A COURSE IN REINFORCEMENT LEARNING

Dimitri P. Bertsekas

Athena Scientific

A Course in Reinforcement Learning

by

Dimitri P. Bertsekas

Arizona State University

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

© 2023 Dimitri P. Bertsekas
All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

Publisher's Cataloging-in-Publication Data

Bertsekas, Dimitri P.
A Course in Reinforcement Learning
Includes Bibliography and Index
1. Mathematical Optimization. 2. Dynamic Programming. I. Title.
QA402.5 .B465 2020 519.703 00-91281

ISBN-10: 1-886529-49-3, ISBN-13: 978-1-886529-49-6

Latest editorial and other changes added on 4/17/2024

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. From 1979 to 2019 he was a professor at the Electrical Engineering and Computer Science Department of the Massachusetts Institute of Technology (M.I.T.), and he still holds the title of McAfee Professor of Engineering. In 2019, he joined the School of Computing and Augmented Intelligence at the Arizona State University, Tempe, AZ, as Fulton Professor of Computational Decision Making.

Professor Bertsekas' teaching and research have spanned several fields, including deterministic optimization, dynamic programming and stochastic control, large-scale and distributed computation, artificial intelligence, and data communication networks. He has authored or coauthored numerous research papers and twenty 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." At ASU, he has been focusing in teaching and research in reinforcement learning, and he has written several textbooks and research monographs in this field since 2019.

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 2022 IEEE Control Systems Award. In 2018 he shared with his coauthor, John Tsitsiklis, the 2018 INFORMS John von Neumann Theory Prize for the contributions of the research monographs "Parallel and Distributed Computation" and "Neuro-Dynamic Programming." Professor Bertsekas was elected in 2001 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. A Course in Reinforcement Learning by Dimitri P. Bertsekas, 2023, ISBN 978-1-886529-49-6, 424 pages
2. Lessons from AlphaZero for Optimal, Model Predictive, and Adaptive Control by Dimitri P. Bertsekas, 2022, ISBN 978-1-886529-17-5, 245 pages
3. Abstract Dynamic Programming, 3rd Edition, by Dimitri P. Bertsekas, 2022, ISBN 978-1-886529-47-2, 420 pages
4. Rollout, Policy Iteration, and Distributed Reinforcement Learning, by Dimitri P. Bertsekas, 2020, ISBN 978-1-886529-07-6, 480 pages
5. Reinforcement Learning and Optimal Control, by Dimitri P. Bertsekas, 2019, ISBN 978-1-886529-39-7, 388 pages
6. Dynamic Programming and Optimal Control, Two-Volume Set, by Dimitri P. Bertsekas, 2017, ISBN 1-886529-08-6, 1270 pages
7. Nonlinear Programming, 3rd Edition, by Dimitri P. Bertsekas, 2016, ISBN 1-886529-05-1, 880 pages
8. Convex Optimization Algorithms, by Dimitri P. Bertsekas, 2015, ISBN 978-1-886529-28-1, 576 pages
9. Convex Optimization Theory, by Dimitri P. Bertsekas, 2009, ISBN 978-1-886529-31-1, 256 pages
10. Introduction to Probability, 2nd Edition, by Dimitri P. Bertsekas and John N. Tsitsiklis, 2008, ISBN 978-1-886529-23-6, 544 pages
11. Convex Analysis and Optimization, by Dimitri P. Bertsekas, Angelia Nedić, and Asuman E. Ozdaglar, 2003, ISBN 1-886529-45-0, 560 pages
12. Network Optimization: Continuous and Discrete Models, by Dimitri P. Bertsekas, 1998, ISBN 1-886529-02-7, 608 pages
13. Network Flows and Monotropic Optimization, by R. Tyrrell Rockafellar, 1998, ISBN 1-886529-06-X, 634 pages
14. Introduction to Linear Optimization, by Dimitris Bertsimas and John N. Tsitsiklis, 1997, ISBN 1-886529-19-1, 608 pages
15. Parallel and Distributed Computation: Numerical Methods, by Dimitri P. Bertsekas and John N. Tsitsiklis, 1997, ISBN 1-886529-01-9, 718 pages
16. Neuro-Dynamic Programming, by Dimitri P. Bertsekas and John N. Tsitsiklis, 1996, ISBN 1-886529-10-8, 512 pages
17. Constrained Optimization and Lagrange Multiplier Methods, by Dimitri P. Bertsekas, 1996, ISBN 1-886529-04-3, 410 pages
18. Stochastic Optimal Control: The Discrete-Time Case, by Dimitri P. Bertsekas and Steven E. Shreve, 1996, ISBN 1-886529-03-5, 330 pages

CONTENTS

2. Approximation in Value Space - Rollout Algorithms

Preface

It is a capital mistake to theorize before one has data.
Sherlock Holmes

We are surrounded by data, but starved for insights.
Jay Baer

This book is based on lecture notes prepared for use in the 2023 ASU research-oriented course on Reinforcement Learning (RL) that I have offered in each of the last five years, as the field was rapidly evolving. The purpose of the book is to give an overview of the RL methodology, particularly as it relates to problems of optimal and suboptimal control, as well as discrete optimization. More broadly, we will aim to provide a framework for structured thinking about RL and its connections to the decision and control methodology, which is couched on, but is not dominated by mathematics.

Generally, RL can be viewed as the art and science of sequential decision making for large and difficult problems, often in the presence of imprecisely known and changing environment conditions. Dynamic Programming (DP) is a broad and well-established algorithmic methodology for making optimal sequential decisions, and is the theoretical foundation upon which RL rests. This is unlikely to change in the future, despite the rapid pace of technological innovation. In fact, there are strong connections between sequential decision making and the new wave of technological change, generative technology, transformers, GPT applications, and natural language processing ideas, as we will aim to show in this book.

In DP there are two principal objects to compute: the *optimal value function* that provides the optimal cost that can be attained starting from any given initial state, and the *optimal policy* that provides the optimal decision to apply at any given state and time. Unfortunately, the exact application of DP runs into formidable computational difficulties, commonly referred to as the *curse of dimensionality*. To address these, RL aims to approximate the optimal value function and policy, by using manageable off-line and/or on-line computation, which often involves neural networks (hence the alternative name Neuro-Dynamic Programming [BeT96]).

Thus there are two major methodological approaches in RL: *approximation in value space*, where we approximate in some way the optimal value function, and *approximation in policy space*, whereby we construct a suboptimal policy by using some form of optimization over a suitably restricted class of policies. In some schemes these approximation approaches may

be combined, aiming to capitalize on the advantages of both. Generally, approximation in value space is tied more closely to the central DP ideas of value and policy iteration than approximation in policy space, which in practice often relies on gradient-like descent, a more broadly applicable optimization methodology.

The book focuses primarily on approximation in value space, with limited coverage of approximation in policy space. However, it is structured so that it can be easily supplemented by an instructor who wishes to go into approximation in policy space in greater detail, using any of a number of available sources.

An important part of our line of development is a new conceptual framework, which aims to bridge the gaps between the artificial intelligence, control theory, and operations research views of our subject. This framework, the focus of the author's recent monograph "Lessons from AlphaZero ...", [Ber22a], centers on approximate forms of DP that are inspired by some of the major successes of RL involving games. Primary examples are the recent (2017) AlphaZero program (which plays chess), and the similarly structured and earlier (1990s) TD-Gammon program (which plays backgammon).

Our framework is couched on two general algorithms that are designed largely independently of each other and operate in synergy through the powerful mechanism of Newton's method, applied to the fundamental Bellman equation of DP. We call these the *off-line training* and the *on-line play* algorithms. In the AlphaZero and TD-Gammon game contexts, the off-line training algorithm is the method used to teach the program how to evaluate positions and to generate good moves at any given position, while the on-line play algorithm is the method used to play in real time against human or computer opponents.

Our synergistic view of off-line training and on-line play is motivated by some striking empirical observations. In particular, both AlphaZero and TD-Gammon were trained off-line extensively using neural networks and an approximate version of the fundamental DP algorithm of policy iteration. Yet the AlphaZero player that was obtained off-line is not used directly during on-line play (it is too inaccurate due to approximation errors that are inherent in off-line neural network training). Instead, a separate on-line player is used to select moves, based on multistep lookahead minimization and a terminal position evaluator that was trained using experience with the off-line player. The on-line player performs a form of policy improvement, which is not degraded by neural network approximations. As a result, it greatly improves the performance of the off-line player.

Similarly, TD-Gammon performs on-line a policy improvement step using one-step or two-step lookahead minimization, which is not degraded by neural network approximations. To this end, it uses an off-line neural network-trained terminal position evaluator, and importantly it also extends its on-line lookahead by rollout (simulation with the one-step looka-

head player that is based on the position evaluator). Thus in summary:

(a) The on-line player of AlphaZero plays much better than its extensively trained off-line player. This is due to the beneficial effect of exact policy improvement with long lookahead minimization, which corrects for the inevitable imperfections of the neural network-trained off-line player, and position evaluator/terminal cost approximation.

(b) The TD-Gammon player that uses long rollout plays much better than TD-Gammon without rollout. This is due to the beneficial effect of the rollout, which serves as a supplement and substitute for long lookahead minimization.

An important lesson from AlphaZero and TD-Gammon is that the performance of an off-line trained policy can be greatly improved by on-line approximation in value space, with long lookahead (involving minimization or rollout with the off-line policy, or both), and terminal cost approximation that is obtained off-line. This performance enhancement is often dramatic and is due to a simple fact, which is couched on algorithmic mathematics and is a focal point of our course: *approximation in value space with one-step lookahead minimization amounts to a step of Newton's method for solving Bellman's equation, while the starting point for the Newton step is based on the results of off-line training, and may be enhanced by longer lookahead minimization and on-line rollout*. Indeed the major determinant of the quality of the on-line policy is the Newton step that is performed on-line, while off-line training plays a secondary role by comparison.

Significantly, the synergy between off-line training and on-line play also underlies Model Predictive Control (MPC), a major control system design methodology that has been extensively developed since the 1980s. This synergy is evident in the context of our narrative and helps to explain the all-important stability issues within the MPC context.

An additional benefit of policy improvement by approximation in value space, not observed in the context of games (which have stable rules and environment), is that it works well with changing problem parameters and on-line replanning, similar to the methodology of indirect adaptive control. In particular, the Bellman equation is perturbed due to the parameter changes, but approximation in value space still operates as a Newton step. An essential requirement here is that a system model is estimated on-line through some identification method, and is used during the one-step or multistep lookahead minimization process.

In this book, we will describe the basic RL methodologies, and we will aim to explain (often with visualization) their effectiveness (or lack thereof) in the context of the synergy between off-line training and on-line play. In the process, we will bring out the strong connections between the artificial intelligence view of RL, the control theory views of MPC and adaptive control, and the operations research view of discrete optimization

algorithms. Moreover, we will describe a broad variety of algorithms (especially truncated rollout, but also other methods) that can be used for on-line play.

In particular, we will aim to show that the methodology of approximation in value space and rollout applies very broadly to deterministic and stochastic optimal control problems, involving both discrete and continuous search spaces, as well as finite and infinite horizon. We will also show that in addition to MPC and adaptive control, our conceptual framework can be effectively integrated with other important methodologies such as multiagent systems and decentralized control, discrete and Bayesian optimization, and heuristic algorithms for discrete optimization.

Supporting Literature

In this book, we will deemphasize mathematical proofs, and focus instead on visualizations and intuitive (but still rigorous) explanations. However, there is considerable related analysis, which supports our narrative, and can be found within a broad range of sources, which we describe next.

The author's approximate DP/RL books

[1] Bertsekas, D. P., 2019. Reinforcement Learning and Optimal Control, Athena Scientific, Belmont, MA.

[2] Bertsekas, D. P., 2020. Rollout, Policy Iteration, and Distributed Reinforcement Learning, Athena Scientific, Belmont, MA.

provide a far more detailed discussion of MPC, adaptive control, discrete optimization, and distributed computation topics, than the present book. Moreover, some other popular methods, such as temporal difference algorithms and Q-learning, are discussed in the books [1] and [2], but not in the present book.

The author's two-volume DP book

[3] Bertsekas, D. P., 2017. Dynamic Programming and Optimal Control, Vol. I, 4th Edition, Athena Scientific, Belmont, MA.

[4] Bertsekas, D. P., 2012. Dynamic Programming and Optimal Control, Vol. II, 4th Edition, Athena Scientific, Belmont, MA.

is a major source on the modeling and mathematical aspects of finite and infinite horizon DP. Modeling aspects and finite horizon problems are the principal focus of [3], while the mathematical aspects of infinite horizon problems are the principal focus of [4]. Both books [3] and [4] provide substantial accounts of approximate DP/RL methods. Thus these books are the best entry points for a research-oriented reader that wishes to go into DP and its connections to RL more deeply.

Two of the author's recent research monographs are also highly relevant to our narrative:

[5] Bertsekas, D. P., 2022. Abstract Dynamic Programming, 3rd Ed., Athena Scientific, Belmont, MA (can be downloaded from the author's website).

This monograph focuses on the analytical aspects of abstract DP on which the Newton-based methodology is couched, and may serve as a mathematical supplement to the present book. It also provides some supportive mathematical foundation for the more visually oriented monograph

[6] Bertsekas, D. P., 2022. Lessons from AlphaZero for Optimal, Model Predictive, and Adaptive Control, Athena Scientific, Belmont, MA (can be downloaded from the author's website).

This monograph focuses in greater detail than the present book on the off-line training/on-line play/Newton's method conceptual framework, as well as on model predictive and adaptive control, and associated issues of stability. Like our Section 1.5, it is visually oriented, but goes into greater detail into various special cases, including pathological exceptions.

All of the above books are available as ebooks as well as in print form; see the Athena Scientific website. A lot of the material in this book is adapted from these books. However, the books themselves collectively provide a presentation that is far more detailed and mathematically rigorous.

An extensive overview on the connections and applications of the conceptual framework of this book with model predictive and adaptive control is given in the paper

Bertsekas, D. P., 2024. "Model Predictive Control, and Reinforcement Learning: A Unified Framework Based on Dynamic Programming," to be published in Proc. IFAC NMPC (can be downloaded from the author's website).

Together with Chapter 1 of the present book, it can be used as a starting point for a course in modern control system design.

The present book can also be fruitfully supplemented by the extensive textbook and research monograph literature on RL. This literature is summarized in Section 1.8, and includes several accounts of RL that are based on alternative viewpoints of artificial intelligence, control theory, and operations research.

Structure of the Book - Course Adaptations

An important structural characteristic of this book is that it is organized in a modular way, with a view towards flexibility, so it can be easily modified to accommodate changes in course content. In particular, the book is divided in two parts:

(1) A *foundational platform*, which consists of Chapter 1. It contains a selective overview of the approximate DP/RL landscape, and a

starting point for a more detailed in-class development of other RL topics, whose choice can be at the instructor's discretion.

(2) An *in-depth coverage* of the methodologies of deterministic and stochastic rollout in Chapter 2, and of the use of neural networks and other approximation architectures for off-line training in Chapter 3.

In a different course, alternative choices for in-depth coverage may be made, using the same foundational platform. In particular, both more and less mathematically-oriented courses can be organized around the platform of Chapter 1.

Let us also mention that the book contains more material than can be reasonably covered in class in one semester. This provides some flexibility to the instructor regarding the choice of material to present.

Videolectures and Slides

The present book and my RL books above, were developed while teaching several versions of my course at ASU over the last four years. Videolectures and slides from this course, as well as links to overview videolectures by the author are available from my website

http://web.mit.edu/dimitrib/www/RLbook.html

and provide a good supplement and companion resource to this book.

Thanks and Appreciation

The hospitable and stimulating environment at ASU contributed much to shaping my course during the period 2019-2023. I am very thankful to my ASU colleagues and the students in my classes. Special thanks go to my teaching assistants, Sushmita Bhattacharya, Sahil Badyal, and Jamison Weber. I have also appreciated fruitful interactions with several colleagues and students outside ASU, particularly Yuchao Li, who additionally provided valuable proofreading support.

1

Exact and Approximate Dynamic Programming

Contents

This chapter has multiple purposes:

(a) *To provide an overview of the exact dynamic programming (DP) methodology, with a view towards suboptimal solution methods.* We will first discuss finite horizon problems, which involve a finite sequence of successive decisions, and are thus conceptually and analytically simpler. We will consider separately deterministic and stochastic finite horizon problems (Sections 1.2 and 1.3, respectively). The reason is that deterministic problems are simpler and have some favorable characteristics, which allow the application of a broader variety of methods. Significantly they include challenging discrete and combinatorial optimization problems, which can be fruitfully addressed with some of the reinforcement learning (RL) methods that are the main subject of the book. We will also discuss somewhat briefly the more intricate infinite horizon methodology (Section 1.4), and refer to the author's DP textbooks [Ber12], [Ber17a], the RL books [Ber19a], [Ber20a], and the neuro-dynamic programming monograph [BeT96] for a fuller presentation.

(b) *To discuss in summary the principal RL methodologies, with primary emphasis on approximation in value space.* This is the architecture that underlies the AlphaZero, AlphaGo, TD-Gammon and other related programs, as well as the Model Predictive Control (MPC) methodology, one of the principal control system design methods. We will also argue later (Chapter 2) that approximation in value space provides the entry point for the use of RL methods for solving discrete optimization and integer programming problems.

(c) *To explain the major principles of approximation in value space, and its division into the off-line training and the on-line play algorithms.* A key idea here is the connection of these two algorithms through the algorithmic methodology of Newton's method for solving the problem's Bellman equation. This viewpoint, recently developed in the author's "Rollout and Policy Iteration ..." book [Ber20a] and the visually oriented "Lessons from AlphaZero ..." monograph [Ber22a], underlies the entire course and is discussed for the simple, intuitive, and important class of linear quadratic problems in Section 1.5.

(d) *To overview the range of problem types where our RL methods apply, and to explain some of their major algorithmic ideas (Section 1.6).* Included here are partial state observation problems (POMDP), multiagent problems, and problems with unknown model parameters, which can be addressed with adaptive control methods.

We will also discuss selectively in this chapter some major algorithmic topics in approximate DP and RL, including rollout and policy iteration. A broader discussion of DP/RL may be found in the RL books [Ber19a], [Ber20a], the DP textbooks [Ber12], [Ber17a], the neuro-dynamic program-

ming monograph [BeT96], as well as the textbook literature described in the last section of this chapter.

The present book reflects the author's decision/control and operations research orientation, which has in turn guided the choices of terminology, notation, and mathematical style for this book. On the other hand, RL methods have been developed within the artificial intelligence community, as well as the decision/control and operations research communities. While the underlying practical problems addressed by these communities are very similar in their mathematical structure, there are notable differences in terminology, notation, and culture, which can be quite bewildering to researchers entering the field. We have thus provided in Section 1.7 a glossary and an orientation to assist the reader in navigating the full range of the DP/RL literature.

1.1 ALPHAZERO, OFF-LINE TRAINING, AND ON-LINE PLAY

One of the most exciting recent success stories in RL is the development of the AlphaGo and AlphaZero programs by DeepMind Inc; see [SHM16], [SHS17], [SSS17]. AlphaZero plays Chess, Go, and other games, and is an improvement in terms of performance and generality over AlphaGo, which plays the game of Go only. Both programs play better than all competitor computer programs available in 2022, and much better than all humans. These programs are remarkable in several other ways. In particular, they have learned how to play without human instruction, just data generated by playing against themselves. Moreover, they learned how to play very quickly. In fact, AlphaZero learned how to play chess better than all humans and computer programs within hours (with the help of awesome parallel computation power, it must be said).

Perhaps the most impressive aspect of AlphaZero/chess is that its play is not just better, but it is also very different than human play in terms of long term strategic vision. Remarkably, AlphaZero has discovered new ways to play a game that has been studied intensively by humans for hundreds of years. Still, for all of its impressive success and brilliant implementation, AlphaZero is couched on well established theory and methodology, which is the subject of the present book, and is portable to far broader realms of engineering, economics, and other fields. This is the methodology of DP, policy iteration, limited lookahead, rollout, and approximation in value space.†

† It is also worth noting that the principles of the AlphaZero design have much in common with the work of Tesauro [Tes94], [Tes95], [TeG96] on computer backgammon. Tesauro's programs stimulated much interest in RL in the middle 1990s, and exhibit similarly different and better play than human backgammon players. A related impressive program for the (one-player) game

To understand the overall structure of AlphaZero, and its connection to our DP/RL methodology, it is useful to divide its design into two parts: *off-line training*, which is an algorithm that learns how to evaluate chess positions, and how to steer itself towards good positions with a default/base chess player, and *on-line play*, which is an algorithm that generates good moves in real time against a human or computer opponent, using the training it went through off-line. We will next briefly describe these algorithms, and relate them to DP concepts and principles.

Off-Line Training and Policy Iteration

This is the part of the program that learns how to play through off-line self-training, and is illustrated in Fig. 1.1.1. The algorithm generates a sequence of *chess players* and *position evaluators*. A chess player assigns "probabilities" to all possible moves at any given chess position (these are the probabilities with which the player selects the possible moves at the given position). A position evaluator assigns a numerical score to any given chess position (akin to a "probability" of winning the game from that position), and thus predicts quantitatively the performance of a player starting from any position. The chess player and the position evaluator are represented by two neural networks, a *policy network* and a *value network*, which accept a chess position and generate a set of move probabilities and a position evaluation, respectively.†

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

---

of Tetris, also based on the method of policy iteration, is described by Scherrer et al. [SGG15], who mention several related antecedent works. For a better understanding of the connections of AlphaZero and AlphaGo Zero with Tesauro's programs and the concepts developed here, the "Methods" section of the paper [SSS17] is recommended.

† Here the neural networks play the role of *function approximators*; see Chapter 3. By viewing a player as a function that assigns move probabilities to a position, and a position evaluator as a function that assigns a numerical score to a position, the policy and value networks provide approximations to these functions based on training with data (training algorithms for neural networks and other approximation architectures are also discussed in the RL books [Ber19a], [Ber20a], and the neuro-dynamic programming book [BeT96]).

‡ One more complication is that chess and Go are two-player games, while most of our development will involve single-player optimization. However, DP theory extends to two-player games, although we will not focus on this extension. Alternately, we can consider training a game program to play against a known fixed opponent; this is a one-player setting.

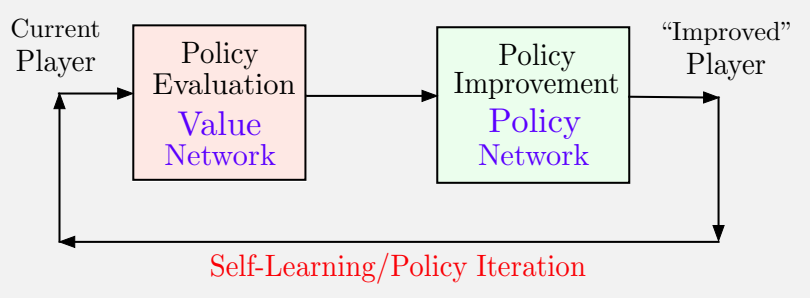

Figure 1.1.1 Illustration of the AlphaZero training algorithm. It generates a sequence of position evaluators and chess players. The position evaluator and the chess player are represented by two neural networks, a value network and a policy network, which accept a chess position and generate a position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

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

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

(b) *Policy improvement*: Given the current player and its position evaluator, trial move sequences are selected and evaluated for the remainder of the game starting from many positions. An improved player is then generated by adjusting the move probabilities of the current player towards the trial moves that have yielded the best results. In Alp-

† Quoting from the paper [SSS17]: "The AlphaGo Zero selfplay algorithm can similarly be understood as an approximate policy iteration scheme in which MCTS is used for both policy improvement and policy evaluation. Policy improvement starts with a neural network policy, executes an MCTS based on that policy's recommendations, and then projects the (much stronger) search policy back into the function space of the neural network. Policy evaluation is applied to the (much stronger) search policy: the outcomes of selfplay games are also projected back into the function space of the neural network. These projection steps are achieved by training the neural network parameters to match the search probabilities and selfplay game outcome respectively." Note, however, that a two-person game player, trained through selfplay, may fail against a particular human or computer player that can exploit training vulnerabilities. This is a theoretical but rare possibility; see our discussion in Section 2.12.

haZero this is done with a complicated algorithm called *Monte Carlo Tree Search*. However, policy improvement can also be done more simply. For example one could try all possible move sequences from a given position, extending forward to a given number of moves, and then evaluate the terminal position with the player's position evaluator. The move evaluations obtained in this way are used to nudge the move probabilities of the current player towards more successful moves, thereby obtaining data that is used to train a policy network that represents the new player.

