad i = 1, \ldots, n.$$

Then the condition (4.104) can be written in terms of J' as

$$\hat{J}(i) = \sum_{j=1}^{n} p_{ij}\big(\mu(i)\big)\Big(g\big(i, \mu(i), j\big) + \alpha J'(j) - \frac{\alpha c}{1-\alpha}\Big) \leq J'(i) - \frac{c}{1-\alpha} + c,$$

or equivalently as

$$\sum_{j=1}^{n} p_{ij}\big(\mu(i)\big)\Big(g\big(i, \mu(i), j\big) + \alpha J'(j)\Big) \leq J'(i).$$

Since adding a constant to the components of $\tilde{J}$ does not change $\tilde{\mu}$, we can replace $\tilde{J}$ with J', without changing $\tilde{\mu}$. Then by using the version of the result already proved, we have $J_{\tilde{\mu}} \leq J'$, which is equivalent to the desired relation (4.105). **Q.E.D.**

The preceding proof allows a relaxation of the condition (4.104). For the relation (4.106) to hold it is sufficient that $\tilde{J}$ and μ satisfy the condition

$$\tilde{J} \geq T_\mu^m \tilde{J} \geq T_\mu^{m+1} \tilde{J},$$

or the even weaker condition

$$\tilde{J} \geq T_\mu^m \tilde{J} \geq T T_\mu^m \tilde{J}.$$

There is also an extension of the preceding condition for the case where $m = 0$, i.e., there is no rollout. It takes the form

$$\tilde{J} \geq T\tilde{J},$$

and it implies the bound $J_{\tilde{\mu}} \leq \tilde{J}$. The proof is based on Eq. (4.106) where m is taken to be zero. In domain-specific contexts, the preceding conditions may be translated into meaningful results.

To prove the performance bound of Prop. 4.6.4, we focus on the discounted problem, and we make use of the contraction property of Prop. 4.3.5:

$$\|T_\mu J - T_\mu J'\| \leq \alpha \|J - J'\|, \qquad \|TJ - TJ'\| \leq \alpha \|J - J'\|,$$

for all J, J', and μ, where $\|J\|$ is the maximum norm

$$\|J\| = \max_{i=1,\ldots,n} \big|J(i)\big|.$$

We want to prove the following performance bound.

Proposition 4.6.4: (Performance Bound for Approximate PI) Consider the discounted problem, and let $\{\mu^k\}$ be the sequence generated by the approximate PI algorithm defined by the approximate policy evaluation (4.44) and the approximate policy improvement (4.45). Then we have

$$\limsup_{k\to\infty} \|J_{\mu^k} - J^*\| \le \frac{\epsilon + 2\alpha\delta}{(1-\alpha)^2}.$$

The essence of the proof is contained in the following lemma, which quantifies the amount of approximate policy improvement at each iteration.

Lemma 4.13.1: Consider the discounted problem, and let J, $\tilde{\mu}$, and μ satisfy

$$\|J - J_\mu\| \le \delta, \qquad \|T_{\tilde{\mu}}J - TJ\| \le \epsilon, \tag{4.107}$$

where δ and ϵ are some scalars. Then

$$\|J_{\tilde{\mu}} - J^*\| \le \alpha\|J_\mu - J^*\| + \frac{\epsilon + 2\alpha\delta}{1-\alpha}. \tag{4.108}$$

Proof: The contraction property of T and $T_{\tilde{\mu}}$ implies that

$$\|T_{\tilde{\mu}}J_\mu - T_{\tilde{\mu}}J\| \le \alpha\delta, \qquad \|TJ - TJ_\mu\| \le \alpha\delta,$$

and hence

$$T_{\tilde{\mu}}J_\mu \le T_{\tilde{\mu}}J + \alpha\delta e, \qquad TJ \le TJ_\mu + \alpha\delta e,$$

where e is the unit vector, i.e., $e(i) = 1$ for all i. Using also Eq. (4.107), we have

$$T_{\tilde{\mu}}J_\mu \le T_{\tilde{\mu}}J + \alpha\delta\, e \le TJ + (\epsilon + \alpha\delta)e \le TJ_\mu + (\epsilon + 2\alpha)e. \tag{4.109}$$

Combining this inequality with $TJ_\mu \le T_\mu J_\mu = J_\mu$, we obtain

$$T_{\tilde{\mu}}J_\mu \le J_\mu + (\epsilon + 2\alpha\delta)e. \tag{4.110}$$

We will show that this relation implies that

$$J_{\tilde{\mu}} \le J_\mu + \frac{\epsilon + 2\alpha\delta}{1-\alpha}e. \tag{4.111}$$

Indeed, by applying $T_{\tilde{\mu}}$ to both sides of Eq. (4.110), we obtain

$$T_{\tilde{\mu}}^2 J_\mu \le T_{\tilde{\mu}} J_\mu + \alpha(\epsilon + 2\alpha\delta)e \le J_\mu + (1+\alpha)(\epsilon + 2\alpha\delta)e.$$

Applying $T_{\tilde{\mu}}$ again to both sides of this relation, and continuing similarly, we have for all k,

$$T_{\tilde{\mu}}^k J_\mu \le J_\mu + (1 + \alpha + \cdots + \alpha^{k-1})(\epsilon + 2\alpha\delta)e.$$

By taking the limit as $k \to \infty$, and by using the VI convergence property $T_{\tilde{\mu}}^k J_\mu \to J_{\tilde{\mu}}$, we obtain Eq. (4.111).