Tesauro's TD-Gammon algorithm [Tes94] program is similarly based on approximate policy iteration, but uses a different methodology for approximate policy evaluation [it is based on the TD(λ) algorithm]; see the book [BeT96], Section 8.6, for a detailed description. Moreover, it does not use a policy network and MCTS. It involves only a value network, which replicates the functionality of a policy network by generating moves on-line via a one-step or two-step lookahead minimization.

On-Line Play and Approximation in Value Space - Rollout

Suppose that a "final" player has been obtained through the AlphaZero off-line training process just described. It could then be used in principle to play chess against any human or computer opponent, since it is capable of generating move probabilities at each given chess position using its policy network. In particular, during on-line play, at a given position the player can simply choose the move of highest probability supplied by the off-line trained policy network. This player would play very fast on-line, but it would not play good enough chess to beat strong human opponents. The extraordinary strength of AlphaZero is attained only after the player and its position evaluator obtained from off-line training have been embedded into another algorithm, which we refer to as the "on-line player." Given the policy network/player obtained off-line and its value network/position evaluator, this algorithm plays as follows (see Fig. 1.1.2).

At a given position, it generates a lookahead tree of all possible multiple move and countermove sequences, up to a given depth. It then runs the off-line obtained player for some more moves, and then evaluates the effect of the remaining moves by using the position evaluator of the off-line obtained value network. Actually the middle portion, called "truncated rollout," is not used in the published version of AlphaZero/chess [SHS17], [SHS17]; the first portion (multistep lookahead) is quite long and implemented efficiently, so that the rollout portion is not essential. Rollout is used in AlphaGo [SHM16], and plays a very important role the final version of Tesauro's backgammon program [TeG96]. The reason is that in backgammon, long multistep lookahead is not possible because of rapid expansion of the lookahead tree with every move.

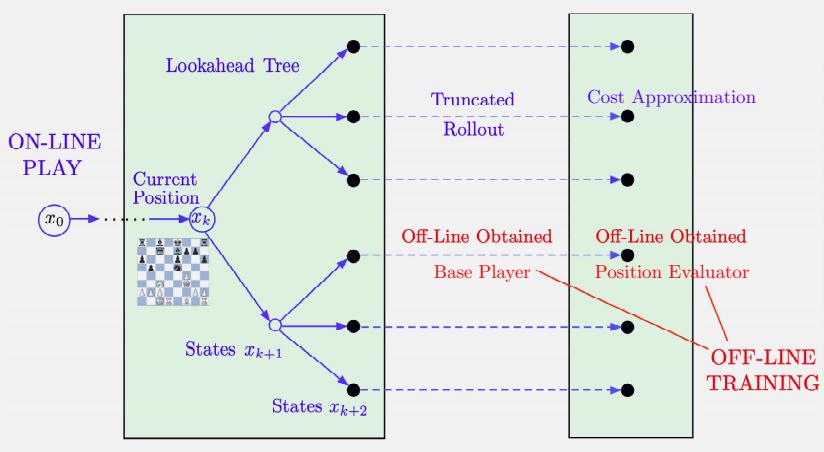

Figure 1.1.2 Illustration of an on-line player such as the one used in AlphaGo, AlphaZero, and Tesauro's backgammon program [TeG96]. At a given position, it generates a lookahead tree of multiple moves up to some depth, then runs the off-line obtained player for some more moves, and evaluates the effect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

We should note that the preceding description of AlphaZero and related games is oversimplified. We will be discussing refinements and details as the book progresses. However, DP ideas with cost function approximations, similar to the on-line player illustrated in Fig. 1.1.2, will be central. Moreover, the algorithmic division between off-line training and on-line policy implementation will be conceptually very important for our purposes in this book.

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

In control system design, similar architectures to the ones of AlphaZero and TD-Gammon are employed in model predictive control (MPC). There, the number of steps in lookahead minimization is called the *control interval*, while the total number of steps in lookahead minimization and truncated rollout is called the *prediction interval*; see e.g., Magni et al. [MDM01].† The benefit of truncated rollout in providing an economical substitute for longer lookahead minimization is well known within this

† The Matlab toolbox for MPC design explicitly allows the user to set these two intervals.

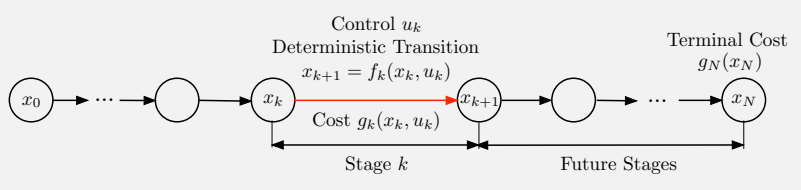

Figure 1.2.1 Illustration of a deterministic N-stage optimal control problem. Starting from state x_k, the next state under control u_k is generated nonrandomly, according to

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

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

context.

Dynamic programming frameworks with cost function approximations that are similar to the on-line player illustrated in Fig. 1.1.2, are also known as *approximate dynamic programming*, or *neuro-dynamic programming*, and will be central for our purposes. They will be generically referred to as *approximation in value space* in this book.†

1.2 DETERMINISTIC DYNAMIC PROGRAMMING

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

1.2.1 Finite Horizon Problem Formulation

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

† The names "approximate dynamic programming" and "neuro-dynamic programming" are often used as synonyms to RL. However, RL is generally thought to also subsume the methodology of approximation in policy space, which involves search for optimal parameters within a parametrized set of policies. The search is done with methods that are largely unrelated to DP, such as for example stochastic gradient or random search methods. Approximation in policy space may be used off-line to design a policy that can be used for on-line rollout. It will be discussed rather briefly here, but a fuller account that is consistent in terminology with the present book may be found in Chapter 5 of the RL book [Ber19a].

see Fig. 1.2.1. Thus, a deterministic DP problem involves a 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,

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

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

In the case of a finite number of states, the system function f_k may be represented by a table that gives the next state x_{k+1} for each possible value of the pair (x_k, u_k). Otherwise a mathematical expression or a computer implementation is necessary to represent f_k.

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

The problem also involves a cost function that is additive in the sense that the cost incurred at time k, denoted by $g_k(x_k, u_k)$, accumulates over time. Formally, g_k is a function of (x_k, u_k) that takes scalar 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

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

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

The generality of our DP model is also partly responsible for our choice of notation. In the artificial intelligence and operations research communities, finite state models, often referred to as Markovian Decision Problems (MDP), are common and use a transition probability notation (see Section 1.7.2). Unfortunately, this notation is not well suited for deterministic models, and also for continuous spaces models, both of which are important for the purposes of this book. For the latter models, it involves transition probability distributions over continuous spaces, and leads to mathematics that are far more complex as well as less intuitive than those based on the use of the system function (1.1).

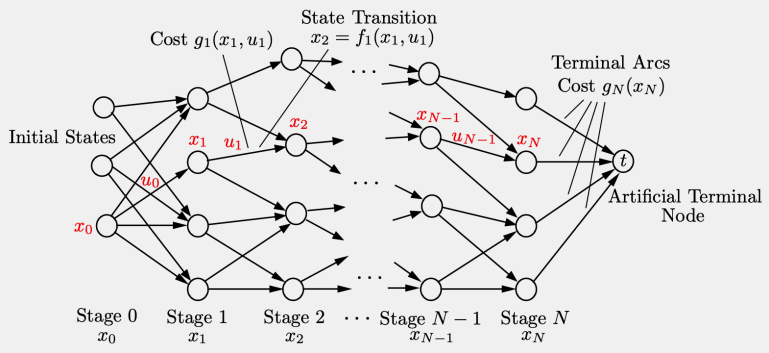

Figure 1.2.2 Transition graph for a deterministic finite-state system. Nodes correspond to states x_k. Arcs correspond to state-control pairs (x_k, u_k). An arc (x_k, u_k) has start and end nodes x_k and $x_{k+1} = f_k(x_k, u_k)$, respectively. We view the cost $g_k(x_k, u_k)$ of the transition as the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

where $g_N(x_N)$ is a terminal cost incurred at the end of the process. This is a well-defined scalar, 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 as a function of x_0:†

$$J^*(x_0) = \min_{\substack{u_k \in U_k(x_k) \\ k=0,\ldots,N-1}} J(x_0; u_0, \ldots, u_{N-1}). \tag{1.3}$$

Discrete Optimal Control Problems

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

Note that control sequences $\{u_0, \ldots, u_{N-1}\}$ correspond to paths originating at the initial state (a node at stage 0) and terminating at one of the

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

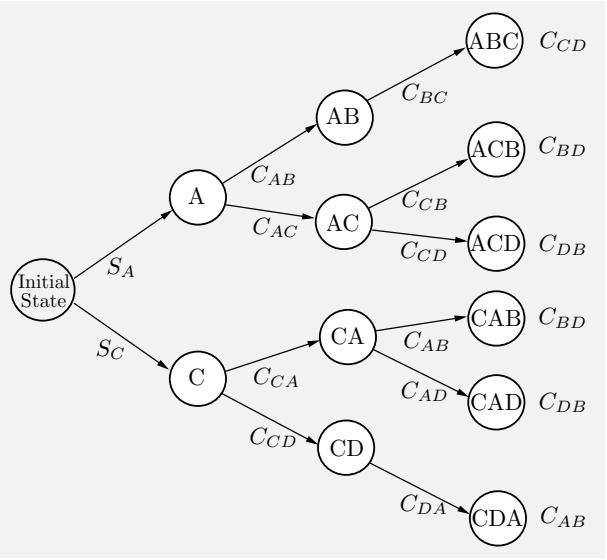

Figure 1.2.3 The transition graph of the deterministic scheduling problem of Example 1.2.1. Each arc of the graph corresponds to a decision leading from some state (the start node of the arc) to some other state (the end node of the arc). The corresponding cost is shown next to the arc. The cost of the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

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 nodes of the graph (stage 0) to the terminal node t*. Here, by the length of a path we mean the sum of the lengths of its arcs.†

Generally, combinatorial optimization problems can be formulated as deterministic finite-state finite-horizon optimal control problems. The idea is to break down the solution into components, which can be computed sequentially. The following is an illustrative example.

Example 1.2.1 (A Deterministic Scheduling Problem)

Suppose that to produce a certain product, four operations must be performed on a given 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

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

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. 1.2.3). 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. 1.2.3. Here the problem is deterministic, i.e., at a given state, each choice of control leads to a uniquely determined state. For example, at state AC the decision to perform operation D leads to state ACD with certainty, and has cost C_{CD}. Thus the problem can be conveniently represented with the transition graph of Fig. 1.2.3 (which in turn is a special case of the graph of Fig. 1.2.2). The optimal solution corresponds to the path that starts at the initial state and ends at some state at the terminal time and has minimum sum of arc costs plus the terminal cost.

1.2.2 The Dynamic Programming Algorithm

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

Principle of Optimality

Let $\{u_0^*, \ldots, u_{N-1}^*\}$ 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.

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

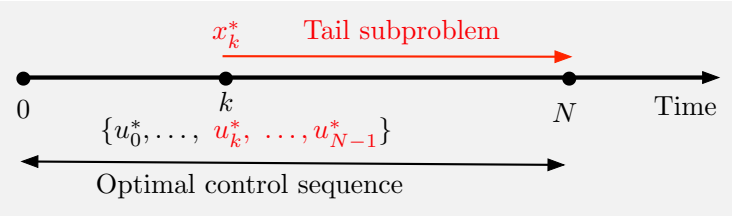

Figure 1.2.4 Schematic illustration of the principle of optimality. The tail $\{u_k^*, \ldots, u_{N-1}^*\}$ of an optimal sequence $\{u_0^*, \ldots, u_{N-1}^*\}$ is optimal for the tail subproblem that starts at the state x_k^* of the optimal state trajectory.

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

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

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

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

Example 1.2.1 (Scheduling Problem - Continued)

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

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

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

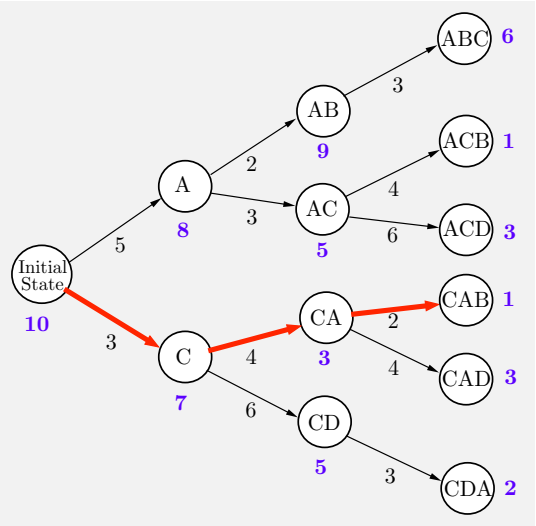

Figure 1.2.5 Transition graph of the deterministic scheduling problem, with the cost of each decision shown next to the corresponding arc. Next to each node/state we show the cost to optimally complete the schedule starting from that state. This is the optimal cost of the corresponding tail subproblem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

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

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

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

State AC: Here the possibilities are to (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. 1.2.5.

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. 1.2.5.

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

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

State A: Here the possibilities are to (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. 1.2.5.

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 C in Fig. 1.2.5.

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. 1.2.5.

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

Finding an Optimal Control Sequence by DP

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

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

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

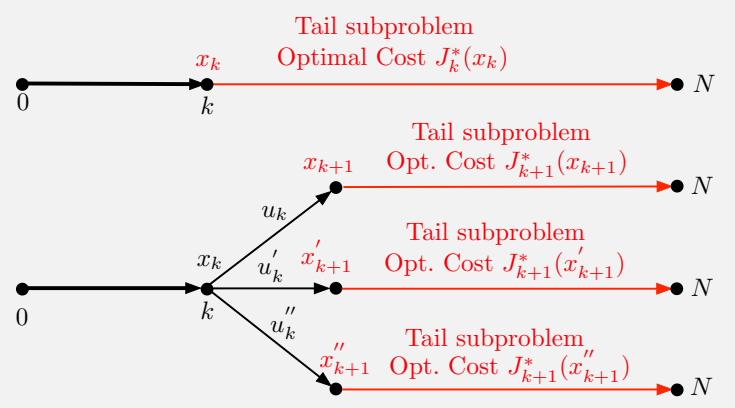

Figure 1.2.6 Illustration of the DP algorithm. The tail subproblem that starts at x_k at time k minimizes over $\{u_k, \ldots, u_{N-1}\}$ the "cost-to-go" from k to N,

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

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

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

DP Algorithm for Deterministic Finite Horizon Problems

Start with

$$J_N^*(x_N) = g_N(x_N), \qquad \text{for all } x_N, \tag{1.4}$$

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

$$J_k^*(x_k) = \min_{u_k \in U_k(x_k)} \Big[g_k(x_k, u_k) + J_{k+1}^*\big(f_k(x_k, u_k)\big) \Big], \qquad \text{for all } x_k. \tag{1.5}$$

The DP algorithm together with the construction of the functions $J_k^*(x_k)$ are illustrated in Fig. 1.2.6. Note that at stage k, the calculation in Eq. (1.5) 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

$$J_k^*(x_k) = \min_{\substack{u_m \in U_m(x_m) \\ m=k,\ldots,N-1}} J(x_k; u_k, \ldots, u_{N-1}), \tag{1.6}$$

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

$$J(x_k; u_k, \ldots, u_{N-1}) = g_N(x_N) + \sum_{t=k}^{N-1} g_t(x_t, u_t). \tag{1.7}$$

Thus, $J_k^*(x_k)$ is the optimal cost for an $(N-k)$-stage tail subproblem that starts at state x_k and time k, and ends at time N.† Based on the interpretation (1.6) of $J_k^*(x_k)$, we call it the *optimal cost-to-go* from state x_k at stage k, and refer to J_k^* as the *optimal cost-to-go function* or *optimal cost function* at time k. In maximization problems the DP algorithm (1.5) is written with maximization in place of minimization, and then J_k^* is referred to as the *optimal value function* at time k.

Once the functions $J_0^*, \ldots, J_N^*$ have been obtained, we can use a forward 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.

---

† 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. (1.6) and (1.7) to write

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

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

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

$$u_k^* \in \arg \min_{u_k \in U_k(x_k^*)} \Big[g_k(x_k^*, u_k) + J_{k+1}^*\big(f_k(x_k^*, u_k)\big) \Big], \qquad (1.8)$$

and

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

The same algorithm can be used to find an optimal control sequence for any tail subproblem. Figure 1.2.5 traces the calculations of the DP algorithm for the scheduling Example 1.2.1. The numbers next to the nodes, give the corresponding cost-to-go values, and the thick-line arcs give the construction of the optimal control sequence using the preceding algorithm.

The following example deals with the classical traveling salesman problem involving N cities. Here, the number of states grows exponentially with N, and so does the corresponding amount of computation for exact DP. We will show later that with rollout, we can solve the problem approximately with computation that grows polynomially with N.

Example 1.2.2 (The 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. 1.2.7). A k-city node/state leads to a $(k+1)$-city node/state by adding a new city at a cost equal to the travel time between the last two of the $k+1$ cities; see Fig. 1.2.7. Each sequence of N cities is connected to an artificial terminal node t with an arc of cost equal to the travel time from the last city of the sequence to the starting city, thus completing the transformation to a DP problem.

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 travel-

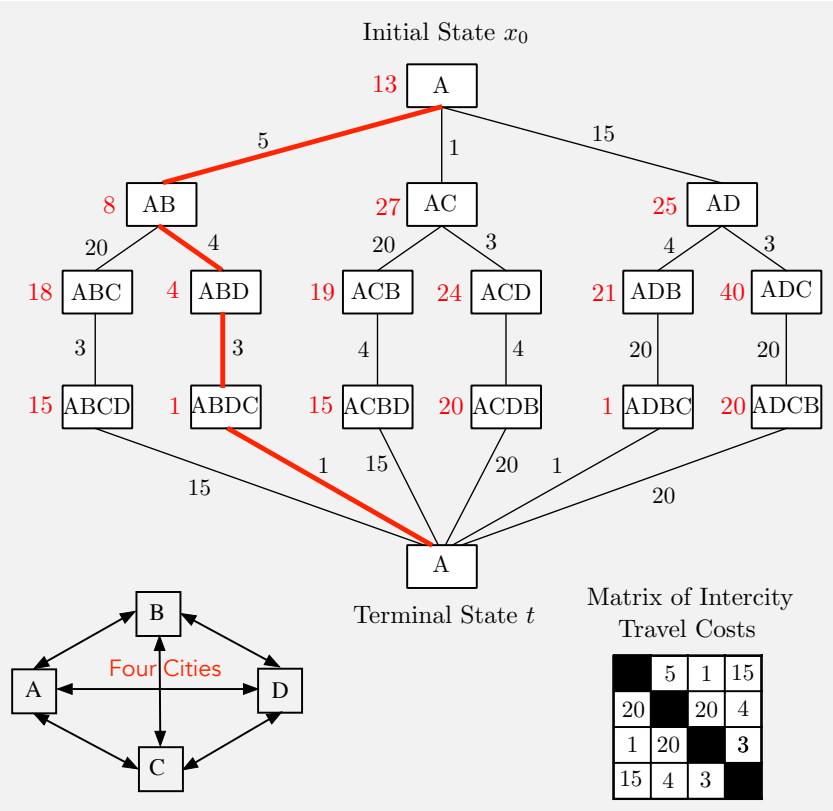

Figure 1.2.7 The DP formulation of the traveling salesman problem of Example 1.2.2. The travel times between the four cities A, B, C, and D are shown in the matrix at the bottom. We form a graph whose nodes are the k-city sequences and correspond to the states of the kth stage, assuming that A is the starting city. The transition costs/travel times are shown next to the arcs. The optimal costs-to-go are generated by DP starting from the terminal state and going backwards towards the initial state, and are shown next to the nodes. There is a unique optimal sequence here (ABDCA), and it is marked with thick lines. The optimal sequence can be obtained by forward minimization [cf. Eq. (1.8)], starting from the initial state x_0.

ing salesman and related scheduling problems are typically not addressed with exact DP, but rather with approximation methods. Some of these methods are based on DP and will be discussed later.

Q-Factors and Q-Learning

An alternative (and equivalent) form of the DP algorithm (1.5), uses the optimal cost-to-go functions J_k^* indirectly. In particular, it generates the

optimal Q-factors, defined for all pairs (x_k, u_k) and k by

$$Q_k^*(x_k, u_k) = g_k(x_k, u_k) + J_{k+1}^*\big(f_k(x_k, u_k)\big). \tag{1.9}$$

Thus the optimal Q-factors are simply the expressions that are minimized in the right-hand side of the DP equation (1.5).†

Note that the optimal cost function J_k^* can be recovered from the optimal Q-factor Q_k^* by means of the minimization

$$J_k^*(x_k) = \min_{u_k \in U_k(x_k)} Q_k^*(x_k, u_k). \tag{1.10}$$

Moreover, the DP algorithm (1.5) can be written in an essentially equivalent form that involves Q-factors only [cf. Eqs. (1.9)-(1.10)]:

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

Exact and approximate forms of this and other related algorithms, including counterparts for stochastic optimal control problems, comprise an important class of RL methods known as *Q-learning*.

1.2.3 Approximation in Value Space and Rollout

The forward optimal control sequence construction of Eq. (1.8) 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 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 an idea that is central in RL: *approximation in value space*.‡ It constructs a suboptimal solution $\{\tilde{u}_0, \ldots, \tilde{u}_{N-1}\}$ in place of the optimal $\{u_0^*, \ldots, u_{N-1}^*\}$, based on using $\tilde{J}_k$ in place of J_k^* in the DP algorithm (1.8).

† The term "Q-factor" has been used in the books [BeT96], [Ber19a], [Ber20a] and is adopted here as well. Another term used is "action value" (at a given state). The terms "state-action value" and "Q-value" are also common in the literature. The name "Q-factor" originated in reference to the notation used in an influential Ph.D. thesis [Wat89] that proposed the use of Q-factors in RL.

‡ Approximation in value space (sometimes called "search" or "tree search" in the AI literature) is a simple idea that has been used quite extensively for deterministic problems, well before the development of the modern RL methodology. For example it conceptually underlies the widely used A^* method for computing approximate solutions to large scale shortest path problems. For a view of A^* that is consistent with our approximate DP framework, the reader may consult the author's DP book [Ber17a].

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 $k = 1, 2, \ldots, N-1$, set

$$\tilde{u}_k \in \arg\min_{u_k \in U_k(\tilde{x}_k)} \Big[g_k(\tilde{x}_k, u_k) + \tilde{J}_{k+1}\big(f_k(\tilde{x}_k, u_k)\big) \Big], \tag{1.11}$$

and

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

In approximation in value space the calculation of the suboptimal sequence $\{\tilde{u}_0, \ldots, \tilde{u}_{N-1}\}$ is done by going forward (no backward calculation is needed once the approximate cost-to-go functions $\tilde{J}_k$ are available). This is similar to the calculation of the optimal sequence $\{u_0^*, \ldots, u_{N-1}^*\}$, and is independent of how the functions $\tilde{J}_k$ are computed. The motivation for approximation in value space for stochastic DP problems is vastly reduced computation relative to the exact DP algorithm (once $\tilde{J}_k$ have been obtained): the minimization (1.11) needs to be performed only for the N states $x_0, \tilde{x}_1, \ldots, \tilde{x}_{N-1}$ that are encountered during the on-line control of the system, and not for every state within the potentially enormous state space, as is the case for exact DP.

The algorithm (1.11) is said to involve a *one-step lookahead minimization*, since it solves a one-stage DP problem for each k. In what follows we will also discuss the possibility of *multistep lookahead*, which involves the solution of an ℓ-step DP problem, where ℓ is an integer, $1 < \ell < N - k$, with a terminal cost function approximation $\tilde{J}_{k+\ell}$. Multistep lookahead typically (but not always) provides better performance over one-step lookahead in RL approximation schemes. For example in AlphaZero chess, long multistep lookahead is critical for good on-line performance. The intuitive reason is that with ℓ stages being treated "exactly" (by optimization), the effect of the approximation error

$$\tilde{J}_{k+\ell} - J_{k+\ell}^*$$

tends to become less significant as ℓ increases. However, the solution of the multistep lookahead optimization problem, instead of the one-step lookahead counterpart of Eq. (1.11), becomes more time consuming.

Rollout with a Base Heuristic for Deterministic Problems

A major issue in value space approximation is the construction of suitable approximate cost-to-go functions $\tilde{J}_k$. This can be done in many different ways, giving rise to some of the principal RL methods. For example, $\tilde{J}_k$ may be constructed with a sophisticated off-line training method, as discussed in Section 1.1. Alternatively, $\tilde{J}_k$ may be obtained on-line with *rollout*, which will be discussed in detail in this book. In rollout, the approximate values $\tilde{J}_k(x_k)$ are obtained when needed by running a heuristic control scheme, called *base heuristic* or *base policy*, for a suitably large number of stages, starting from the state x_k, and accumulating the costs incurred at these stages.

The major theoretical property of rollout is *cost improvement*: the cost obtained by rollout using some base heuristic is less or equal to the corresponding cost of the base heuristic. This is true for any starting state, provided the base heuristic satisfies some simple conditions, which will be discussed in Chapter 2.†

There are also several variants of rollout, including versions involving multiple heuristics, combinations with other forms of approximation in value space methods, multistep lookahead, and stochastic uncertainty. We will discuss such variants later. For the moment we will focus on a deterministic DP problem with a finite number of controls. Given a state x_k at time k, this algorithm 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

$$x_{t+1} = f_t(x_t, u_t), \qquad t = k, \ldots, N-1,$$

and the corresponding cost

$$H_{k+1}(x_{k+1}) = g_{k+1}(x_{k+1}, u_{k+1}) + \cdots + g_{N-1}(x_{N-1}, u_{N-1}) + g_N(x_N).$$

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:

$$g_k(x_k, u_k) + H_{k+1}(x_{k+1}).$$

† For an intuitive justification of the cost improvement mechanism, note that the rollout control $\tilde{u}_k$ is calculated from Eq. (1.11) to attain the minimum over u_k over the sum of two terms: the first stage cost $g_k(\tilde{x}_k, u_k)$ plus the cost of the remaining stages ($k+1$ to N) using the heuristic controls. Thus rollout involves a first stage optimization (rather than just using the base heuristic), which accounts for the cost improvement. This reasoning also explains why multistep lookahead tends to provide better performance than one-step lookahead in rollout schemes.

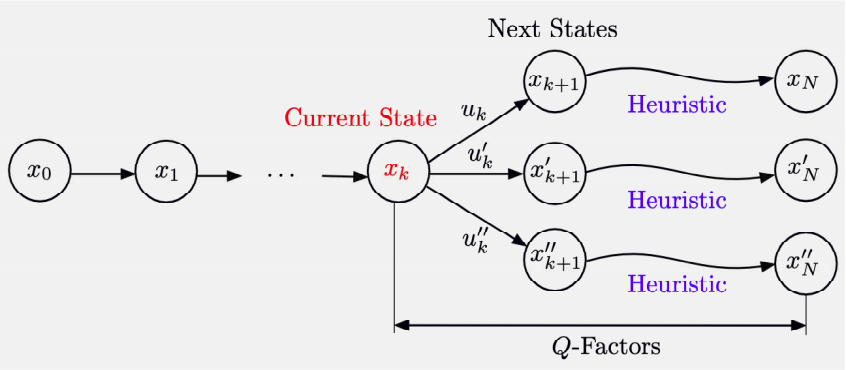

Figure 1.2.9 Schematic illustration of rollout with one-step lookahead for a deterministic problem. At state x_k, for every pair (x_k, u_k), $u_k \in U_k(x_k)$, the base heuristic generates an approximate Q-factor

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

and selects the control $\tilde{\mu}_k(x_k)$ with minimal Q-factor.

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

$$\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k), \tag{1.12}$$

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

$$\tilde{Q}_k(x_k, u_k) = g_k(x_k, u_k) + H_{k+1}\big(f_k(x_k, u_k)\big); \tag{1.13}$$

see Fig. 1.2.9.

Note that the rollout algorithm requires running the base heuristic for a number of times that is bounded by Nn, where n is an upper bound on the number of control choices available at each state. Thus if n is small relative to N, it requires computation equal to a small multiple of N times the computation time for a single application of the base heuristic. Similarly, if n is bounded by a polynomial in N, the ratio of the rollout algorithm computation time to the base heuristic computation time is a polynomial in N.

Example 1.2.3 (Traveling Salesman Problem)

Let us consider the traveling salesman problem of Example 1.2.2, whereby a salesman wants to find a minimum cost tour that visits each of N given cities $c = 0, \ldots, N-1$ exactly once and returns to the city he started from. With each pair of distinct cities c, c', we associate a traversal cost $g(c, c')$. Note

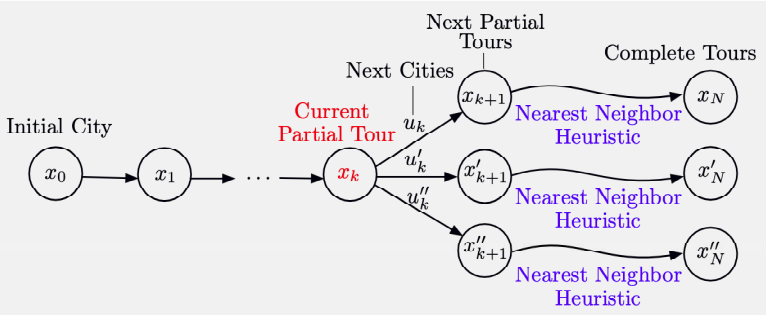

Figure 1.2.10 Rollout with the nearest neighbor heuristic for the traveling salesman problem of Example 1.2.3. The initial state x_0 consists of a single city. The final state x_N is a complete tour of N cities, containing each city exactly once.

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.

There are many heuristic approaches for solving the traveling salesman problem. For illustration purposes, let us focus on the simple *nearest neighbor* heuristic, which starts with a partial tour, i.e., an ordered collection of distinct cities, and constructs a sequence of partial tours, adding to the each partial tour a new city that does not close a cycle and minimizes the cost of the enlargement. In particular, given a sequence $\{c_0, c_1, \ldots, c_k\}$ (with $k < N-1$) consisting of distinct cities, the nearest neighbor heuristic adds a city c_{k+1} that minimizes $g(c_k, c_{k+1})$ over all cities $c_{k+1} \neq c_0, \ldots, c_k$, thereby forming the sequence $\{c_0, c_1, \ldots, c_k, c_{k+1}\}$. Continuing in this manner, the heuristic eventually forms a sequence of N cities, $\{c_0, c_1, \ldots, c_{N-1}\}$, thus yielding a complete tour with cost

$$g(c_0, c_1) + \cdots + g(c_{N-2}, c_{N-1}) + g(c_{N-1}, c_0). \tag{1.14}$$

We can formulate the traveling salesman problem as a DP problem as we discussed in Example 1.2.2. 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 $(c_0, c_1, \ldots, c_k, c_{k+1})$ that correspond to adding one more unvisited city $c_{k+1} \neq c_0, c_1, \ldots, c_k$ (thus the unvisited cities are the feasible controls at a given partial tour/state). 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 given by Eq. (1.14). Note that the number of states at stage k increases exponentially with k, and so does the computation required to solve the problem by exact DP.

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., a sequence $\{c_0, \ldots, c_k\}$ consisting of distinct cities. At the next iteration, we add one more city by running the

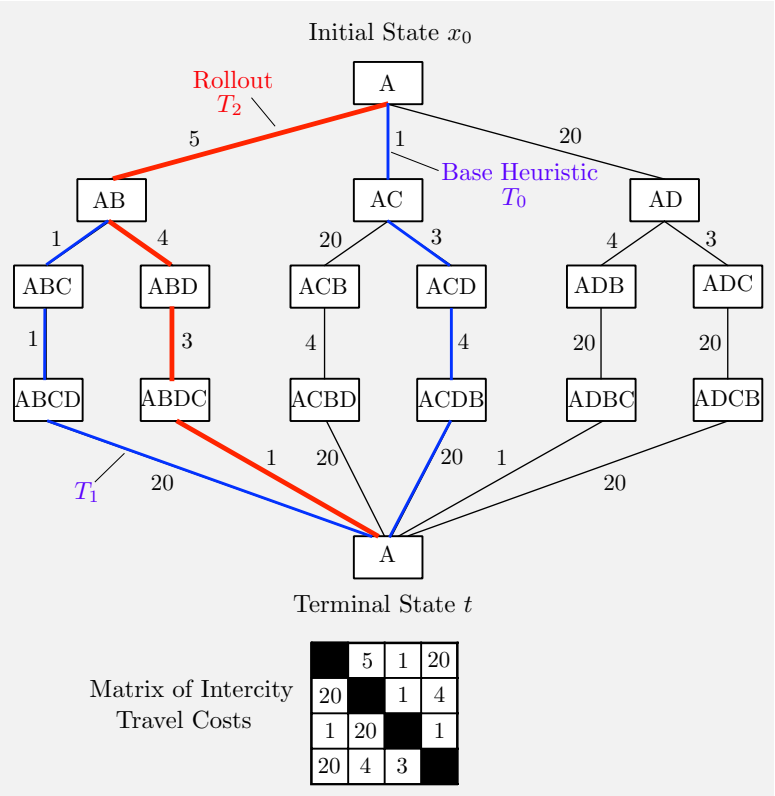

Figure 1.2.11 Rollout with the nearest neighbor base heuristic, applied to a traveling salesman problem. At city A, the nearest neighbor heuristic generates the tour ACDBA (labelled T_0). At city A, the rollout algorithm compares the tours ABCDA, ACDBA, and ADCBA, finds ABCDA (labelled T_1) to have the least cost, and moves to city B. At AB, the rollout algorithm compares the tours ABCDA and ABDCA, finds ABDCA (labelled T_2) to have the least cost, and moves to city D. The rollout algorithm then moves to cities C and A (it has no other choice). The final tour T_2 generated by rollout turns out to be optimal in this example, while the tour T_0 generated by the base heuristic is suboptimal. This is suggestive of a general result: the rollout algorithm for deterministic problems generates a sequence of solutions of decreasing cost under some conditions on the base heuristic that we will discuss in Chapter 2, and which are satisfied by the nearest neighbor heuristic.

nearest neighbor heuristic starting from each of the sequences of the form $\{c_0, \ldots, c_k, c\}$ where $c \neq c_0, \ldots, c_k$. 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. 1.2.10. The overall computation for the rollout solution is bounded by a polynomial in N, and is much smaller than the exact DP computation. Figure 1.2.11 provides an example where the nearest neighbor heuristic and the corresponding rollout algorithm are compared; see also Exercise 1.1.

1.3 STOCHASTIC EXACT AND APPROXIMATE DYNAMIC PROGRAMMING

We will now extend the DP algorithm and our discussion of approximation in value space to problems that involve stochastic uncertainty in their system equation and cost function. We will first discuss the finite horizon case, and the extension of the ideas underlying the principle of optimality and approximation in value space schemes. We will then consider the infinite horizon version of the problem, and provide an overview of the underlying theory and algorithmic methodology.

1.3.1 Finite Horizon Problems

The stochastic optimal control problem differs from its deterministic counterpart 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 with 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 earlier x_k is an element of some state space, the control u_k is an element of some control space.† The cost per stage is denoted by $g_k(x_k, u_k, w_k)$ and also depends on the random disturbance w_k; see Fig. 1.3.1. The control u_k is constrained to take values in a given subset $U_k(x_k)$, which depends on the current state x_k.

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

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

† The discrete equation format and corresponding x-u-w notation is standard in the optimal control literature. For finite-state stochastic problems, also called *Markovian Decision Problems* (MDP), the system is often represented conveniently in terms of control-dependent transition probabilities. A common notation in the RL literature is $p(s, a, s')$ for transition probability from s to s' under action a. This type of notation is not well suited for deterministic problems, which involve no probabilistic structure at all and are of major interest in this book. The transition probability notation is also cumbersome for problems with a continuous state space; see Sections 1.7.1 and 1.7.2 for further discussion. The reader should note, however, that mathematically the system equation and transition probabilities are equivalent, and any analysis that can be done in one notational system can be translated to the other notational system.

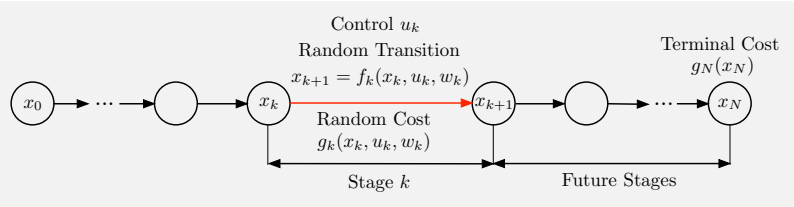

Figure 1.3.1 Illustration of an N-stage stochastic optimal control problem. Starting from state x_k, the next state under control u_k is generated randomly, according to $x_{k+1} = f_k(x_k, u_k, w_k)$, where w_k is the random disturbance, and a random stage cost $g_k(x_k, u_k, w_k)$ is incurred.

and the given distributions $P_k(\cdot \mid x_k, u_k)$. Thus, for given functions g_k, $k = 0, 1, \ldots, N$, the expected cost of π starting at x_0 is

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

where the expected value operation $E\{\cdot\}$ is taken with respect to the joint distribution of 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 policies.

An important difference from the deterministic case is that we optimize not over control sequences $\{u_0, \ldots, u_{N-1}\}$ [cf. Eq. (1.3)], but rather over *policies* (also called *closed-loop control laws*, or *feedback policies*) that consist of a sequence of functions

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

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. 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.

† We assume an introductory probability background on the part of the reader. For an account that is consistent with our use of probability in this book, see the textbook by Bertsekas and Tsitsiklis [BeT08].

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

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

We 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*.

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 from state x_k at stage 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

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

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

$$J_k^*(x_k) = \min_{u_k \in U_k(x_k)} E_{w_k}\Big\{ g_k(x_k, u_k, w_k) + J_{k+1}^*\big(f_k(x_k, u_k, w_k)\big)\Big\}. \tag{1.15}$$

For each x_k and k, define $\mu_k^*(x_k) = u_k^*$ where u_k^* attains the minimum in the right side of this equation. Then, the policy $\pi^* = \{\mu_0^*, \ldots, \mu_{N-1}^*\}$ is optimal.

The key fact is that starting from any initial state x_0, the optimal cost is equal to the number $J_0^*(x_0)$, 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 (which incidentally involves some mathematical fine points; see the discussion of Section 1.3 in the textbook [Ber17a]).

Simultaneously with the off-line computation of the optimal cost-to-go functions $J_0^*, \ldots, J_N^*$, we can compute and store an optimal policy $\pi^* = \{\mu_0^*, \ldots, \mu_{N-1}^*\}$ by minimization in Eq. (1.15). We can then use this

policy on-line to retrieve from memory and apply the control $\mu_k^*(x_k)$ once we reach state x_k. The alternative is to forego the storage of the policy π^* and to calculate the control $\mu_k^*(x_k)$ by executing the minimization (1.15) on-line.

There are a few favorable cases where the optimal cost-to-go functions J_k^* and the optimal policies μ_k^* can be computed analytically using the stochastic DP algorithm. A prominent such case involves a linear system and a quadratic cost function, which is a fundamental problem in control theory. We illustrate the scalar version of this problem next. The analysis can be generalized to multidimensional systems (see optimal control textbooks such as [Ber17a]).

Example 1.3.1 (Linear Quadratic Optimal Control)

Here the system is linear,

$$x_{k+1} = ax_k + bu_k + w_k, \qquad k = 0, \ldots, N-1,$$

and the state, control, and disturbance are scalars. The cost is quadratic of the form:

$$qx_N^2 + \sum_{k=0}^{N-1} (qx_k^2 + ru_k^2),$$

where q and r are known positive weighting parameters. We assume no constraints on x_k and u_k (in reality such problems include constraints, but it is common to neglect the constraints initially, and check whether they are seriously violated later).

As an illustration, consider a vehicle that moves on a straight-line road under the influence of a force u_k and without friction. Our objective is to maintain the vehicle's velocity at a constant level $\bar{v}$ (as in an oversimplified cruise control system). The velocity v_k at time k, after time discretization of its Newtonian dynamics and addition of stochastic noise, evolves according to

$$v_{k+1} = v_k + bu_k + w_k, \tag{1.16}$$

where w_k is a stochastic disturbance with zero mean and given variance σ^2. By introducing $x_k = v_k - \bar{v}$, the deviation between the vehicle's velocity v_k at time k from the desired level $\bar{v}$, we obtain the system equation

$$x_{k+1} = x_k + bu_k + w_k.$$

Here the coefficient b relates to a number of problem characteristics including the weight of the vehicle, the road conditions. The cost function expresses our desire to keep x_k near zero with relatively little force.

We will apply the DP algorithm, and derive the optimal cost-to-go functions J_k^* and optimal policy. We have

$$J_N^*(x_N) = qx_N^2,$$

and by applying Eq. (1.15), we obtain

$$\begin{aligned} J^*_{N-1}(x_{N-1}) &= \min_{u_{N-1}} E\Big\{qx^2_{N-1} + ru^2_{N-1} + J^*_N(ax_{N-1} + bu_{N-1} + w_{N-1})\Big\} \\ &= \min_{u_{N-1}} E\Big\{qx^2_{N-1} + ru^2_{N-1} + q(ax_{N-1} + bu_{N-1} + w_{N-1})^2\Big\} \\ &= \min_{u_{N-1}} \Big[qx^2_{N-1} + ru^2_{N-1} + q(ax_{N-1} + bu_{N-1})^2 \\ &\qquad + 2qE\{w_{N-1}\}(ax_{N-1} + bu_{N-1}) + qE\{w^2_{N-1}\}\Big], \end{aligned}$$

and finally, using the assumptions $E\{w_{N-1}\} = 0$, $E\{w^2_{N-1}\} = \sigma^2$, and bringing out of the minimization the terms that do not depend on u_{N-1},

$$J^*_{N-1}(x_{N-1}) = qx^2_{N-1} + q\sigma^2 + \min_{u_{N-1}} \Big[ru^2_{N-1} + q(ax_{N-1} + bu_{N-1})^2\Big]. \quad (1.17)$$

The expression minimized over u_{N-1} in the preceding equation is convex quadratic in u_{N-1}, so by setting to zero its derivative with respect to u_{N-1},

$$0 = 2ru_{N-1} + 2qb(ax_{N-1} + bu_{N-1}),$$

we obtain the optimal policy for the last stage:

$$\mu^*_{N-1}(x_{N-1}) = -\frac{abq}{r + b^2q}\, x_{N-1}.$$

Substituting this expression into Eq. (1.17), we obtain with a straightforward calculation

$$J^*_{N-1}(x_{N-1}) = K_{N-1}x^2_{N-1} + q\sigma^2,$$

where

$$K_{N-1} = \frac{a^2rq}{r + b^2q} + q.$$

We can now continue the DP algorithm to obtain J^*_{N-2} from J^*_{N-1}. An important observation is that J^*_{N-1} is quadratic (plus an inconsequential constant term), so with a similar calculation we can derive μ^*_{N-2} and J^*_{N-2} in closed form, as a linear and a quadratic (plus constant) function of x_{N-2}, respectively. This process can be continued going backwards, and it can be verified by induction that for all k, we obtain the optimal policy and optimal cost-to-go function in the form

$$\mu^*_k(x_k) = L_kx_k, \qquad k = 0, 1, \ldots, N-1,$$

$$J^*_k(x_k) = K_kx^2_k + \sigma^2 \sum_{t=k}^{N-1} K_{t+1}, \qquad k = 0, 1, \ldots, N-1,$$

where

$$L_k = -\frac{abK_{k+1}}{r + b^2K_{k+1}}, \qquad k = 0, 1, \ldots, N-1, \quad (1.18)$$

and the sequence $\{K_k\}$ is generated backwards by the equation

$$K_k = \frac{a^2 r K_{k+1}}{r + b^2 K_{k+1}} + q, \qquad k = 0, 1, \ldots, N-1, \tag{1.19}$$

starting from the terminal condition $K_N = q$.

The process by which we obtained an analytical solution in this example is noteworthy. 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, even for multi-dimensional linear systems (see [Ber17a], Section 3.1). The optimal policy is a linear function of the state, and the optimal cost function is a quadratic in the state plus a constant.

Another remarkable feature of this example, which can also be extended to multi-dimensional systems, is that the optimal policy does not depend on the variance of w_k, and remains unaffected when w_k is replaced by its mean (which is zero in our example). This is known as *certainty equivalence*, and occurs in several types of problems involving a linear system and a quadratic cost; see [Ber17a], Sections 3.1 and 4.2. For example it holds even when w_k has nonzero mean. 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. This is an important part of the RL methodology, which we will discuss later in this chapter, and in more detail in Chapter 2.

Note that the linear quadratic type of problem illustrated in the preceding example is exceptional in that it admits an elegant analytical solution. Most DP problems encountered in practice require a computational solution.

Q-Factors and Q-Learning for Stochastic Problems

Similar to the case of deterministic problems [cf. Eq. (1.9)], we can define optimal Q-factors for a stochastic problem, as the expressions that are minimized in the right-hand side of the stochastic DP equation (1.15). They are given by

$$Q_k^*(x_k, u_k) = E_{w_k}\Big\{ g_k(x_k, u_k, w_k) + J_{k+1}^*\big(f_k(x_k, u_k, w_k)\big)\Big\}. \tag{1.20}$$

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

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

We will later be interested in approximate Q-factors, where J_{k+1}^* in Eq. (1.20) is replaced by an approximation $\tilde{J}_{k+1}$. Again, the Q-factor corresponding to a state-control pair (x_k,u_k) is the sum of the expected first stage cost using (x_k,u_k), plus the expected cost of the remaining stages starting from the next state as estimated by the function $\tilde{J}_{k+1}$.

1.3.2 Approximation in Value Space for Stochastic DP

Generally the computation of the optimal cost-to-go functions J_k^* can be very time-consuming or impossible. One of the principal RL methods to deal with this difficulty is approximation in value space. Here approximations $\tilde{J}_k$ are used in place of J_k^*, similar to the deterministic case; cf. Eqs. (1.8) and (1.11).

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

At any state x_k encountered at stage k, set

$$\tilde{\mu}_k(x_k) \in \arg\min_{u_k\in U_k(x_k)} E_{w_k}\Big\{g_k(x_k,u_k,w_k) + \tilde{J}_{k+1}\big(f_k(x_k,u_k,w_k)\big)\Big\}. \tag{1.21}$$

The one-step lookahead minimization (1.21) needs to be performed only for the N states $x_0,\ldots,x_{N-1}$ that are encountered during the on-line control of the system. By contrast, exact DP requires that this type of minimization be done for every state and stage.