Using now the contraction property of $T_{\tilde{\mu}}$ and Eq. (4.111), we have

$$J_{\tilde{\mu}} = T_{\tilde{\mu}} J_{\tilde{\mu}} = T_{\tilde{\mu}} J_\mu + (T_{\tilde{\mu}} J_{\tilde{\mu}} - T_{\tilde{\mu}} J_\mu) \le T_{\tilde{\mu}} J_\mu + \frac{\alpha(\epsilon + 2\alpha\delta)}{1-\alpha} e.$$

Subtracting J^* from both sides, we obtain

$$J_{\tilde{\mu}} - J^* \le T_{\tilde{\mu}} J_\mu - J^* + \frac{\alpha(\epsilon + 2\alpha\delta)}{1-\alpha} e. \tag{4.112}$$

Also from the contraction property of T,

$$TJ_\mu - J^* = TJ_\mu - TJ^* \le \alpha \|J_\mu - J^*\| e$$

which, in conjunction with Eq. (4.109), yields

$$T_{\tilde{\mu}} J_\mu - J^* \le TJ_\mu - J^* + (\epsilon + 2\alpha\delta)e \le \alpha \|J_\mu - J^*\| + (\epsilon + 2\alpha\delta)e.$$

Combining this relation with Eq. (4.112), we obtain

$$J_{\tilde{\mu}} - J^* \le \alpha \|J_\mu - J^*\|\, e + \frac{\alpha(\epsilon + 2\alpha\delta)}{1-\alpha} e + (\epsilon + 2\alpha\delta)e = \alpha \|J_\mu - J^*\|\, e + \frac{\epsilon + 2\alpha\delta}{1-\alpha} e,$$

which is equivalent to the desired relation (4.108). **Q.E.D.**

Proof of Prop. 4.6.4: Applying Lemma 4.13.1, we have

$$\|J_{\mu^{k+1}} - J^*\| \le \alpha \|J_{\mu^k} - J^*\| + \frac{\epsilon + 2\alpha\delta}{1-\alpha},$$

which by taking the lim sup of both sides as $k \to \infty$ yields the desired result. **Q.E.D.**

We next prove the performance bound for approximate PI, assuming that the generated policy sequence is convergent. For this proof we use the triangle inequality, which holds for any norm $\|\cdot\|$,

$$\|J + J'\| \le \|J\| + \|J'\|, \qquad \text{for all } J, J'.$$

Proposition 4.6.5: (Performance Bound for Approximate PI when Policies Converge) Let $\tilde{\mu}$ be a policy generated by the approximate PI algorithm under conditions (4.44), (4.45), and (4.46). Then we have

$$\max_{i=1,\dots,n} \big|J_{\tilde{\mu}}(i) - J^*(i)\big| \leq \frac{\epsilon + 2\alpha\delta}{1-\alpha}.$$

Proof: Let $\bar{J}$ be the cost vector obtained by approximate policy evaluation of $\tilde{\mu}$. Then in view of Eqs. (4.44), (4.45), we have

$$\|\bar{J} - J_{\tilde{\mu}}\| \leq \delta, \qquad \|T_{\tilde{\mu}}\bar{J} - T\bar{J}\| \leq \epsilon.$$

From this relation, the fact $J_{\tilde{\mu}} = T_{\tilde{\mu}}J_{\tilde{\mu}}$, and the triangle inequality, we have

$$\begin{aligned} \|TJ_{\tilde{\mu}} - J_{\tilde{\mu}}\| &\leq \|TJ_{\tilde{\mu}} - T\bar{J}\| + \|T\bar{J} - T_{\tilde{\mu}}\bar{J}\| + \|T_{\tilde{\mu}}\bar{J} - J_{\tilde{\mu}}\| \\ &= \|TJ_{\tilde{\mu}} - T\bar{J}\| + \|T\bar{J} - T_{\tilde{\mu}}\bar{J}\| + \|T_{\tilde{\mu}}\bar{J} - T_{\tilde{\mu}}J_{\tilde{\mu}}\| \\ &\leq \alpha\|J_{\tilde{\mu}} - \bar{J}\| + \epsilon + \alpha\|\bar{J} - J_{\tilde{\mu}}\| \\ &\leq \epsilon + 2\alpha\delta. \end{aligned} \tag{4.113}$$

For every k, by using repeatedly the triangle inequality and the contraction property of T, we have

$$\|T^kJ_{\tilde{\mu}} - J_{\tilde{\mu}}\| \leq \sum_{\ell=1}^{k} \|T^\ell J_{\tilde{\mu}} - T^{\ell-1}J_{\tilde{\mu}}\| \leq \sum_{\ell=1}^{k} \alpha^{\ell-1}\|TJ_{\tilde{\mu}} - J_{\tilde{\mu}}\|,$$

and by taking the limit as $k \to \infty$,

$$\|J^* - J_{\tilde{\mu}}\| \leq \frac{1}{1-\alpha}\|TJ_{\tilde{\mu}} - J_{\tilde{\mu}}\|.$$

Combining this relation with Eq. (4.113), we obtain the desired performance bound. **Q.E.D.**