Methods for Approximation in Value Space

When designing approximation in value space schemes, one may consider several interesting simplification ideas, which are aimed at alleviating the computational overhead. Aside from cost function approximation (use $\tilde{J}_{k+1}$ in place of J_{k+1}^*), there are other possibilities. One of them is to simplify the lookahead minimization over $u_k \in U_k(x_k)$ [cf. Eq. (1.15)] by replacing $U_k(x_k)$ with a suitably chosen subset of controls that are viewed as most promising based on some heuristic criterion.

In Section 1.6.5, we will discuss a related idea for control space simplification for the multiagent case where the control consists of multiple

components, $u_k = (u_k^1, \ldots, u_k^m)$. Then, a sequence of m single component minimizations can be used instead, with potentially enormous computational savings resulting.

Another type of simplification relates to approximations in the computation of the expected value in Eq. (1.21) by using limited Monte Carlo simulation. The Monte Carlo Tree Search method, which will be discussed in Chapter 2, Section 2.7.4, is one possibility of this type.

Still another type of expected value simplification is based on the *certainty equivalence approach*, which will be discussed in more detail in Chapter 2, Section 2.7.2. In this approach, at stage k, we replace the future random variables $w_{k+1}, \ldots, w_{k+m}$ by some deterministic values $\overline{w}_{k+1}, \ldots, \overline{w}_{k+m}$, such as their expected values. We may also view this as a form of problem approximation, whereby for the purpose of computing $\tilde{J}_{k+1}(x_{k+1})$, we "pretend" that the problem is deterministic, with the future random quantities replaced by deterministic typical values. This is one of the most effective techniques to make approximation in value space for stochastic problems computationally tractable, particularly when it is also combined with multistep lookahead minimization, as we will discuss later.

Figure 1.3.2 illustrates the three approximations involved in approximation in value space for stochastic problems: *cost-to-go approximation, simplified minimization,* and *expected value approximation*. They may be designed largely independently of each other, and may be implemented with a variety of methods. Much of the discussion in this book will revolve around different ways to organize these three approximations for both cases of one-step and multistep lookahead.

As indicated in Fig. 1.3.2, an important approach for cost-to-go approximation is *problem approximation*, whereby the functions $\tilde{J}_{k+1}$ in Eq. (1.21) 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. Several types of problem approximation approaches are discussed in the author's RL book [Ber19a]. A major approach is *aggregation*, which will be discussed in Section 3.5. In this book, problem approximation will not receive much attention, despite the fact that it can often be combined very effectively with the approximation in value space methodology that is our main focus.

Another important approach for on-line cost-to-go approximation is rollout, which we discuss next. This is similar to the rollout approach for deterministic problems, discussed in Section 1.2.

Rollout for Stochastic Problems - Truncated Rollout

In the rollout approach, we select $\tilde{J}_{k+1}$ in Eq. (1.21) to be the cost function of a suitable base policy (perhaps with some approximation). Note that

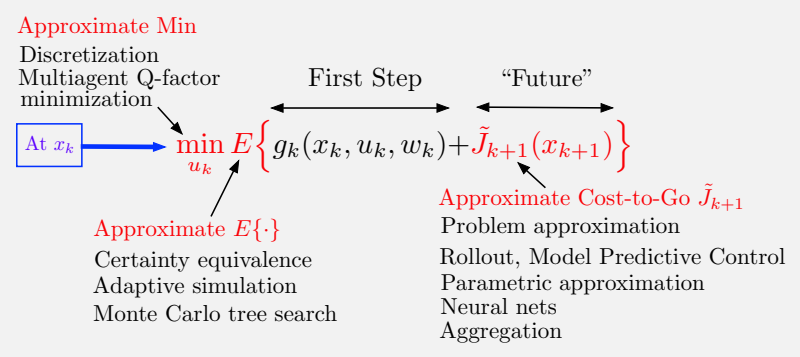

Figure 1.3.2 Schematic illustration of approximation in value space for stochastic problems, and the three approximations involved in its design. Typically the approximations can be designed independently of each other, and with a variety of approaches. There are also multistep lookahead versions of approximation in value space, which will be discussed later.

any policy can be used on-line as base policy, including policies obtained by a sophisticated off-line procedure, using for example neural networks and training data. The rollout algorithm has a cost improvement property, whereby it yields an improved cost relative to its underlying base policy. We will discuss this property and some conditions under which it is guaranteed to hold in Chapter 2.

A major variant of rollout is *truncated rollout*, which combines the use of one-step optimization, simulation of the base policy for a certain number of steps m, and then adds an approximate cost $\tilde{J}_{k+m+1}(x_{k+m+1})$ to the cost of the simulation, which depends on the state x_{k+m+1} obtained at the end of the rollout. Note that if one foregoes the use of a base policy (i.e., $m = 0$), one recovers as a special case the general approximation in value space scheme (1.21); see Fig. 1.3.3. Thus rollout provides an extra layer of lookahead to the one-step minimization, but this lookahead need not extend to the end of the horizon.

Note also that versions of truncated rollout with multistep lookahead minimization are possible. They will be discussed later. The terminal cost approximation is necessary in infinite horizon problems, since an infinite number of stages of the base policy rollout is impossible. However, even for finite horizon problems it may be necessary and/or beneficial to artificially truncate the rollout horizon. Generally, a large combined number of multistep lookahead minimization and rollout steps is likely to be beneficial.

Cost Versus Q-Factor Approximations - Robustness and On-Line Replanning

Similar to the deterministic case, Q-learning involves the calculation of either the optimal Q-factors (1.20) or approximations $\tilde{Q}_k(x_k, u_k)$. The

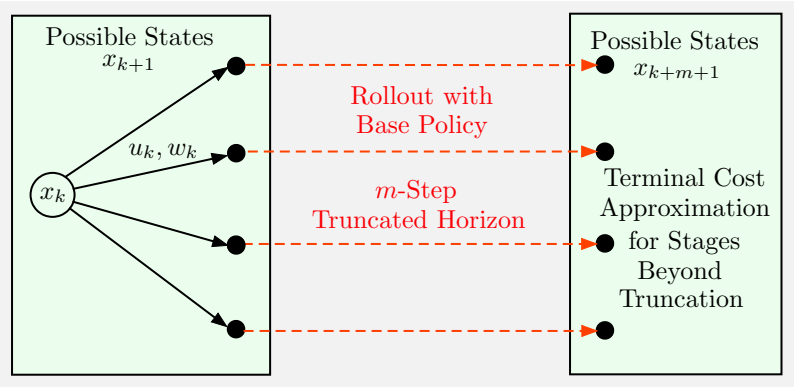

Figure 1.3.3 Schematic illustration of truncated rollout. One-step lookahead is followed by simulation of the base policy for m steps, and an approximate cost $\tilde{J}_{k+m+1}(x_{k+m+1})$ is added to the cost of the simulation, which depends on the state x_{k+m+1} obtained at the end of the rollout. If the base policy simulation is omitted (i.e., $m = 0$), one recovers the general approximation in value space scheme (1.21). Truncated rollout with multistep lookahead is also possible and is discussed in some detail in Chapter 2.

approximate Q-factors may be obtained using approximation in value space schemes, and can be used to obtain approximately optimal policies through the Q-factor minimization

$$\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k). \tag{1.22}$$

Since it is possible to implement approximation in value space by using cost function approximations [cf. Eq. (1.21)] or by using Q-factor approximations [cf. Eq. (1.22)], the question arises which one to use in a given practical situation. One important consideration is the facility of obtaining suitable cost or Q-factor approximations. This depends largely on the problem and also on the availability of data on which the approximations can be based. However, there are some other major considerations.

In particular, the cost function approximation scheme

$$\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} E_{w_k} \Big\{ g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}\big(f_k(x_k, u_k, w_k)\big) \Big\}, \tag{1.23}$$

has an important disadvantage: *the expected value above needs to be computed on-line for all $u_k \in U_k(x_k)$, and this may involve substantial computation*. It also has an important advantage in situations where the system function f_k, the cost per stage g_k, or the control constraint set $U_k(x_k)$ can change as the system is operating. Assuming that the new f_k, g_k, or $U_k(x_k)$ become known to the controller at time k, *on-line replanning may be used, and this may improve substantially the robustness of the approximation in*

value space scheme. By comparison, the Q-factor function approximation scheme (1.22) does not allow for on-line replanning. On the other hand, for problems where there is no need for on-line replanning, the Q-factor approximation scheme may not require the on-line computation of expected values and may allow a much faster on-line computation of the minimizing control $\tilde{\mu}_k(x_k)$ via Eq. (1.22).

One more disadvantage of using Q-factors will emerge later, as we discuss the synergy between off-line training and on-line play based on Newton's method; see Section 1.5. In particular, we will interpret the cost function of the lookahead minimization policy $\{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$ as the result of one step of Newton's method for solving the Bellman equation that underlies the DP problem, starting from the terminal cost function approximations $\{\tilde{J}_1, \ldots, \tilde{J}_N\}$. This synergy tends to be negatively affected when Q-factor (rather than cost) approximations are used.

1.3.3 Approximation in Policy Space

The major alternative to approximation in value space is *approximation in policy space*, whereby we select the policy from a suitably restricted class of policies, usually a parametric class of some form. In particular, we can introduce a parametric family of policies (or approximation architecture, as we will call it in Chapter 3),

$$\tilde{\mu}_k(x_k, r_k), \qquad k = 0, \ldots, N-1,$$

where r_k is a parameter, and then estimate the parameters r_k using some type of training process or optimization; cf. Fig. 1.3.4.

Neural networks, described in Chapter 3, are often used to generate the parametric class of policies, in which case r_k is the vector of weights/parameters of the neural network. In Chapter 3, we will also discuss methods for obtaining the training data required for obtaining the parameters r_k, and we will consider several other classes of approximation architectures.

A general scheme for parametric approximation in policy space is to somehow obtain a training set, consisting of 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

$$\min_{r_k} \sum_{s=1}^{q} \left\| u_k^s - \tilde{\mu}_k(x_k^s, r_k) \right\|^2 \tag{1.24}$$

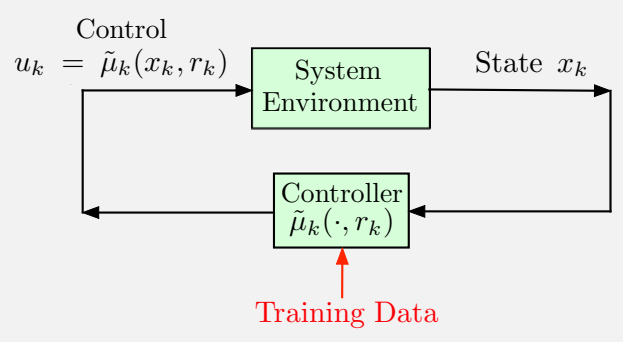

Figure 1.3.4 Schematic illustration of parametric approximation in policy space. A policy

$$\tilde{\mu}_k(x_k, r_k), \qquad k = 0, 1, \ldots, N-1,$$

from a parametric class is computed off-line based on data, and it is used to generate the control $u_k = \tilde{\mu}_k(x_k, r_k)$ on-line, when at state x_k.

(possibly modified to add 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.

An important approach for generating the training set (x_k^s, u_k^s), $s = 1, \ldots, q$, for the least squares training problem (1.24) is based on approximation in value space. In particular, we may use a one-step lookahead minimization of the form

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

† Here $\|\cdot\|$ denotes the standard quadratic Euclidean norm. It is implicitly assumed here (and in similar situations later) that the controls are members of a Euclidean space (i.e., the space of finite dimensional vectors with real-valued components) so that the distance between two controls can be measured by their normed difference (randomized controls, i.e., probabilities that a particular action will be used, fall in this category). Regression problems of this type arise in the training of *parametric classifiers* based on data, including the use of neural networks (see Section 3.4). Assuming a finite control space, the classifier is trained using the data $\big(x_k^s, u_k^s\big)$, $s = 1, \ldots, q$, which are viewed as state-category pairs, and then a state x_k is classified as being of "category" $\tilde{\mu}_k(x_k, r_k)$. Parametric approximation architectures, and their training through the use of classification and regression techniques are described in Chapter 3. An important modification is to use *regularized regression* where a quadratic regularization term is added to the least squares objective. This term is a positive multiple of the squared deviation $\|r - \hat{r}\|^2$ of r from some initial guess $\hat{r}$.

where $\tilde{J}_{k+1}$ is a suitable (separately obtained) approximation in value space. Alternatively, we may use an approximate Q-factor based minimization

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

where $\tilde{Q}_k$ is a (separately obtained) Q-factor approximation. We may view this as *approximation in policy space built on top of approximation in value space*.

There is a significant advantage of the least squares training procedure of Eq. (1.24), and more generally approximation in policy space: once the parametrized policy $\tilde{\mu}_k$ is obtained, the computation of controls

$$u_k = \tilde{\mu}_k(x_k, r_k), \qquad k = 0, \ldots, N-1,$$

during on-line operation of the system is often much easier compared with the lookahead minimization (1.23). For this reason, one of the major uses of approximation in policy space is to provide an *approximate implementation of a known policy* (no matter how obtained) for the purpose of convenient on-line use. On the negative side, such an implementation is less well suited for on-line replanning.

Model-Free Approximation in Policy Space

There are also alternative optimization-based approaches for policy space approximation. The main idea is that once we use a vector $(r_0, r_1, \ldots, r_{N-1})$ to parametrize the policies π, the expected cost $J_\pi(x_0)$ is parametrized as well, and can be viewed as a function of $(r_0, r_1, \ldots, r_{N-1})$. We can then optimize this cost by using a gradient-like or random search method. This is a widely used approach for optimization in policy space, which, however, will receive limited attention in this book (see Section 3.5, and the RL book [Ber19a], Section 5.7).

An interesting feature of this approach is that in principle it does not require a mathematical model of the system and the cost function; a computer simulator (or availability of the real system for experimentation) suffices instead. This is sometimes called a *model-free implementation*. The advisability of implementations of this type, particularly when they rely exclusively on simulation (i.e., without the use of prior mathematical model knowledge), is a hotly debated and much contested issue; see for example the review paper by Alamir [Ala22].

We finally note an important conceptual difference between approximation in value space and approximation in policy space. The former is primarily an on-line method (with off-line training used optionally to construct cost function approximations for one-step or multistep lookahead). The latter is primarily an off-line training method (which may be used without modification for on-line play or optionally to provide a policy for on-line rollout).

1.3.4 Off-Line Training of Cost Function and Policy Approximations

When it comes to off-line constructed approximations, a major approach is based on the use of parametric approximation. Feature-based architectures and neural networks are very useful within our RL context, and will be discussed in Chapter 3, together with methods that can be used for training them.†

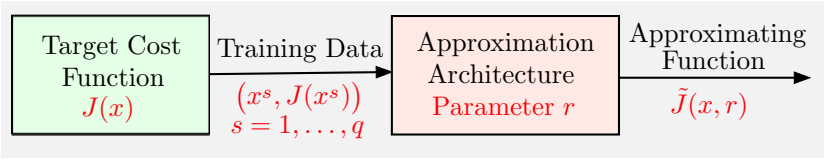

Figure 1.3.5 The general structure for parametric cost approximation. We approximate the target cost function $J(x)$ with a member from a parametric class $\tilde{J}(x, r)$ that depend on a parameter vector r. We use training data $\big(x^s, J(x^s)\big)$, $s = 1, \ldots, q$, and a form of optimization that aims to find a parameter $\hat{r}$ that "minimizes" the size of the errors $J(x^s) - \tilde{J}(x^s, \hat{r})$, $s = 1, \ldots, q$.

A general structure for parametric cost function approximation is illustrated in Fig. 1.3.5. We have a target function $J(x)$ that we want to approximate with a member of a parametric class of functions $\tilde{J}(x, r)$ that depend on a parameter vector r (to simplify, we drop the time index, using J in place of J_k). To this end, we collect training data $\big(x^s, J(x^s)\big)$, $s = 1, \ldots, q$, which we use to determine a parameter $\hat{r}$ that leads to a good "fit" between the data $J(x^s)$ and the predictions $\tilde{J}(x^s, \hat{r})$ of the parametrized function. This is usually done through some form of optimization that

† The principal role of neural networks within the context of this book is to provide the means for approximating various target functions from input-output data. This includes cost functions and Q-factors of given policies, and optimal cost-to-go functions and Q-factors; in this case the neural network is referred to as a *value network* (sometimes the alternative term *critic network* is also used). In other cases the neural network represents a policy viewed as a function from state to control, in which case it is called a *policy network* (the alternative term *actor network* is also used). The training methods for constructing the cost function, Q-factor, and policy approximations themselves from data are mostly based on optimization and regression, and will be reviewed in Chapter 3. Further DP-oriented discussions are found in many sources, including the RL books [Ber19a], [Ber20a], and the neuro-dynamic programming book [BeT96]. Machine learning books, including those describing at length neural network architectures and training are also recommended; see e.g., the recent book by Bishop and Bishop [BiB24], and the references quoted therein.

aims to minimize in some sense the size of the errors $J(x^s) - \tilde{J}(x^s, \hat{r})$, $s = 1, \ldots, q$.

The methodological ideas for parametric cost approximation can also be used for approximation of a target policy μ with a policy from a parametric class $\tilde{\mu}(x, r)$. The training data may be obtained, for example, from rollout control calculations, thus enabling the construction of both value and policy networks that can be combined for use in a perpetual rollout scheme. However, there is an important difference: the approximate cost values $\tilde{J}(x, r)$ are real numbers, whereas the approximate policy values $\tilde{\mu}(x, r)$ are elements of a control space U. Thus if U consists of m dimensional vectors, $\tilde{\mu}(x, r)$ consists of m numerical components. In this case the parametric approximation problems for cost functions and for policies are fairly similar, and both involve continuous space approximations.

On the other hand, the case where the control space is finite, $U = \{u^1, \ldots, u^m\}$. is markedly different. In this case, for any x, $\tilde{\mu}(x, r)$ consists of one of the m possible controls $u^1, \ldots, u^m$. This ushers a connection with traditional classification schemes, whereby objects x are classified as belonging to one of the categories $u^1, \ldots, u^m$, so that $\mu(x)$ *defines the category of* x*, and can be viewed as a classifier*. Some of the most prominent classification schemes actually produce randomized outcomes, i.e., x is associated with a probability distribution

$$\{\tilde{\mu}(u^1, r), \ldots, \tilde{\mu}(u^m, r)\}, \tag{1.25}$$

which is a randomized policy in our policy approximation context; see Fig. 1.3.6. This is done usually for reasons of algorithmic convenience, since many optimization methods, including least squares regression, require that the optimization variables are continuous. In this case, the randomized policy (1.25) can be converted to a nonrandomized policy using a maximization operation: associate x with the control of maximum probability (cf. Fig. 1.3.6),

$$\tilde{\mu}(x, r) \in \arg \max_{i=1,\ldots,m} \tilde{\mu}^i(x, r). \tag{1.26}$$

The use of classification methods for approximation in policy space will be discussed in Chapter 3 (Section 3.4).

1.4 INFINITE HORIZON PROBLEMS - AN OVERVIEW

We will now provide an outline of infinite horizon stochastic DP with an emphasis on its aspects that relate to our RL/approximation methods. We will deal primarily with infinite horizon stochastic problems, where we aim to minimize the total cost over an infinite number of stages, given by

$$J_\pi(x_0) = \lim_{N \to \infty} \mathop{E}_{\substack{w_k \\ k=0,1,\ldots}} \left\{ \sum_{k=0}^{N-1} \alpha^k g\big(x_k, \mu_k(x_k), w_k\big) \right\}; \tag{1.27}$$

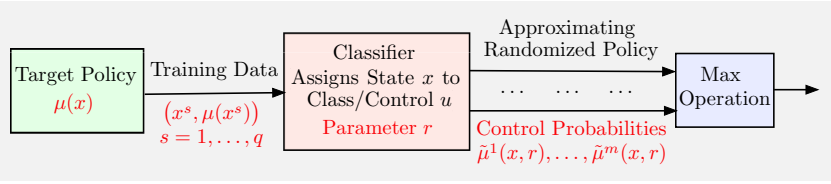

Figure 1.3.6 A general structure for parametric policy approximation for the case where the control space is finite, $U = \{u^1, \ldots, u^m\}$, and its relation to a classification scheme. It produces a randomized policy of the form (1.25), which is converted to a nonrandomized policy through the maximization operation (1.26).

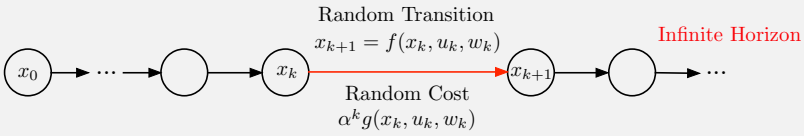

Figure 1.4.1 Illustration of an infinite horizon problem. The system and cost per stage are stationary, except for the use of a discount factor α. If $\alpha = 1$, there is typically a special cost-free termination state that we aim to reach.

see Fig. 1.4.1. 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 scalar in the interval $(0, 1]$. The functions g and f that define the cost per stage and the system equation

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

do not change from one stage to the next. The stochastic disturbances, $w_0, w_1, \ldots$, have a common probability distribution $P(\cdot \mid x_k, u_k)$.

When α is strictly less that 1, it has the meaning of a *discount factor*, and its effect is that future costs matter to us less than the same costs incurred at the present time. Among others, a discount factor guarantees that the limit defining $J_\pi(x_0)$ exists and is finite (assuming that the range of values of the stage cost g is bounded). This is a nice mathematical property that makes discounted problems analytically and algorithmically tractable.

Thus, by definition, the infinite horizon cost of a policy is the limit of its finite horizon costs as the horizon tends to infinity. The three types of problems that we will focus on 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 that state it remains there at no further cost. In some types of problems, the termination state may represent a goal state that we are trying to reach at minimum cost, while in others it may be a state that we are trying to avoid for as long as possible. We will mostly

assume a problem structure such that termination is inevitable under all policies. Thus the horizon is in effect finite, but its length is random and may be affected by the policy being used. A significantly more complicated type of SSP problems, which we will discuss selectively, arises when termination can be guaranteed only for a subset of policies, which includes all optimal policies. Some common types of SSP belong to this category, including deterministic shortest path problems that involve graphs with cycles.

(b) *Discounted problems*. Here, $\alpha < 1$ and there need not be a termination state. However, we will see that a discounted problem with a finite number of states 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 state and stage, thus making termination inevitable. As a result, algorithms and analysis for SSP problems can be easily adapted to discounted problems; the DP textbook [Ber17a] provides a detailed account of this conversion, and an accessible introduction to discounted and SSP problems with a finite number of states.

(c) *Deterministic nonnegative cost problems*. Here, the disturbance w_k takes a single known value. Equivalently, there is no disturbance in the system equation and the cost expression, which now take the form

$$x_{k+1} = f(x_k, u_k), \qquad k = 0, 1, \ldots, \tag{1.28}$$

and

$$J_\pi(x_0) = \lim_{N\to\infty} \sum_{k=0}^{N-1} \alpha^k g\big(x_k, \mu_k(x_k)\big). \tag{1.29}$$

We assume further that there is a cost-free and absorbing termination state t, and that we have

$$g(x, u) \geq 0, \qquad \text{for all } x \neq t,\ u \in U(x), \tag{1.30}$$

and $g(t, u) = 0$ for all $u \in U(t)$. This type of structure expresses the objective to reach or approach t at minimum cost, a classical control problem. An extensive analysis of the undiscounted version of this problem was given in the author's paper [Ber17b].

Discounted stochastic problems with a finite number of states [also referred to as *discounted MDP (abbreviation for Markovian Decision Problem)*] are very common in the DP/RL literature, particularly because of their benign analytical and computational nature. Moreover, there is a widespread belief that discounted MDP can be used as a universal model, i.e., that in practice any other kind of problem (e.g., undiscounted problems with a termination state and/or a continuous state space) can be painlessly

converted to a discounted MDP with a discount factor that is close enough to 1. This is questionable, however, for a number of reasons:

(a) Deterministic models are common as well as natural in many practical contexts (including discrete optimization/integer programming problems), so to convert them to MDP does not make sense.

(b) The conversion of a continuous-state problem to a finite-state problem through some kind of discretization involves mathematical subtleties that can lead to serious practical/algorithmic complications. In particular, the character of the optimal solution may be seriously distorted by converting to a discounted MDP through some form of discretization, regardless of how fine the discretization is.

(c) For some practical shortest path contexts it is essential that the termination state is ultimately reached. However, when a discount factor α is introduced in such a problem, the character of the problem may be fundamentally altered. In particular, the threshold for an appropriate value of α may be very close to 1 and may be unknown in practice. For a simple example consider a shortest path problem with states 1 and 2 plus a termination state t. From state 1 we can go to state 2 at cost 0, from state 2 we can go to either state 1 at a small cost $\epsilon > 0$ or to the termination state at a substantial cost $C > 0$. The optimal policy over an infinite horizon is to go from 1 to 2 and from 2 to t. Suppose now that we approximate the problem by introducing a discount factor $\alpha \in (0, 1)$. Then it can be shown that if $\alpha < 1 - \epsilon/C$, it is optimal to move indefinitely around the cycle $1 \to 2 \to 1 \to 2$ and never reach t, while for $\alpha > 1 - \epsilon/C$ the shortest path $2 \to 1 \to t$ will be obtained. Thus the solution of the discounted problem varies discontinuously with α: it changes radically at some threshold, which in general may be unknown.

An important class of problems that we will consider in some detail in this book is finite-state deterministic problems with a large number of states. Finite horizon versions of these problems include challenging discrete optimization problems, whose exact solution is practically impossible. An important fact to keep in mind is that we can transform such problems to infinite horizon SSP problems with a termination state at the end of the horizon, so that the conceptual framework of the present section applies. The approximate solution of discrete optimization problems by RL methods, and particularly by rollout, will be considered in Chapter 2, and has been discussed at length in the books [Ber19a] and [Ber20a].

1.4.1 Infinite Horizon Methodology

There are several analytical and computational issues regarding our infinite horizon problems. Many of them revolve around the relation between the

optimal cost function J^* of the infinite horizon problem and the optimal cost functions of the corresponding N-stage problems.

In particular, 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 algorithm

$$J_{k+1}(x) = \min_{u\in U(x)} E_w\Big\{g(x,u,w) + \alpha J_k\big(f(x,u,w)\big)\Big\}, \qquad k = 0,1,\ldots, \tag{1.31}$$

starting from $J_0(x) \equiv 0$.† The algorithm (1.31) 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:

(a) The optimal infinite horizon cost is the limit of the corresponding N-stage optimal costs as $N \to \infty$; i.e.,

$$J^*(x) = \lim_{N\to\infty} J_N(x) \tag{1.32}$$

for all states x.

(b) The following equation should hold for all states x,

$$J^*(x) = \min_{u\in U(x)} E_w\Big\{g(x,u,w) + \alpha J^*\big(f(x,u,w)\big)\Big\}. \tag{1.33}$$

This is obtained by taking the limit as $N \to \infty$ in the VI algorithm (1.31) using Eq. (1.32). The preceding equation, called *Bellman's equation*, is really a system of equations (one equation per state x), which has as solution the optimal costs-to-go of all the states.

(c) If $\mu(x)$ attains the minimum in the right-hand side of the Bellman equation (1.33) for each x, then the policy $\{\mu,\mu,\ldots\}$ should be optimal. This type of policy is called *stationary*, and for simplicity it is denoted by μ.

(d) The cost function J_μ of a stationary policy μ satisfies

$$J_\mu(x) = E_w\Big\{g\big(x,\mu(x),w\big) + \alpha J_\mu\big(f(x,\mu(x),w)\big)\Big\}, \qquad \text{for all } x. \tag{1.34}$$

---

† This is just the finite horizon DP algorithm of Section 1.3.1, except that we have reversed the time indexing to suit our infinite horizon context. In particular, consider the N-stages problem and let $V_{N-k}(x)$ be the optimal cost-to-go starting at x with k stages to go, and with terminal cost equal to 0. Applying DP, we have for all x,

$$V_{N-k}(x) = \min_{u\in U(x)} E_w\Big\{\alpha^{N-k} g(x,u,w) + V_{N-k+1}\big(f(x,u,w)\big)\Big\}, \quad V_N(x) = 0.$$

By defining $J_k(x) = V_{N-k}(x)/\alpha^{N-k}$, we obtain the VI algorithm (1.31).

We can view this as just the Bellman equation (1.33) for a different problem, where for each x, the control constraint set $U(x)$ consists of just one control, namely $\mu(x)$. Moreover, we expect that J_μ is obtained in the limit by the VI algorithm:

$$J_\mu(x) = \lim_{N\to\infty} J_{\mu,N}(x), \qquad \text{for all } x,$$

where $J_{\mu,N}$ is the N-stage cost function of μ generated by

$$J_{\mu,k+1}(x) = E_w\Big\{g\big(x,\mu(x),w\big) + \alpha J_{\mu,k}\big(f(x,\mu(x),w)\big)\Big\}, \qquad (1.35)$$

starting from $J_{\mu,0}(x) \equiv 0$ or some other initial condition; cf. Eqs. (1.31)-(1.32).

All four of the preceding results can be shown to hold for finite-state discounted problems, and also for finite-state SSP problems under reasonable assumptions. The results also hold for infinite-state discounted problems, provided the cost per stage function g is bounded over the set of possible values of (x, u, w), in which case we additionally can show that J^* is the unique solution of Bellman's equation. The VI algorithm is also valid under these conditions, in the sense that $J_k \to J^*$, even if the initial function J_0 is nonzero. The motivation for a different choice of J_0 is faster convergence to J^*; generally the convergence is faster as J_0 is chosen closer to J^*. The associated mathematical proofs can be found in several sources, e.g., [Ber12], Chapter 1, or [Ber19a], Chapter 4.†

It is important to note that for infinite horizon problems, there are additional important algorithms that are amenable to approximation in value space. Approximate policy iteration, Q-learning, temporal difference methods, linear programming, and their variants are some of these; see the RL books [Ber19a], [Ber20a]. For this reason, in the infinite horizon case, there is a richer set of algorithmic options for approximation in value space, despite the fact that the associated mathematical theory is more complex. In this book, we will only discuss approximate forms and variations of the policy iteration algorithm, which we describe next.

Policy Iteration

A major infinite horizon algorithm is *policy iteration* (PI for short). We will argue that PI, together with its variations, forms the foundation for

† For undiscounted problems and discounted problems with unbounded cost per stage, we may still adopt the four preceding results as a working hypothesis. However, we should also be aware that exceptional behavior is possible under unfavorable circumstances, including nonuniqueness of solution of Bellman's equation, and nonconvergence of the VI algorithm to J^* from some initial conditions; see the books [Ber12], [Ber22b].

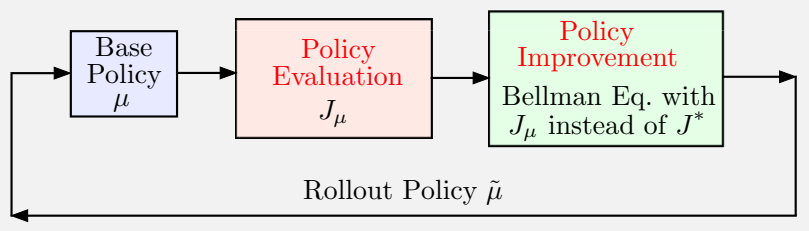

Figure 1.4.2 Schematic illustration of PI as repeated rollout. It generates a sequence of policies, with each policy μ in the sequence being the base policy that generates the next policy $\tilde{\mu}$ in the sequence as the corresponding rollout policy. This rollout policy is used as the base policy in the subsequent iteration.

self-learning in RL, i.e., learning from data that is self-generated (from the system itself as it operates) rather than from data supplied from an external source. Figure 1.4.2 describes the method as repeated rollout, and indicates that each of its iterations consists of two phases:

(a) *Policy evaluation*, which computes the cost function J_μ of the current (or base) policy μ. One possibility is to solve the corresponding Bellman equation

$$J_\mu(x) = E_w\Big\{g\big(x,\mu(x),w\big) + \alpha J_\mu\big(f(x,\mu(x),w)\big)\Big\}, \qquad \text{for all } x,$$

cf. Eq. (1.34). However, the value $J_\mu(x)$ for any x can also be computed by Monte Carlo simulation, by averaging over many randomly generated trajectories the cost of the policy starting from x.

(b) *Policy improvement*, which computes the "improved" (or rollout) policy $\tilde{\mu}$ using the one-step lookahead minimization

$$\tilde{\mu}(x) \in \arg\min_{u\in U(x)} E_w\Big\{g(x,u,w) + \alpha J_\mu\big(f(x,u,w)\big)\Big\}, \qquad \text{for all } x.$$

We call $\tilde{\mu}$ "improved policy" because we can generally prove that

$$J_{\tilde{\mu}}(x) \leq J_\mu(x), \qquad \text{for all } x.$$

This cost improvement property will be shown in Chapter 2, Section 2.7, and can be used to show that PI produces an optimal policy in a finite number of iterations under favorable conditions (for example for finite-state discounted problems; see the DP books [Ber12], [Ber17a], or the RL book [Ber19a]).

The rollout algorithm in its pure form is just *a single iteration of the PI algorithm*. It starts from a given base policy μ and produces the rollout policy $\tilde{\mu}$. It may be viewed as approximation in value space with one-step lookahead that uses J_μ as terminal cost function approximation.

It has the advantage that it can be applied on-line by computing the needed values of $J_\mu(x)$ by simulation. By contrast, approximate forms of PI for challenging problems, involving for example neural network training, can only be implemented off-line.

1.4.2 Approximation in Value Space - Infinite Horizon

The approximation in value space approach that we discussed in connection with finite horizon problems can be extended in a natural way to infinite horizon problems. Here in place of J^*, we use an approximation $\tilde{J}$, and generate at any state x, a control $\tilde{\mu}(x)$ by the *one-step lookahead minimization*

$$\tilde{\mu}(x) \in \arg\min_{u \in U(x)} E\Big\{ g(x,u,w) + \alpha \tilde{J}\big(f(x,u,w)\big) \Big\}. \tag{1.36}$$

This minimization yields a stationary policy $\{\tilde{\mu}, \tilde{\mu}, \ldots\}$, with cost function denoted $J_{\tilde{\mu}}$ [i.e., $J_{\tilde{\mu}}(x)$ is the total infinite horizon discounted cost obtained when using $\tilde{\mu}$ starting at state x]; see Fig. 1.4.3. Note that when $\tilde{J} = J^*$, the one-step lookahead policy attains the minimum in the Bellman equation (1.33) and is expected to be optimal. This suggests that one should try to use $\tilde{J}$ as close as possible to J^*, which is generally true as we will argue later.

Naturally an important goal to strive for is that $J_{\tilde{\mu}}$ is close to J^* in some sense. However, for classical control problems, which involve steering and maintaining the state near a desired reference state (e.g., problems with a cost-free and absorbing terminal state, and positive cost for all other states), *stability of $\tilde{\mu}$ may be a principal objective*. In this book, we will discuss stability issues primarily for this one class of problems, and *we will consider the policy $\tilde{\mu}$ to be stable if $J_{\tilde{\mu}}$ is real-valued*, i.e.,

$$J_{\tilde{\mu}}(x) < \infty, \qquad \text{for all states } x.$$

Selecting $\tilde{J}$ so that $\tilde{\mu}$ is stable is a question of major interest for some application contexts, such as model predictive and adaptive control, and will be discussed in the next section within the limited context of linear quadratic problems.

ℓ-Step Lookahead

An important extension of one-step lookahead minimization is *ℓ-step lookahead*, whereby at a state x_k we minimize the cost of the first $\ell > 1$ stages with the future costs approximated by a function $\tilde{J}$ (see the bottom half

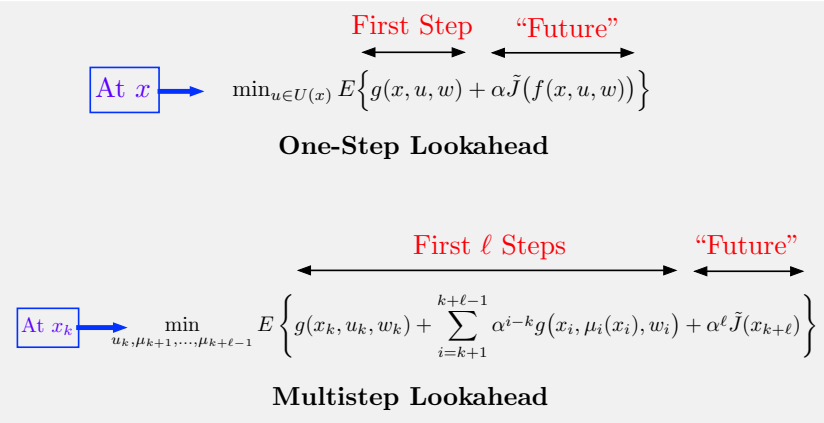

Figure 1.4.3 Schematic illustration of approximation in value space with one-step and ℓ-step lookahead minimization for infinite horizon problems. In the former case, the minimization yields at state x a control $\tilde{u}$, which defines the one-step lookahead policy $\tilde{\mu}$ via

$$\tilde{\mu}(x) = \tilde{u}.$$

In the latter case, the minimization yields a control $\tilde{u}_k$ policies $\tilde{\mu}_{k+1},\ldots,\tilde{\mu}_{k+\ell-1}$. The control $\tilde{u}_k$ is applied at x_k while the remaining sequence $\tilde{\mu}_{k+1},\ldots,\tilde{\mu}_{k+\ell-1}$ is discarded. The control $\tilde{u}_k$ defines the ℓ-step lookahead policy $\tilde{\mu}$.

of Fig. 1.4.3).† This minimization yields a control $\tilde{u}_k$ and a sequence $\tilde{\mu}_{k+1},\ldots,\tilde{\mu}_{k+\ell-1}$. The control $\tilde{u}_k$ is applied at x_k, and defines the ℓ-step lookahead policy $\tilde{\mu}$ via $\tilde{\mu}(x_k) = \tilde{u}_k$, while $\tilde{\mu}_{k+1},\ldots,\tilde{\mu}_{k+\ell-1}$ are discarded. Actually, we may view ℓ-step lookahead minimization as the special case of its one-step counterpart where the lookahead function is the optimal cost function of an $(\ell-1)$-stage DP problem with a terminal cost $\tilde{J}(x_{k+\ell})$ on the state $x_{k+\ell}$ obtained after $\ell-1$ stages.

The motivation for ℓ-step lookahead minimization is that *by increasing the value of ℓ, we may require a less accurate approximation $\tilde{J}$ to obtain good performance*. Otherwise expressed, for the same quality of cost function approximation, better performance may be obtained as ℓ becomes larger. This will be explained visually later, using the formalism of Newton's method in Section 1.5. In particular, for AlphaZero chess, long multistep lookahead is critical for good on-line performance. Another motivation for multistep lookahead is to *enhance the stability properties of the generated on-line policy*, as we will discuss later in Section 1.5. On the other

† On-line play with multistep lookahead minimization (and possibly truncated rollout) is referred to by a number of different names in the RL literature, such as *on-line search*, *predictive learning*, *learning from prediction*, etc; in the model predictive control literature the combined interval of lookahead minimization and truncated rollout is referred as the *prediction interval*.

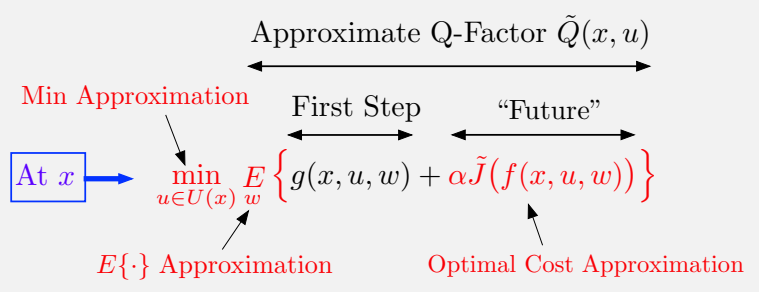

Figure 1.4.4 Approximation in value space with one-step lookahead for infinite horizon problems. There are three potential areas of approximation, which can be considered independently of each other: optimal cost approximation, expected value approximation, and minimization approximation.

hand, solving the multistep lookahead minimization problem, instead of the one-step lookahead counterpart of Eq. (1.36), is more time consuming.

The Three Approximations: Optimal Cost, Expected Value, and Lookahead Minimization Approximations

There are three potential areas of approximation for infinite horizon problems: optimal cost approximation, expected value approximation, and minimization approximation; cf. Fig. 1.4.4. They are similar to their finite horizon counterparts that we discussed in Section 1.3.2. In particular, we have potentially:

(a) A *terminal cost approximation* $\tilde{J}$ *of the optimal cost function* J^*: A major advantage of the infinite horizon context is that only one approximate cost function $\tilde{J}$ is needed, rather than the N functions $\tilde{J}_1, \ldots, \tilde{J}_N$ of the N-step horizon case.

(b) An *approximation of the expected value operation*: This operation can be very time consuming. It may be simplified in various ways. For example some of the random quantities $w_k, w_{k+1}, \ldots, w_{k+\ell-1}$ appearing in the ℓ-step lookahead minimization may be replaced by deterministic quantities; this is another example of the *certainty equivalence approach*, which we discussed in Section 1.3.2.

(c) A *simplification of the minimization operation*: For example in multiagent problems the control consists of multiple components,

$$u = (u^1, \ldots, u^m),$$

with each component u^i chosen by a different agent/decision maker. In this case the size of the control space can be enormous, but it can be simplified in ways that will be discussed later (e.g., choosing

components sequentially, one-agent-at-a-time). This will form the core of our approach to multiagent problems; see Section 1.6.5 and Chapter 2, Section 2.9.

We will next describe briefly various approaches for selecting the terminal cost function approximation.

Constructing Terminal Cost Approximations for On-Line Play

A major issue in value space approximation is the construction of a suitable approximate cost function $\tilde{J}$. This can be done in many different ways, giving rise to some of the principal RL methods.

For example, $\tilde{J}$ may be constructed with sophisticated off-line training methods. Alternatively, the approximate values $\tilde{J}(x)$ may be obtained on-line as needed with truncated rollout, by running an off-line obtained policy for a suitably large number of steps, starting from x, and supplementing it with a suitable, perhaps primitive, terminal cost approximation.

For orientation purposes, let us describe briefly four broad types of approximation. We will return to these approaches later, and we also refer to the RL and approximate DP literature for more detailed discussions.

(a) *Off-line problem approximation*: Here the function $\tilde{J}$ is computed off-line as the optimal or nearly optimal cost function of a simplified optimization problem, which is more convenient for computation. Simplifications may include exploiting decomposable structure, reducing the size of the state space, neglecting some of the constraints, and ignoring various types of uncertainties. For example we may consider using as $\tilde{J}$ the cost function of a related deterministic problem, obtained through some form of certainty equivalence approximation, thus allowing computation of $\tilde{J}$ by gradient-based optimal control methods or shortest path-type methods.
A major type of problem approximation method is *aggregation*, described in Section 3.6, and in the books [Ber12], [Ber19a] and papers [Ber18a], [Ber18b]. Aggregation provides a systematic procedure to simplify a given problem by grouping states together into a relatively small number of subsets, called aggregate states. The optimal cost function of the simpler aggregate problem is computed by exact DP methods, possibly involving the use of simulation. This cost function is then used to provide an approximation $\tilde{J}$ to the optimal cost function J^* of the original problem, using some form of interpolation.

(b) *On-line simulation*: This possibility arises in rollout algorithms for stochastic problems, where we use Monte-Carlo simulation and some suboptimal policy μ (the base policy) to compute (whenever needed) values $\tilde{J}(x)$ that are exactly or approximately equal to $J_\mu(x)$. The policy μ may be obtained by any method, e.g., one based on heuristic reasoning (such as in the case of the traveling salesman Example

1.2.3), or off-line training based on a more principled approach, such as approximate policy iteration or approximation in policy space. Note that while simulation is time-consuming, it is uniquely well-suited for the use of parallel computation. Moreover, it can be simplified through the use of certainty equivalence approximations.

(c) *On-line approximate optimization*. This approach involves the solution of a suitably constructed shorter horizon version of the problem, with a simple terminal cost approximation. It can be viewed as either approximation in value space with multistep lookahead, or as a form of rollout algorithm. It is often used in model predictive control (MPC).

(d) *Parametric cost approximation*, where $\tilde{J}$ is obtained from a given parametric class of functions $J(x, r)$, where r is a parameter vector, selected by a suitable algorithm. The parametric class typically involves prominent characteristics of x 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 (see Chapter 3).

Such methods include approximate forms of PI, as discussed in Section 1.1 in connection with chess and backgammon. The policy evaluation portion of the PI algorithm can be done by approximating the cost function of the current policy using an approximation architecture such as a neural network (see Chapter 3). It can also be done with stochastic iterative algorithms such as TD(λ), LSPE(λ), and LSTD(λ), which are described in the DP book [Ber12] and the RL book [Ber19a]. These methods are somewhat peripheral to our course, and will not be discussed at any length. We note, however, that approximate PI methods do not just yield a parametric approximate cost function $J(x, r)$, but also a suboptimal policy, which can be improved on-line by using (possibly truncated) rollout.

Aside from approximate PI, parametric approximate cost functions $J(x, r)$ may be obtained off-line with methods such as Q-learning, linear programming, and aggregation methods, which are also discussed in the books [Ber12] and [Ber19a].

Let us also mention that for problems with special structure, $\tilde{J}$ may be chosen so that the one-step lookahead minimization (1.36) is facilitated. In fact, under favorable circumstances, the lookahead minimization may be carried out in closed form. An example is when the system is nonlinear, but the control enters linearly in the system equation and quadratically in the cost function, while the terminal cost approximation is quadratic. Then the one-step lookahead minimization can be carried out analytically, because it involves a function that is quadratic in u.

From Off-Line Training to On-Line Play - Infinite Horizon

Generally off-line training will produce either just a cost approximation (as in the case of TD-Gammon), or just a policy (as for example by some approximation in policy space/policy gradient approach), or both (as in the case of AlphaZero). We have already discussed in this section one-step lookahead and multistep lookahead schemes to implement on-line approximation in value space using $\tilde{J}$; cf. Fig. 1.4.3. Let us now consider some additional possibilities, which involve the use of a policy μ that has been obtained off-line (possibly in addition to a terminal cost approximation). Here are some of the main possibilities:

(a) *Given a policy* μ that has been obtained off-line, we may use as terminal cost approximation $\tilde{J}$ the cost function J_μ of the policy. For the case of one-step lookahead, this requires a policy evaluation operation, and can be done on-line, by computing (possibly by simulation) just the values of

$$E\Big\{J_\mu\big(f(x_k,u_k,w_k)\big)\Big\}$$

that are needed [cf. Eq. (1.36)]. For the case of ℓ-step lookahead, the values

$$E\big\{J_\mu(x_{k+\ell})\big\}$$

for all states $x_{k+\ell}$ that are reachable in ℓ steps starting from x_k are needed. This is the simplest form of rollout, and only requires the off-line construction of the policy μ.

(b) *Given a terminal cost approximation* $\tilde{J}$ that has been obtained off-line, we may use it on-line to compute fast when needed the controls of a corresponding one-step or multistep lookahead policy $\tilde{\mu}$. The policy $\tilde{\mu}$ can in turn be used for rollout as in (a) above. In a truncated variation of this scheme, we may also use $\tilde{J}$ to approximate the tail end of the rollout process (an example of this is the rollout-based TD-Gammon algorithm).

(c) *Given a policy* μ *and a terminal cost approximation* $\tilde{J}$, we may use them together in a truncated rollout scheme, whereby the tail end of the rollout with μ is approximated using the cost approximation $\tilde{J}$. This is similar to the truncated rollout scheme noted in (b) above, except that the policy μ is computed off-line rather than on-line using $\tilde{J}$ and one-step or multistep lookahead.

The preceding three possibilities are the principal ones for using the results of off-line training within on-line play schemes. Naturally, there are variations where additional information is computed off-line to facilitate and/or expedite the on-line play algorithm. As an example, in MPC, in addition to a terminal cost approximation, a target tube may need to be computed off-line in order to guarantee that some state constraints can

be satisfied on-line; see the discussion of MPC in Section 1.6.7. Other examples of this type will be noted in the context of specific applications.

Finally, let us note that while we have emphasized approximation in value space with cost function approximation, our discussion applies to Q-factor approximation, involving functions

$$\tilde{Q}(x,u) \approx E\Big\{g(x,u,w) + \alpha J^*\big(f(x,u,w)\big)\Big\}.$$

The corresponding one-step lookahead scheme has the form

$$\tilde{\mu}(x) \in \arg\min_{u\in U(x)} E\Big\{g(x,u,w)+\alpha \min_{u'\in U(f(x,u,w))} \tilde{Q}\big(f(x,u,w),u'\big)\Big\}; \quad (1.37)$$

cf. Eq. (1.36). The second term on the right in the above equation represents the cost function approximation

$$\tilde{J}\big(f(x,u,w)\big) = \min_{u'\in U(f(x,u,w))} \tilde{Q}\big(f(x,u,w),u'\big).$$

The use of Q-factors is common in the "model-free" case where a computer simulator is used to generate samples of w, and corresponding values of g and f. Then, having obtained $\tilde{Q}$ through off-line training, the one-step lookahead minimization in Eq. (1.37) must be performed on-line with the use of the simulator.

1.4.3 Understanding Approximation in Value Space

We will now discuss some of our objectives as we try to get insight into the process of approximation in value space. Clearly, it makes sense to approximate J^* with a function $\tilde{J}$ that is as close as possible to J^*. However, we should also try to understand quantitatively the relation between $\tilde{J}$ and $J_{\tilde{\mu}}$, the cost function of the resulting one-step lookahead (or multistep lookahead) policy $\tilde{\mu}$. Interesting questions in this regard are the following:

(a) *How is the quality of the lookahead policy $\tilde{\mu}$ affected by the quality of the off-line training?* A related question is how much should we care about improving $\tilde{J}$ through a longer and more sophisticated training process, for a given approximation architecture? A fundamental fact that provides a lot of insight in this respect is that *$J_{\tilde{\mu}}$ is the result of a step of Newton's method that starts at $\tilde{J}$ and is applied to the Bellman Eq. (1.33).* This will be the focus of our discussion in the next section, and has been a major point in the narrative of the author's books, [Ber20a] and [Ber22a].

A related fact is that in approximation in value space with multistep lookahead, *$J_{\tilde{\mu}}$ is the result of a step of Newton's method that starts at the function obtained by applying multiple value iterations to $\tilde{J}$.*

(b) *How do simplifications in the multistep lookahead implementation affect $J_{\tilde{\mu}}$?* The Newton step interpretation of approximation in value space leads to an important insight into the special character of the initial step of the multistep lookahead. In particular, *it is only the first step that acts as the Newton step, and needs to be implemented with precision*. The subsequent steps are value iterations, which only serve to enhance the quality of the starting point of the Newton step, and hence *their precise implementation is not critical*.

This idea suggests that simplifications of the lookahead steps after the first can be implemented with relatively small (if any) performance loss for the multistep lookahead policy. Important examples of such simplifications are the use of certainty equivalence (Sections 1.6.7, 2.7.2, 2.8.3), and forms of pruning of the lookahead tree (Section 2.4). In practical terms, simplifications after the first step of the multistep lookahead can save a lot of on-line computation, which can be fruitfully invested in extending the length of the lookahead.

(c) *When is $\tilde{\mu}$ stable?* The question of stability is very important in many control applications where the objective is to keep the state near some reference point or trajectory. Indeed, in such applications, stability is the dominant concern, and optimality is secondary by comparison. Among others, here we are interested to characterize the set of terminal cost approximations $\tilde{J}$ that lead to a stable $\tilde{\mu}$.

(d) *How does the length of lookahead minimization or the length of the truncated rollout affect the stability and quality of the multistep lookahead policy $\tilde{\mu}$?* While it is generally true that the length of lookahead has a beneficial effect on quality, it turns out that it also has a beneficial effect on the stability properties of the multistep lookahead policy, and we are interested in the mechanism by which this occurs.

In what follows we will be keeping in mind these questions. In particular, in the next section, we will discuss them in the context of the simple and convenient linear quadratic problem. Our conclusions, however, hold within a far more general context with the aid of the abstract DP formalism; see the author's books [Ber20a] and [Ber22a] for a broader presentation and analysis, which address these questions in greater detail and generality.

1.5 NEWTON'S METHOD - LINEAR QUADRATIC PROBLEMS

We will now aim to understand the character of the Bellman equation, approximation in value space, and the VI and PI algorithms within the context of an important deterministic problem. This is the classical continuous-spaces problem where the system is linear, with no control constraints, and the cost function is nonnegative quadratic. While this prob-

lem can be solved analytically, it provides a uniquely insightful context for understanding visually the Bellman equation and its algorithmic solution, both exactly and approximately.

In its general form, the problem deals with the system

$$x_{k+1} = Ax_k + Bu_k,$$

where x_k and u_k are elements of the Euclidean spaces $\Re^n$ and $\Re^m$, respectively, A is an $n \times n$ matrix, and B is an $n \times m$ matrix. It is assumed that there are no control constraints. The cost per stage is quadratic of the form

$$g(x, u) = x'Qx + u'Ru,$$

where Q and R are positive definite symmetric matrices of dimensions $n \times n$ and $m \times m$, respectively (all finite-dimensional vectors in this work are viewed as column vectors, and a prime denotes transposition). The analysis of this problem is well known and is given with proofs in several control theory texts, including the author's DP books [Ber17a] and [Ber12].

In what follows, we will focus for simplicity only on the one-dimensional version of the problem, where the system has the form

$$x_{k+1} = ax_k + bu_k; \tag{1.38}$$

cf. Example 1.3.1. Here the state x_k and the control u_k are scalars, and the coefficients a and b are also scalars, with $b \neq 0$. The cost function is undiscounted and has the form

$$\sum_{k=0}^{\infty} (qx_k^2 + ru_k^2), \tag{1.39}$$

where q and r are positive scalars. The one-dimensional case allows a convenient and insightful analysis of the algorithmic issues that are central for our purposes. This analysis generalizes to multidimensional linear quadratic problems and beyond, but requires a more demanding mathematical treatment.

The Riccati Equation and its Justification

The analytical results for our problem may be obtained by taking the limit in the results derived in the finite horizon Example 1.3.1, as the horizon length tends to infinity. In particular, we can show that the optimal cost function is expected to be quadratic of the form

$$J^*(x) = K^* x^2, \tag{1.40}$$

where the scalar K^* solves the equation

$$K = F(K), \tag{1.41}$$

with F defined by

$$F(K) = \frac{a^2 rK}{r + b^2 K} + q. \tag{1.42}$$

This is the limiting form of Eq. (1.19).

Moreover, the optimal policy is linear of the form

$$\mu^*(x) = L^* x, \tag{1.43}$$

where L^* is the scalar given by

$$L^* = -\frac{abK^*}{r + b^2 K^*}. \tag{1.44}$$

To justify Eqs. (1.41)-(1.44), we show that J^* as given by Eq. (1.40), satisfies the Bellman equation

$$J(x) = \min_{u \in \Re} \big\{ qx^2 + ru^2 + J(ax + bu) \big\}, \tag{1.45}$$

and that $\mu^*(x)$, as given by Eqs. (1.43)-(1.44), attains the minimum above for every x when $J = J^*$. Indeed for any quadratic cost function $J(x) = Kx^2$ with $K \geq 0$, the minimization in Bellman's equation (1.45) is written as

$$\min_{u \in \Re} \big\{ qx^2 + ru^2 + K(ax + bu)^2 \big\}. \tag{1.46}$$

Thus it involves minimization of a positive definite quadratic in u and can be done analytically. By setting to 0 the derivative with respect to u of the expression in braces in Eq. (1.46), we obtain

$$0 = 2ru + 2bK(ax + bu),$$

so the minimizing control and corresponding policy are given by

$$\mu_K(x) = L_K x, \tag{1.47}$$

where

$$L_K = -\frac{abK}{r + b^2 K}. \tag{1.48}$$

By substituting this control, the minimized expression (1.46) takes the form

$$\Big(q + rL_K^2 + K(a + bL_K)^2 \Big) x^2.$$

After straightforward algebra, using Eq. (1.48) for L_K, it can be verified that this expression is written as $F(K)x^2$, with F given by Eq. (1.42). Thus when $J(x) = Kx^2$, the Bellman equation (1.45) takes the form

$$Kx^2 = F(K)x^2$$

or equivalently $K = F(K)$ [cf. Eq. (1.41)].

In conclusion, when restricted to quadratic functions $J(x) = Kx^2$ with $K \geq 0$, the Bellman equation (1.45) is equivalent to the equation

$$K = F(K) = \frac{a^2 rK}{r + b^2 K} + q. \tag{1.49}$$

We refer to this equation as the *Riccati equation*† and to the function F as the *Riccati operator*.‡ Moreover, the policy corresponding to K^*, as per Eqs. (1.47)-(1.48), attains the minimum in Bellman's equation, and is given by Eqs. (1.43)-(1.44).

The Riccati equation can be visualized and solved graphically as illustrated in Fig. 1.5.1. As shown in the figure, the quadratic coefficient K^* that corresponds to the optimal cost function J^* [cf. Eq. (1.40)] is the unique solution of the Riccati equation $K = F(K)$ within the nonnegative real line.

The Riccati Equation for a Stable Linear Policy

We can also characterize the cost function of a policy μ that is linear of the form $\mu(x) = Lx$, and is also stable, in the sense that the scalar L satisfies $|a + bL| < 1$, so that the corresponding closed-loop system

$$x_{k+1} = (a + bL)x_k$$

is stable (its state x_k converges to 0 as $k \to \infty$). In particular, we can show that its cost function has the form

$$J_\mu(x) = K_L x^2,$$

† This is an algebraic form of the Riccati differential equation, which was invented in its one-dimensional form by count Jacopo Riccati in the 1700s, and has played an important role in control theory. It has been studied extensively in its differential and difference matrix versions; see the book by Lancaster and Rodman [LR95], and the paper collection by Bittanti, Laub, and Willems [BLW91], which also includes a historical account by Bittanti [Bit91] of Riccati's remarkable life and accomplishments.

‡ The Riccati operator is a special case of the *Bellman operator*, denoted by T, which transforms a function J into the right side of Bellman's equation:

$$(TJ)(x) = \min_{u \in U(x)} E_w \Big\{ g(x,u,w) + \alpha J\big(f(x,u,w)\big) \Big\}, \qquad \text{for all } x.$$

Thus the Bellman operator T transforms a function J of x into another function TJ also of x. Bellman operators allow a succinct abstract description of the problem's data, and are fundamental in the theory of abstract DP (see the author's monographs [Ber22a] and [Ber22b]). We may view the Riccati operator as the restriction of the Bellman operator to the subspace of quadratic functions of x.

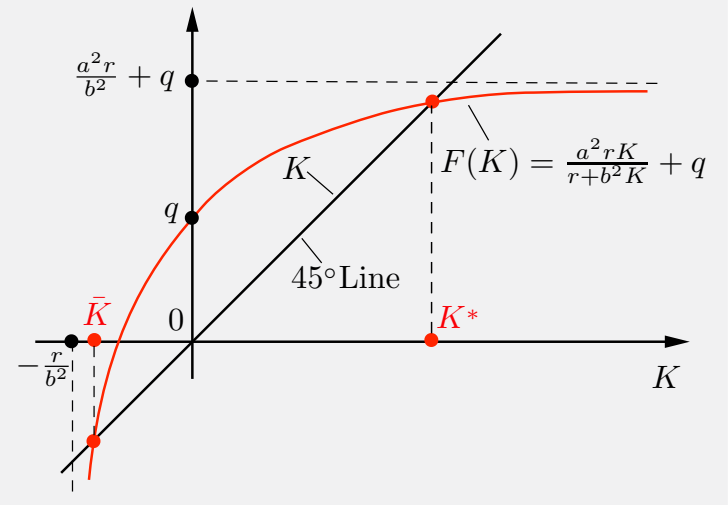

Figure 1.5.1 Graphical construction of the solutions of the Riccati equation (1.41)-(1.42) for the linear quadratic problem. The optimal cost function is $J^*(x) = K^*x^2$, where the scalar K^* solves the fixed point equation $K = F(K)$, with F being the function given by

$$F(K) = \frac{a^2rK}{r + b^2K} + q.$$

Note that F is concave and monotonically increasing in the interval $(-r/b^2, \infty)$ and "flattens out" as $K \to \infty$, as shown in the figure. The quadratic Riccati equation $K = F(K)$ also has another solution, denoted by $\bar{K}$, which is negative and is thus of no interest.

where K_L solves the equation

$$K = F_L(K), \tag{1.50}$$

with F_L defined by

$$F_L(K) = (a + bL)^2K + q + rL^2. \tag{1.51}$$

This equation is called the *Riccati equation for the stable policy* $\mu(x) = Lx$. It is illustrated in Fig. 1.5.2, and it is linear, with linear coefficient $(a+bL)^2$ that is strictly less than 1. Hence the line that represents the graph of F_L intersects the 45-degree line at a unique point, which defines the quadratic cost coefficient K_L.

The Riccati equation (1.50)-(1.51) for $\mu(x) = Lx$ may be justified by verifying that it is in fact the Bellman equation for μ,

$$J(x) = (q + rL^2)x^2 + J\big((a + bL)x\big),$$

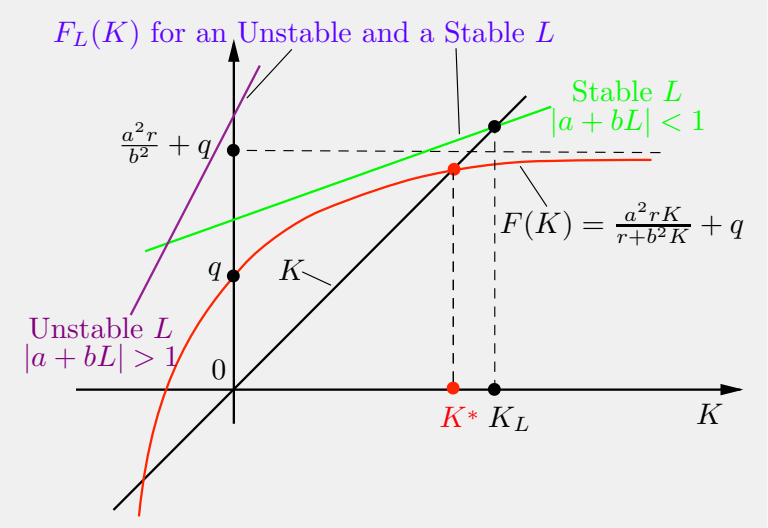

Figure 1.5.2 Illustration of the construction of the cost function of a linear policy $\mu(x) = Lx$, which is stable, i.e., $|a + bL| < 1$. The cost function $J_\mu(x)$ has the form

$$J_\mu(x) = K_L x^2,$$

with K_L obtained as the unique solution of the linear equation $K = F_L(K)$, where

$$F_L(K) = (a + bL)^2 K + q + rL^2,$$

is the Riccati equation operator corresponding to $\mu(x) = Lx$. If μ is not stable, i.e., $|a + bL| \geq 1$, we have $J_\mu(x) = \infty$ for all $x \neq 0$, but the equation has $K = F_L(K)$ still has a solution that is of no interest within our context.

[cf. Eq. (1.34)], restricted to quadratic functions of the form $J(x) = Kx^2$.

We note, however, that $J_\mu(x) = K_L x^2$ is the solution of the Riccati equation (1.50)-(1.51) only when $\mu(x) = Lx$ is stable. If μ is not stable, i.e., $|a + bL| \geq 1$, then (since $q > 0$ and $r > 0$) we have $J_\mu(x) = \infty$ for all $x \neq 0$. Then, the Riccati equation (1.50)-(1.51) is still defined, but its solution is negative and is of no interest within our context.

Value Iteration

The VI algorithm for our linear quadratic problem is given by

$$J_{k+1}(x) = \min_{u \in \Re} \big\{ qx^2 + ru^2 + J_k(ax + bu) \big\}.$$

When J_k is quadratic of the form $J_k(x) = K_k x^2$ with $K_k \geq 0$, it can be seen that the VI iterate J_{k+1} is also quadratic of the form $J_{k+1}(x) = K_{k+1} x^2$, where

$$K_{k+1} = F(K_k),$$

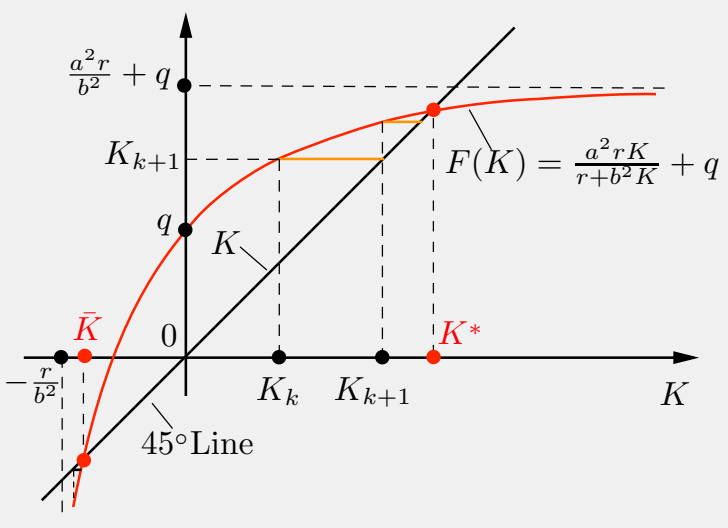

Figure 1.5.3 Graphical illustration of value iteration for the linear quadratic problem. It has the form $K_{k+1} = F(K_k)$, where F is the Riccati operator,

$$F(K) = \frac{a^2rK}{r + b^2K} + q.$$

The algorithm converges to K^* starting from any $K_0 \geq 0$.

with F being the Riccati operator of Eq. (1.49). The algorithm is illustrated in Fig. 1.5.3. As can be seen from the figure, when starting from any $K_0 \geq 0$, the algorithm generates a sequence $\{K_k\}$ of nonnegative scalars that converges to K^*.

1.5.1 Visualizing Approximation in Value Space - Region of Stability

The use of Riccati equations allows insightful visualization of approximation in value space. This visualization, although specialized to linear quadratic problems, is consistent with related visualizations for more general infinite horizon problems; this is a recurring theme in what follows. In particular, in the books [Ber20a] and [Ber22a], Bellman operators, which define the Bellman equations, are used in place of Riccati operators, which define the Riccati equations.

In summary, we will aim to show that:

(a) Approximation in value space with one-step lookahead can be viewed as a Newton step for solving the Bellman equation, and maps the terminal cost function approximation $\tilde{J}$ to the cost function $J_{\tilde{\mu}}$ of the one-step lookahead policy; see Fig. 1.5.4.

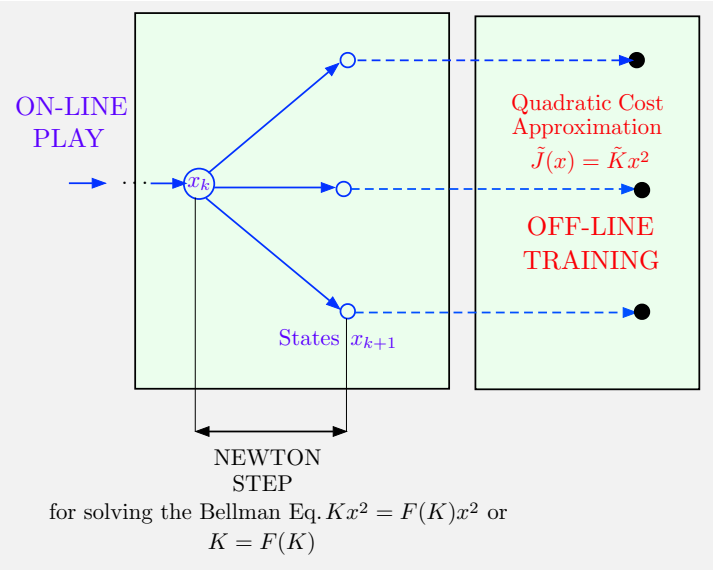

Figure 1.5.4 Illustration of the interpretation of approximation in value space with one-step lookahead as a Newton step that maps $\tilde{J}$ to the cost function $J_{\tilde{\mu}}$ of the one-step lookahead policy.

(b) Approximation in value space with multistep lookahead and truncated rollout can be viewed as a Newton step for solving the Bellman equation, and maps the result of multiple VI iterations starting with the terminal cost function approximation $\tilde{J}$ to the cost function $J_{\tilde{\mu}}$ of the multistep lookahead policy; see Fig. 1.5.5.

Our derivation will be given for the one-dimensional linear quadratic problem, but *applies far more generally*. The reason is that the Bellman equation is valid universally in DP, and the corresponding Bellman operator has a concavity property that is well-suited for the application of Newton's method; see the books [Ber20a] and [Ber22a], where the connection of approximation in value space with Newton's method was first developed in detail.

Let us consider one-step lookahead minimization with any terminal cost function approximation of the form $\tilde{J}(x) = Kx^2$, where $K \geq 0$. We have derived the one-step lookahead policy $\mu_K(x)$ in Eqs. (1.47)-(1.48), by minimizing the right side of Bellman's equation when $J(x) = Kx^2$:

$$\min_{u \in \Re} \left\{ qx^2 + ru^2 + K(ax + bu)^2 \right\}.$$

We can break this minimization into a sequence of two minimizations as follows:

$$F(K)x^2 = \min_{L \in \Re} \min_{u = Lx} \left\{ qx^2 + ru^2 + K(ax+bu)^2 \right\} = \min_{L \in \Re} \left\{ q + bL + K(a+bL)^2 \right\} x^2.$$

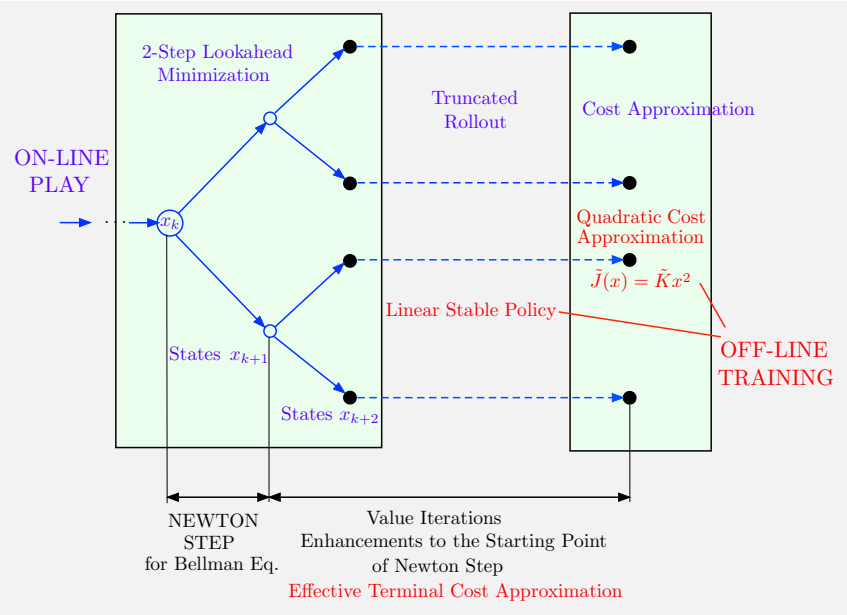

Figure 1.5.5 Illustration of the interpretation of approximation in value space with multistep lookahead and truncated rollout as a Newton step, which maps the result of multiple VI iterations starting with the terminal cost function approximation $\tilde{J}$ to the cost function $J_{\tilde{\mu}}$ of the multistep lookahead policy.

From this equation, it follows that

$$F(K) = \min_{L \in \Re} F_L(K), \tag{1.52}$$

where the function $F_L(K)$ is defined by

$$F_L(K) = (a + bL)^2 K + q + bL. \tag{1.53}$$

Figure 1.5.6 illustrates the relation (1.52)-(1.53), and shows how the graph of *the Riccati operator F can be obtained as the lower envelope of the linear operators F_L*, as L ranges over the real numbers.

One-Step Lookahead Minimization and Newton's Method

Let us now fix the terminal cost function approximation to some $\tilde{K}x^2$, where $\tilde{K} \geq 0$, and consider the corresponding one-step lookahead policy, which we will denote by $\tilde{\mu}$. Figure 1.5.7 illustrates the corresponding linear function $F_{\tilde{L}}$, and shows that its graph is a tangent line to the graph of F at the point K [cf. Fig. 1.5.6 and Eq. (1.53)].

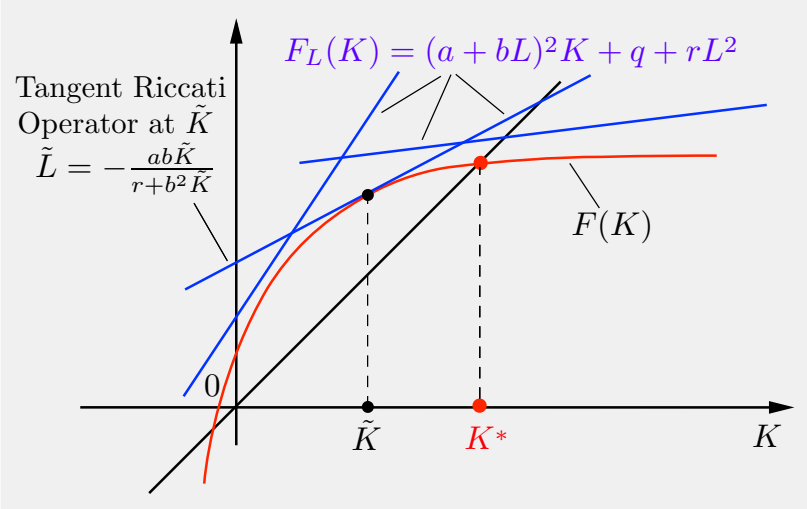

Figure 1.5.6 Illustration of how the graph of the Riccati operator F can be obtained as the lower envelope of the linear operators

$$F_L(K) = (a + bL)^2 K + q + bL,$$

as L ranges over the real numbers. We have

$$F(K) = \min_{L \in \Re} F_L(K);$$

cf. Eq. (1.52). Moreover, for any fixed $\tilde{K}$, the scalar $\tilde{L}$ that attains the minimum is given by

$$\tilde{L} = -\frac{ab\tilde{K}}{r + b^2\tilde{K}}$$

[cf. Eq. (1.48)], and is such that the line corresponding to the graph of $F_{\tilde{L}}$ is tangent to the graph of F at $\tilde{K}$, as shown in the figure.

Thus the function $F_{\tilde{L}}$ can be viewed as a linearization of F at the point K, and defines a linearized problem: to find a solution of the equation

$$K = F_{\tilde{L}}(K) = q + b\tilde{L}^2 + K(a + b\tilde{L})^2.$$

The important point now is that *the solution of this equation, denoted $K_{\tilde{L}}$, is the same as the one obtained from a single iteration of Newton's method for solving the Riccati equation, starting from the point $\tilde{K}$*. This is illustrated in Fig. 1.5.7, and is also justified analytically in Exercise 1.7.

To explain this connection, we note that the classical form of Newton's method for solving a fixed point problem of the form $y = T(y)$, where y is an n-dimensional vector, operates as follows: At the current iterate y_k, we linearize T and find the solution y_{k+1} of the corresponding linear fixed

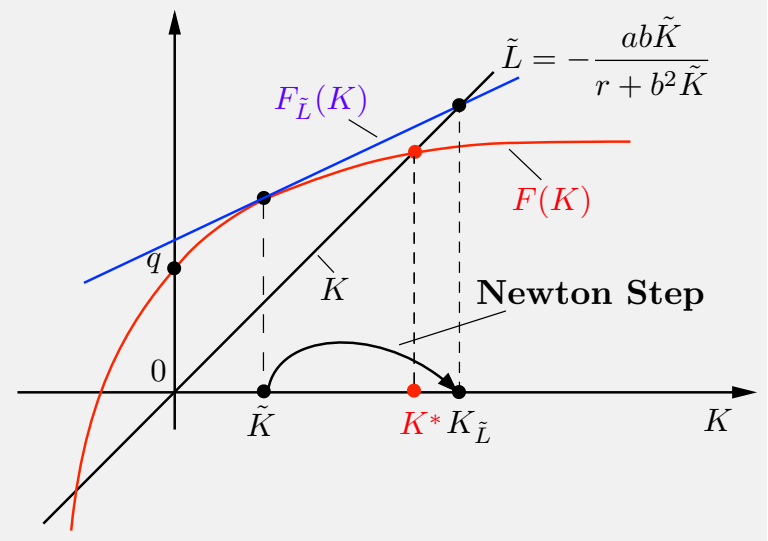

Figure 1.5.7 Illustration of approximation in value space with one-step lookahead for the linear quadratic problem. Given a terminal cost approximation $\tilde{J} = \tilde{K}x^2$, we compute the corresponding linear policy $\tilde{\mu}(x) = \tilde{L}x$, where

$$\tilde{L} = -\frac{ab\tilde{K}}{r + b^2\tilde{K}},$$

and the corresponding cost function $K_{\tilde{L}}x^2$, using the Newton step shown.

point problem. Assuming T is differentiable, the linearization is obtained by using a first order Taylor expansion:

$$y_{k+1} = T(y_k) + \frac{\partial T(y_k)}{\partial y}(y_{k+1} - y_k),$$

where $\partial T(y_k)/\partial y$ is the $n \times n$ Jacobian matrix of T evaluated at the vector y_k, as indicated in Fig. 1.5.7.

The most commonly given convergence rate property of Newton's method is *quadratic convergence*. It states that near the solution y^*, we have

$$\|y_{k+1} - y^*\| = O\big(\|y_k - y^*\|^2\big),$$

where $\|\cdot\|$ is the Euclidean norm, and holds assuming the Jacobian matrix exists and is Lipschitz continuous (see [Ber16], Section 1.4). There are extensions of Newton's method that are based on solving a linearized system at the current iterate, but relax the differentiability requirement to piecewise differentiability, and/or component concavity, while maintaining the either a quadratic or a similarly fast superlinear convergence property of the method; see the monograph [Ber22a] (Appendix A) and the paper [Ber22c], which also provide a convergence analysis.

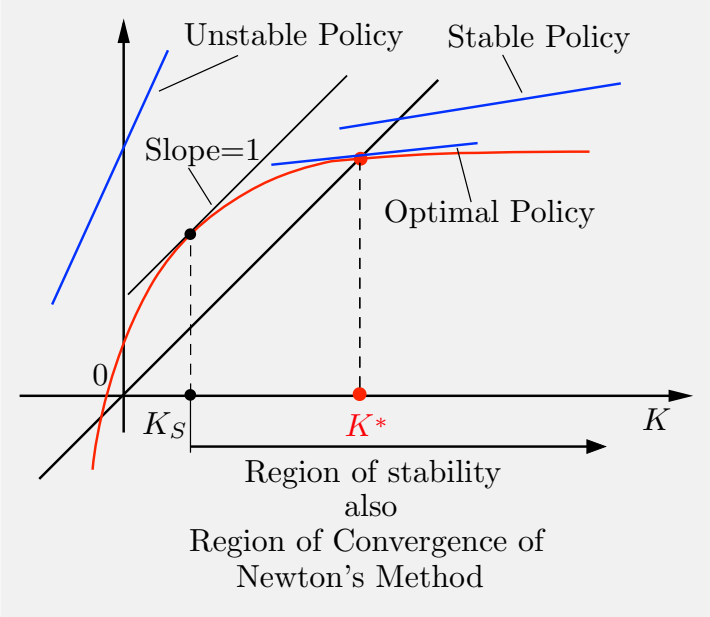

Figure 1.5.8 Illustration of the region of stability, i.e., the set of $K \geq 0$ such that the one-step lookahead policy μ_K is stable. This is also the set of initial conditions for which Newton's method converges to K^* asymptotically.

Note also that if the one-step lookahead policy is stable, i.e., $|a+b\tilde{L}| < 1$, then $K_{\tilde{L}}$ is the quadratic cost coefficient of its cost function, i.e.,

$$J_{\tilde{\mu}}(x) = K_{\tilde{L}}x^2.$$

The reason is that $J_{\tilde{\mu}}$ solves the Bellman equation for policy $\tilde{\mu}$. On the other hand, if $\tilde{\mu}$ is not stable, then in view of the positive definite quadratic cost per stage, we have $J_{\tilde{\mu}}(x) = \infty$ for all $x \neq 0$.

Multistep Lookahead

In the case of ℓ-step lookahead minimization, a similar Newton step interpretation is possible. Instead of linearizing F at $\tilde{K}$, we linearize at

$$K_{\ell-1} = F^{\ell-1}(\tilde{K}),$$

i.e., the result of $\ell - 1$ successive applications of F starting with $\tilde{K}$. Each application of F corresponds to a value iteration. Thus *the effective starting point for the Newton step is* $F^{\ell-1}(\tilde{K})$. Figure 1.5.9 depicts the case $\ell = 2$.

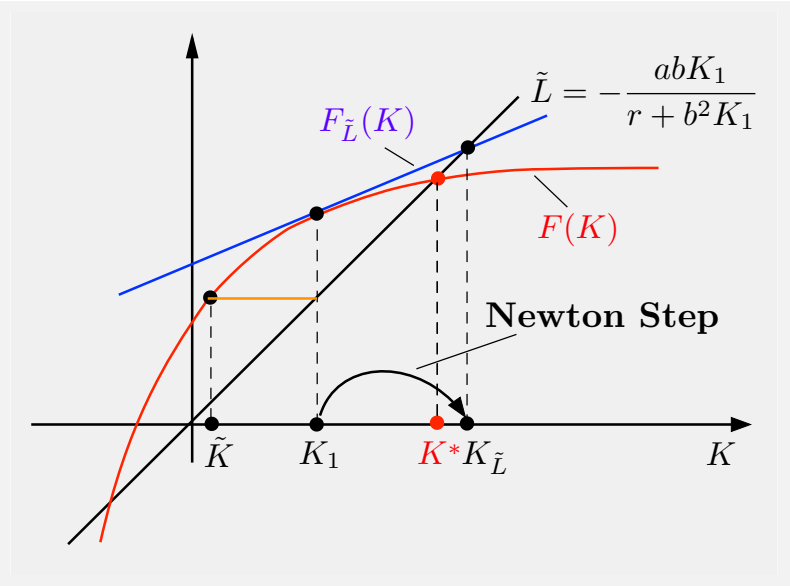

Figure 1.5.9 Illustration of approximation in value space with two-step lookahead for the linear quadratic problem. Starting with a terminal cost approximation $\tilde{J} = \tilde{K}x^2$, we obtain K_1 using a single value iteration. We then compute the corresponding linear policy $\tilde{\mu}(x) = \tilde{L}x$, where

$$\tilde{L} = -\frac{abK_1}{r + b^2K_1}$$

and the corresponding cost function $K_{\tilde{L}}x^2$, using the Newton step shown. The figure shows that for any $K \geq 0$, the corresponding ℓ-step lookahead policy will be stable for all ℓ larger than some threshold.

Region of Stability

It is also useful to define the *region of stability* as the set of $K \geq 0$ such that

$$|a + bL_K| < 1,$$

where L_K is the linear coefficient of the one-step lookahead policy corresponding to K; cf. Eq. (1.48). The region of stability may also be viewed as *the region of convergence of Newton's method*. It is the set of starting points K for which Newton's method, applied to the Riccati equation $F = F(K)$, converges to K^* asymptotically, and with a quadratic convergence rate (asymptotically as $K \to K^*$). Note that for our one-dimensional problem, the region of stability is the interval (K_S, ∞) that is characterized by the single point K_S where F has derivative equal to 1; see Fig. 1.5.8.

Riccati Equation Formulas for One-Dimensional Problems

Riccati equation for minimization [cf. Eqs. (1.41) and (1.42)]

$$K = F(K), \qquad F(K) = \frac{a^2 r K}{r + b^2 K} + q.$$

Riccati equation for a linear policy $\mu(x) = Lx$

$$K = F_L(K), \qquad F_L(K) = (a + bL)^2 K + q + rL^2.$$

Cost coefficient K_L **of a stable linear policy** $\mu(x) = Lx$

$$K_L = \frac{q + rL^2}{1 - (a + bL)^2}.$$

Linear coefficient L_K **of the one-step lookahead linear policy** μ_K **for** K **in the region of stability** [cf. Eq. (1.48)]

$$L_K = \arg\min_L F_L(K) = -\frac{abK}{r + b^2 K}.$$

Quadratic cost coefficient $\tilde{K}$ **of a one-step lookahead linear policy** μ_K **for** K **in the region of stability**

Obtained as the solution of the linearized Riccati equation

$$\tilde{K} = F_{L_K}(\tilde{K}),$$

or equivalently by a Newton iteration starting from K.

For multidimensional problems, the region of stability may not be characterized as easily. Still, however, it is generally true that *the region of stability is enlarged as the length of the lookahead increases*.

Indeed, with increased lookahead, the effective starting point

$$F^{\ell-1}(\tilde{K})$$

is pushed more and more within the region of stability. In particular, *for any given* $K \geq 0$, *the corresponding* ℓ*-step lookahead policy will be stable for all* ℓ *larger than some threshold*; see Fig. 1.5.9. The book [Ber22a],

Section 3.3, contains a broader discussion of the region of stability and the role of multistep lookahead in enhancing it; see also Exercise 1.8.

Newton Step Interpretation of Approximation in Value Space in General Infinite Horizon Problems

The interpretation of approximation in value space as a Newton step, and related notions of stability that we have discussed in this section admit a broad generalization to the infinite horizon problems that we consider in this book and beyond. The key fact in this respect is that our DP problem formulation allows arbitrary state and control spaces, both discrete and continuous, and can be extended even further to general abstract models with a DP structure; see the abstract DP book [Ber22b].

Within this context, the Riccati operator is replaced by an abstract Bellman operator, and valuable insight can be obtained from graphical interpretations of the Bellman equation, the VI and PI algorithms, one-step and multistep approximation in value space, the region of stability, and exceptional behavior; see the book [Ber22a] for an extensive discussion. Naturally, the graphical interpretations and visualizations are limited to one dimension. However, the visualizations provide insight and motivate conjectures and mathematical analysis, much of which is given in the book [Ber20a].

The Importance of the First Step in Multistep Lookahead

The Newton step interpretation of approximation in value space leads to an important insight into the special character of the initial step in ℓ-step lookahead implementations. In particular, *it is only the first step that acts as the Newton step*, and needs to be implemented with precision; cf. Fig. 1.5.5. The subsequent $\ell-1$ steps are a sequence of value iterations starting with $\tilde{J}$, and only serve to enhance the quality of the starting point of the Newton step. As a result, *their precise implementation is not critical*, a major point in the narrative of the author's book [Ber22a].

This idea suggests that we can simplify (within reason) the lookahead steps after the first with small (if any) performance loss for the multistep lookahead policy. An important example of such a simplification is the use of certainty equivalence, which will be discussed later in various contexts (Sections 1.6.7, 2.7.2, 2.8.3). Other possibilities include the "pruning" of the lookahead tree after the first step; see Section 2.4. In practical terms, simplifications after the first step of the multistep lookahead can save a lot of on-line computation, which can be fruitfully invested in extending the length of the lookahead. This insight is supported by substantial computational experimentation, starting with the paper by Bertsekas and Castañon [BeC98], which verified the beneficial effect of using certainty equivalence after the first step.

1.5.2 Rollout and Policy Iteration

We will now consider the rollout algorithm for the linear quadratic problem, starting from a linear stable base policy μ. It generates the rollout policy $\tilde{\mu}$ by using a policy improvement operation, which by definition, yields the one-step lookahead policy that corresponds to terminal cost approximation $\tilde{J} = J_\mu$. Figure 1.5.10 illustrates the rollout algorithm. It can be seen from the figure that the rollout policy is in fact an improved policy, in the sense that $J_{\tilde{\mu}}(x) \leq J_\mu(x)$ for all x. Among others, this implies that the rollout policy is stable, since μ is assumed stable so that $J_\mu(x) < \infty$ for all x.

Since the rollout policy is a one-step lookahead policy, it can also be described using the formulas that we developed earlier in this section. In particular, let the base policy have the form

$$\mu^0(x) = L_0 x,$$

where L_0 is a scalar. We require that the base policy must be stable, i.e., $|a + bL_0| < 1$. From our earlier calculations, we have that the cost function of μ^0 is

$$J_{\mu^0}(x) = K_0 x^2, \tag{1.54}$$

where

$$K_0 = \frac{q + rL_0^2}{1 - (a + bL_0)^2}. \tag{1.55}$$

Moreover, the rollout policy μ^1 has the form $\mu^1(x) = L_1 x$, where

$$L_1 = -\frac{abK_0}{r + b^2 K_0}; \tag{1.56}$$

cf. Eqs. (1.47)-(1.48).

The PI algorithm is simply the repeated application of nontruncated rollout, and generates a sequence of stable linear policies $\{\mu^k\}$. By replicating our earlier calculations, we see that the policies have the form

$$\mu^k(x) = L_k x, \qquad k = 0, 1, \ldots,$$

where L_k is generated by the iteration

$$L_{k+1} = -\frac{abK_k}{r + b^2 K_k},$$

with K_k given by

$$K_k = \frac{q + rL_k^2}{1 - (a + bL_k)^2},$$

[cf. Eqs. (1.55)-(1.56)].

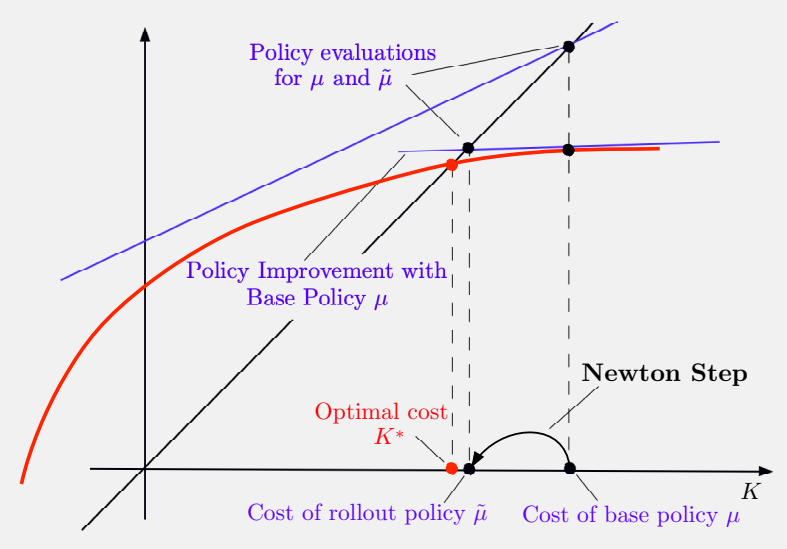

Figure 1.5.10 Illustration of the rollout algorithm for the linear quadratic problem. Starting from a linear stable base policy μ, it generates a stable rollout policy $\tilde{\mu}$. The quadratic cost coefficient of $\tilde{\mu}$ is obtained from the quadratic cost coefficient of μ with a Newton step for solving the Riccati equation.

The corresponding cost function sequence has the form

$$J_{\mu^k}(x) = K_k x^2;$$

cf. Eq. (1.54). Part of the classical linear quadratic theory is that J_{μ^k} converges to the optimal cost function J^*, while the generated sequence of linear policies $\{\mu^k\}$, where $\mu^k(x) = L_k x$, converges to the optimal policy, assuming that the initial policy is linear and stable. The convergence rate of the sequence $\{K_k\}$ is quadratic, as indicated earlier. This result was proved by Kleinman [Kle68] for the continuous-time multidimensional version of the linear quadratic problem, and it was extended later to more general problems; see the references given in the books [Ber20a] and [Ber22a] (Kleinman gives credit to Bellman and Kalaba [BeK65] for the one-dimensional version of his results).

Truncated Rollout

An m-step truncated rollout scheme with a stable linear base policy $\mu(x) = Lx$, one-step lookahead minimization, and terminal cost approximation $\tilde{J}(x) = \tilde{K}x^2$ is geometrically interpreted as in Fig. 1.5.11. The truncated rollout policy $\tilde{\mu}$ is obtained by starting at $\tilde{K}$, executing m VI steps using μ, followed by a Newton step for solving the Riccati equation.

We mentioned some interesting performance issues in our discussion of truncated rollout in Section 1.1. In particular we noted that:

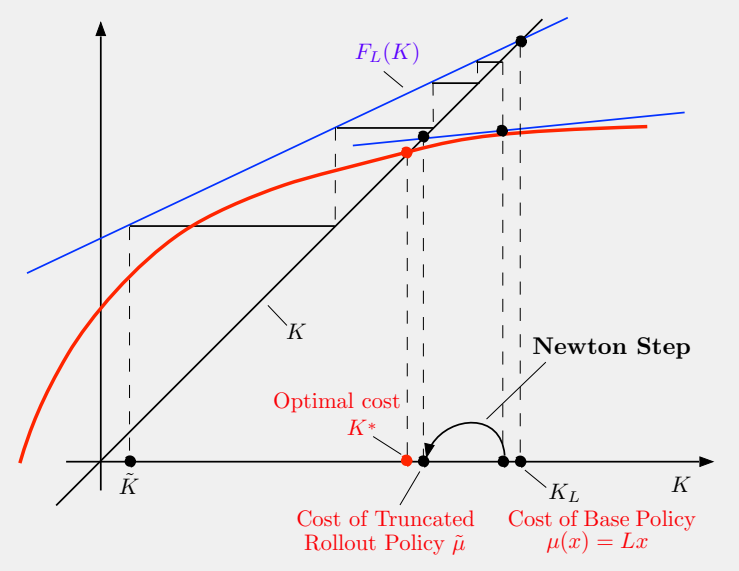

Figure 1.5.11 Illustration of truncated rollout with one-step lookahead minimization, a stable base policy $\mu(x) = Lx$, and terminal cost approximation $\tilde{K}$ for the linear quadratic problem. In this figure the number of rollout steps is $m = 4$.

(a) Lookahead by rollout may be an economic substitute for lookahead by minimization, in the sense that it may achieve a similar performance for the truncated rollout policy at significantly reduced computational cost.

(b) Lookahead by rollout with a stable policy has a beneficial effect on the stability properties of the lookahead policy.

These statements are difficult to establish analytically in some generality. However, they can be intuitively understood in the context with our one-dimensional linear quadratic problem, using geometrical constructions like the one of Fig. 1.5.11. They are also consistent with the results of computational experimentation. We refer to the monograph [Ber22a] for further discussion.

1.5.3 Local and Global Error Bounds for Approximation in Value Space

In approximation in value space, an important analytical issue is to quantify the level of suboptimality of the one-step or multistep lookahead policy obtained. It is thus important to understand the character of the critical mapping between the approximation error $\tilde{J} - J^*$ and the performance error $J_{\tilde{\mu}} - J^*$, where as earlier, $J_{\tilde{\mu}}$ is the cost function of the lookahead policy $\tilde{\mu}$ and J^* is the optimal cost function.

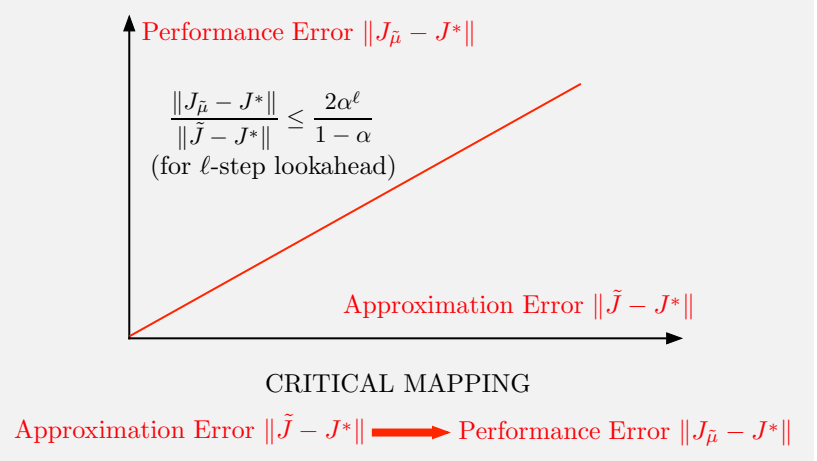

Figure 1.5.12 Illustration of the linear error bound (1.58) for ℓ-step lookahead approximation in value space. For $\ell = 1$, we obtain the one-step bound (1.57).

There is a classical one-step lookahead error bound for the case of an α-discounted problem with finite state space X, which has the form

$$\|J_{\tilde{\mu}} - J^*\| \leq \frac{2\alpha}{1-\alpha}\|\tilde{J} - J^*\|; \tag{1.57}$$

where $\|\cdot\|$ denotes the maximum norm,

$$\|J_{\tilde{\mu}} - J^*\| = \max_{x \in X} \big|J_{\tilde{\mu}}(x) - J^*(x)\big|, \qquad \|\tilde{J} - J^*\| = \max_{x \in X} \big|\tilde{J}(x) - J^*(x)\big|;$$

see e.g., [Ber19a], Prop. 5.1.1. The bound (1.57) predicts a linear relation between the size of the approximation error $\|\tilde{J} - J^*\|$ and the performance error $\|J_{\tilde{\mu}} - J^*\|$. For a generalization, we may view ℓ-step lookahead as one-step lookahead with a terminal cost function $T^{\ell-1}\tilde{J}$, i.e., $\tilde{J}$ transformed by $\ell - 1$ value iterations. We then obtain the ℓ-step bound

$$\|J_{\tilde{\mu}} - J^*\| \leq \frac{2\alpha^\ell}{1-\alpha}\|\tilde{J} - J^*\|. \tag{1.58}$$

The linear bounds (1.57)-(1.58) are illustrated in Fig. 1.5.12, and apply beyond the α-discounted case, to problems where the Bellman equation involves a contraction mapping over a subset of functions; see the RL book [Ber19a], Section 5.9.1, or the abstract DP book [Ber22b], Section 2.2.

Unfortunately, the linear error bounds are very conservative, and do not reflect practical reality, even qualitatively so. The main reason is that they are *global* error bounds, i.e., they hold for all $\tilde{J}$, even the worst possible. In practice, $\tilde{J}$ is often chosen sufficiently close to J^*, so that the error $J_{\tilde{\mu}} - J^*$ behaves consistently with the superlinear convergence rate of the Newton

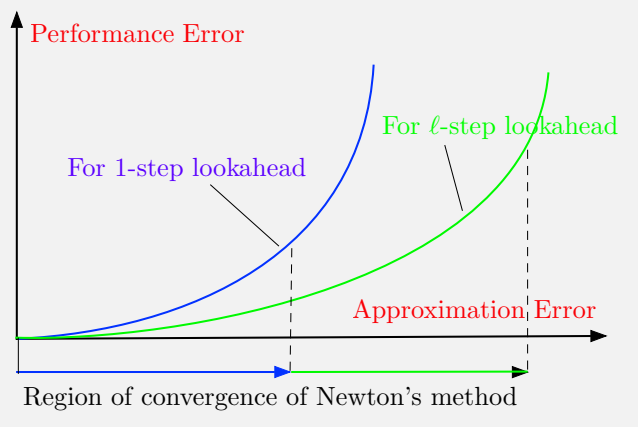

Figure 1.5.13 Schematic illustration of the correct superlinear error bound (1.59) for the case of ℓ-step lookahead approximation in value space scheme. The performance error rises rapidly outside the region of convergence of Newton's method [the illustration in the figure is not realistic; in fact the region of convergence is not bounded as it contains lines of the form γe, where γ is a scalar and e is the unit vector (all components equal to 1)]. Note that this region expands as the size of lookahead ℓ increases. Furthermore, with long enough lookahead ℓ, the ℓ-step lookahead policy $\tilde{\mu}$ can be shown to be *exactly optimal* for many problems of interest; this is a theoretical result, which holds for α-discounted finite-state problems, among others, and has been known since the 60s-70s (Prop. 2.3.1 of [Ber22a] proves a general form of this result that applies beyond discounted problems).

step that starts at $\tilde{J}$. In other words, for $\tilde{J}$ relatively close to J^*, we have the *local* estimate

$$\|J_{\tilde{\mu}} - J^*\| = o\big(\|\tilde{J} - J^*\|\big), \tag{1.59}$$

illustrated in Fig. 1.5.13.

A salient characteristic of this superlinear relation is that the performance error rises rapidly outside the region of superlinear convergence of Newton's method. Note that small improvements in the quality of $\tilde{J}$ (e.g., better sampling methods, improved confidence intervals, and the like, without changing the approximation architecture) have *little effect, both inside and outside the region of convergence*.

In practical terms, *there is often a huge difference, both quantitative and qualitative, between the linear error bounds (1.57)-(1.58) and the superlinear error bound (1.59)*. Moreover, the linear bounds, despite their popularity in academia, often misdirect academic research and confuse practitioners.† Note that as we have mentioned earlier, the qualitative

† A study by Laidlaw, Russell, and Dragan [LRD23] has assessed the practical performance of popular methods on a set of 155 problems, and found wide disparities relative to theoretical predictions. Quoting from this paper: "we find that prior bounds do not correlate well with when deep RL succeeds vs. fails."

performace behavior predicted in Fig. 1.5.13 holds very broadly in approximation in value space, because it relies on notions of abstract DP that apply very generally, for arbitrary state spaces, control spaces, and other problem characteristics; see the abstract DP book [Ber22a].

We illustrate the failure of the linear error bound (1.57) to predict the performance of the one-step lookahead policy with an example given in Fig. 1.5.14.

Example 1.5.1 (Global and Actual Error Bounds for a Linear Quadratic Problem)

Consider the one-dimensional linear quadratic problem, involving the system $x_{k+1} = ax_k + bu_k$, and the cost per stage $qx_k^2 + ru_k^2$. We will consider one-step lookahead, and a quadratic cost function approximation

$$\tilde{J}(x) = \tilde{K}x^2,$$

with $\tilde{K}$ within the region of stability, which is some interval of the form (S, ∞). The Riccati operator is

$$F(K) = \frac{a^2 rK}{r + b^2 K} + q,$$

and the one-step lookahead policy $\tilde{\mu}$ has cost function

$$J_{\tilde{\mu}}(x) = K_{\tilde{\mu}}x^2,$$

where $K_{\tilde{\mu}}$ is obtained by applying one step of Newton's method for solving the Riccati equation $K = F(K)$, starting at $K = \tilde{K}$.

Let S be the boundary of the region of stability, i.e., the value of K at which the derivative of F with respect to K is equal to 1:

$$\frac{\partial F(K)}{\partial K}\Big|_{K=S} = 1.$$

Then the Riccati operator F is a contraction within any interval $[\overline{S}, \infty)$ with $\overline{S} > S$, with a contraction modulus α that depends on $\overline{S}$. In particular, α is given by

$$\alpha = \frac{\partial F(K)}{\partial K}\Big|_{K=\overline{S}}$$

The study goes on to assert the importance of long multistep lookahead (the size of ℓ) in stabilizing the performance of approximation in value space schemes. It also confirms computationally a known theoretical result, namely that with long enough lookahead ℓ, the ℓ-step lookahead policy $\tilde{\mu}$ is exactly optimal (but the required length of ℓ depends on the approximation error $\tilde{J} - J^*$). This fact has been known since the 60s-70s for α-discounted finite-state problems. A generalization of this result is given as Prop. 2.3.1 of the abstract DP book [Ber22b]; see also Section A.4 of the book [Ber22a], which discusses the convergence of Newton's method for systems of equations that involve nondifferentiable mappings (such as the Bellman operator).

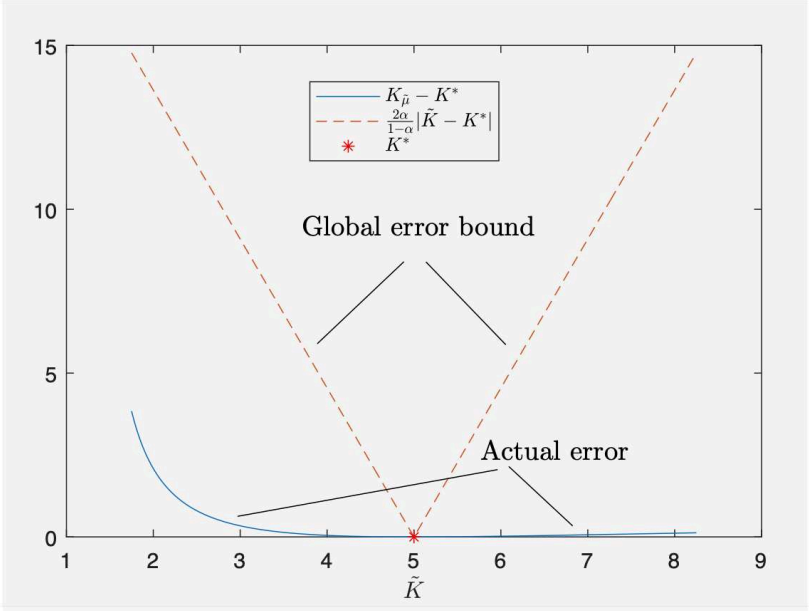

Figure 1.5.14 Illustration of the global error bound for the one-step lookahead error $K_{\tilde{\mu}} - K^*$ as a function of $\tilde{K}$, compared with the true error obtained by one step of Newton's method starting from $\tilde{K}$; cf. Example 1.5.1.

The problem data are $a = 2$, $b = 2$, $q = 1$, and $r = 5$. With these numerical values, we have $K^* = 5$ and the region of stability is (S, ∞) with $S = 1.25$. The modulus of contraction α used in the figure is computed at $\overline{S} = S + 0.5$. Depending on the chosen value of $\overline{S}$, α can be arbitrarily close to 1, but decreases as $\overline{S}$ increases. Note that the error $K_{\tilde{\mu}} - K^*$ is much smaller when $\tilde{K}$ is larger than K^* than when it is lower, because the slope of F diminishes as K increases. This is not reflected by the global error bound.

and satisfies $0 < \alpha < 1$ because $\overline{S} > S$, and the derivative of F is positive and monotonically decreasing to 0 as K increases to ∞.

The error bound (1.57) can be rederived for the case of quadratic functions and can be rewritten in terms of quadratic cost coefficients as

$$K_{\tilde{\mu}} - K^* \leq \frac{2\alpha}{1-\alpha}|\tilde{K} - K^*|, \tag{1.60}$$

where $K_{\tilde{\mu}}$ is the quadratic cost coefficient of the lookahead policy $\tilde{\mu}$ [and also the result of a Newton step for solving the fixed point Riccati equation $F = F(K)$ starting from $\tilde{K}$]. A plot of $(K_{\tilde{\mu}} - K^*)$ as a function of $\tilde{K}$, compared with the bound on the right side of this equation is shown in Fig. 1.5.14. It can be seen that $(K_{\tilde{\mu}} - K^*)$ exhibits the qualitative behavior of Newton's method, which is very different than the bound (1.60). An interesting fact is that the bound (1.60) depends on α, which in turn depends on how close $\tilde{K}$ is to the boundary S of the region of stability, while the local behavior of Newton's method is independent of S.

How Approximation in Value Space Can Fail and What to Do About It

Let us finally discuss the most common way that approximation in value space can fail. Consider the case where the terminal cost approximation $\tilde{J}$ is obtained through training with data of an approximation architecture such as a neural network (e.g., as in AlphaZero and TD-Gammon). Then there are three components that determine the approximation error $\tilde{J} - J^*$:

(a) The *power of the architecture*, which roughly speaking is a measure of the error that would be obtained if infinite data were available and were used optimally to obtain $\tilde{J}$.

(b) The *error degradation due the limited availability of training data.*

(c) The additional *error degradation due to imperfections in the training methodology.*

Thus *if the architecture is not powerful enough to bring $\tilde{J} - J^*$ within the region of convergence of Newton's method, approximation in value space with one-step lookahead will likely fail, no matter how much data is collected and how effective the associated training method is.*

In this case, there are two potential practical remedies:

(1) Use a more powerful architecture/neural network for representing $\tilde{J}$.

(2) Extend the combined length of the lookahead minimization and truncated rollout in order to bring the effective value of $\tilde{J}$ within the region of convergence of Newton's method.

The first remedy typically requires a deep neural network or transformer, which uses more weights and requires more expensive training (see Chapter 3).† The second remedy requires longer on-line computation and/or simulation, which may run up against some practical real-time implementation constraint. Parallel computation and sophisticated multistep lookahead implementation methods may help to mitigate these requirements (see Chapter 2).

1.6 EXAMPLES, REFORMULATIONS, AND SIMPLIFICATIONS

In this section we provide a few examples that illustrate problem formulation techniques, exact and approximate solution methods, and adaptations of the basic DP algorithm to various contexts. We refer to DP textbooks for extensive additional discussions of modeling and problem formulation techniques (see e.g., the many examples that can be found in the author's

† For a recent example of implementation of a grandmaster-level chess program with *one-step lookahead* and a huge-size (270M parameter) neural network position evaluator, see Ruoss et al. [RDM24].

DP and RL textbooks [Ber12], [Ber17a], [Ber19a], [Ber20a], as well as in the neuro-dynamic programming book [BeT96]).

An important fact to keep in mind is that there are many ways to model a given practical problem in terms of DP, and that there is no unique choice for state and control variables. This will be brought out by the examples in this section, and is facilitated by the generality of DP: its basic algorithmic principles apply for arbitrary state, control, and disturbance spaces, and system and cost functions.

1.6.1 A Few Words About Modeling

In practice, optimization problems seldom come neatly packaged as mathematical problems that can be solved by DP/RL or some other methodology. Generally, a practical problem is a prime candidate for a DP formulation if it involves multiple sequential decisions, which are separated by feedback, i.e., by observations that can be used to enhance the effectiveness of future decisions.

However, there are other types of problems that can be fruitfully formulated by DP. These include the entire class of deterministic problems, where there is no information to be collected: all the information needed in a deterministic problem is either known or can be predicted from the problem data that is available at time 0 (see, e.g., the traveling salesman Example 1.2.3). Moreover, for deterministic problems there is a plethora of non-DP methods, such as linear, nonlinear, and integer programming, random and nonrandom search, discrete optimization heuristics, etc. Still, however, the use of RL methods for deterministic optimization is a major subject in this book, which will be discussed in Chapter 2. We will argue there that rollout and its variations, when suitably applied, can improve substantially on the performance of other heuristic or suboptimal methods, however derived. Moreover, we will see that often for discrete optimization problems the DP sequential structure is introduced artificially, with the aim to facilitate the use of approximate DP/RL methods.

There are also problems that fit quite well into the sequential structure of DP, but can be fruitfully addressed by RL methods that do not have a fundamental connection with DP. An important case in point is *policy gradient* and *policy search* methods, which will not be considered in this book. Here the policy of the problem is parametrized by a set of parameters, so that the cost of the policy becomes a function of these parameters, and can be optimized by non-DP methods such as gradient or random search-based suboptimal approaches. This generally relates to the approximation in policy space approach, which we have discussed in Section 1.3.3 and we will discuss further in Section 3.4; see also Section 5.7 of the RL book [Ber19a].

As a guide for formulating optimal control problems in a manner that is suitable for a 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 relevant for future optimization*, i.e., 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 an analogy 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.

The control and state selection may also have to be refined or specialized in order to enhance the application of known results and algorithms. This includes the choice of a finite or an infinite horizon, and the availability of good base policies or heuristics in the context of rollout.

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 may thus be 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 that would have to be executed 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 providing alternative descriptions of the state space. An important paradigm is problems involving *partial* or *imperfect* state information, where x_k evolves over time but is not fully accessible for measurement (for example, x_k may be the position/velocity vector of a moving vehicle, but we may obtain measurements of just the position). If I_k is the collection of all measurements and controls up to time k (the

information vector), it is correct to use I_k as state in a reformulated DP problem that involves perfect state observation. However, a better alternative may be to use as state the conditional probability distribution

$$P_k(x_k \mid I_k),$$

called *belief state*, which (as it turns out) subsumes all the information that is useful for the purposes of choosing a control. On the other hand, the belief state $P_k(x_k \mid I_k)$ is an infinite-dimensional object, whereas I_k may be finite dimensional, so the best choice may be problem-dependent. Still, in either case, the stochastic DP algorithm applies, with the sufficient statistic [whether I_k or $P_k(x_k \mid I_k)$] playing the role of the state.

A Few Words about the Choice of an RL Method

An attractive aspect of the current RL methodology, inherited by the generality of our DP formulation, is that it can address a very broad range of challenging problems, deterministic as well as stochastic, discrete as well as continuous, etc. However, in the practical application of RL methods one has to contend with limited theoretical guarantees. In particular, several of the RL methods that have been successful in practice have less than solid performance properties, and may not work on a given problem, even one of the type for which they are designed.

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 DP problems*. However, there are enough methods to try on a given problem with a reasonable chance of success in the end (after some heuristic and problem specific tuning). For this reason, it is important to develop insight into the inner workings of various methods, as a means of selecting the proper type of methodology to try on a given problem.†

A related consideration is the context within which a method is applied. In particular, is it a single problem that is being addressed, such as chess that has fixed rules and a fixed initial condition, or is it a family of related problems that must be periodically be solved with small variations in its data or its initial conditions? Also, are the problem data fixed or may they change over time as the system is being controlled?

Generally, convenient but relatively unreliable methods, which can be tuned to the problem at hand, may be tried with a reasonable chance of success if a single problem is addressed. Similarly, RL methods that require

† Aside from insight and intuition, it is also important to have a foundational understanding of the analytical principles of the field and of the mechanisms underlying the central computational methods. The role of the theory in this respect is to structure mathematically the methodology, guide the art, and delineate the sound from the flawed ideas.

extensive tuning of parameters, including ones that involve approximation in policy space and the use of neural networks, may be well suited for a stable problem environment and a single problem solution. However, they not well suited for problems with a variable environment and/or real-time changes of model parameters. For such problems, RL methods based on approximation in value space and on-line play, possibly involving on-line replanning, are much better suited.

Note also that even when on-line replanning is not needed, on-line play may improve substantially the performance of off-line trained policies, so we may wish to use it in conjunction with off-line training. This is due to the Newton step that is implicit in one-step or multistep lookahead minimization, cf. our discussion of the AlphaZero and TD-Gammon architectures in Section 1.1. Of course the computational requirements of an on-line play method may be substantial and have to be taken into account when assessing its suitability for a particular application. In this connection, deterministic problems are better suited than stochastic problems for on-line play. Moreover, methods that are well-suited for parallel computation, and/or involve the use of certainty equivalence approximations are generally better suited for a stochastic control environment.

1.6.2 Problems with a Termination State

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

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

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 random transition from another state. Problems involving games, such as chess, Go, backgammon, and others involve a termination state (the end of the game) and have played an important role in the development of the RL methodology.†

Generally, when it is known that an optimal policy will reach the termination state with certainty within at most some given number of stages N, the DP problem can be formulated as an N-stage horizon problem, with a very large termination cost for the nontermination states.‡ The reason

† Games often involve two players/decision makers, in which case they can be addressed by suitably modified exact or approximate DP algorithms. The DP algorithm that we have discussed in this chapter involves a single decision maker, but can be used to find an optimal policy for one player against a fixed and known policy of the other player.

‡ When an upper bound on the number of stages to termination is not known, the problem may be formulated as an infinite horizon problem of the stochastic shortest path problem.

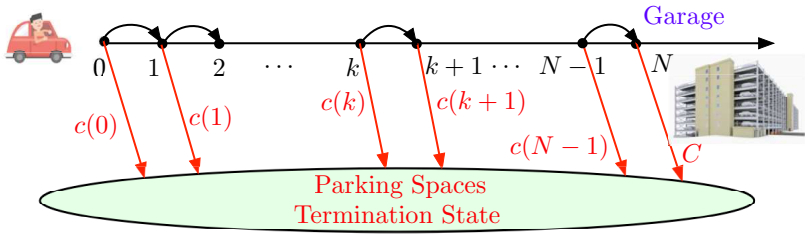

Figure 1.6.1 Cost structure of the parking problem. The driver may park at space $k = 0, 1, \ldots, N-1$ at cost $c(k)$, if the space is free, or continue to the next space $k+1$ at no cost. At space N (the garage) the driver must park at cost C.

is that even if the termination 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, so the optimal policy will still be optimal, since it will not incur the large termination cost at the end of the horizon.

Example 1.6.1 (Parking)

A driver is looking for inexpensive parking on the way to his destination. The parking area contains N spaces, numbered $0, \ldots, N-1$, and a garage following space $N-1$. 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 $N-1$ 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 termination state t that corresponds to having parked; see Fig. 1.6.1. At each stage $k = 1, \ldots, N-1$, we have three states: the artificial termination state t, and the two states F and $\overline{F}$, corresponding to space k being free or taken, respectively. At stage 0, we have only two states, F and $\overline{F}$, and at the final stage there is only one state, the termination state t. The decision/control is to park or continue at state F [there is no choice at states $\overline{F}$ and state t]. From location k, the termination state t is reached at cost $c(k)$ when a parking decision is made (assuming location k is free). Otherwise, the driver continues to the next state at no cost. At stage N, the driver must park at cost C.

Let us now derive the form of the 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.

$J_k^*(t)$: The cost-to-go of the "parked"/termination state t.

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

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

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

for the states other than the termination state t, while for t we have

$$J_k^*(t) = 0, \qquad k = 1, \ldots, N.$$

The minimization above corresponds to the two choices (park or not park) at the states F that correspond to a free parking space.

While this algorithm is easily executed, it can be written in a simpler and equivalent form. 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:

Park at space $k = 0, \ldots, N-1$ if it is free and we have $c(k) \leq \hat{J}_{k+1}$.

This is an example of DP simplification that occurs when the state involves components that are not affected by the choice of control, and will be addressed in the next section.

1.6.3 General Discrete Optimization Problems

Discrete deterministic optimization problems, including challenging combinatorial problems, can be typically formulated as DP problems by breaking down each feasible solution into a sequence of decisions/controls, similar to the preceding four queens example, the scheduling Example 1.2.1, and the traveling salesman Examples 1.2.2 and 1.2.3. This formulation often leads to an intractable exact DP computation because of an exponential explosion of the number of states as time progresses. However, a reformulation to a discrete optimal control problem brings to bear approximate DP methods, such as rollout and others, to be discussed shortly, which can deal with the exponentially increasing size of the state space.

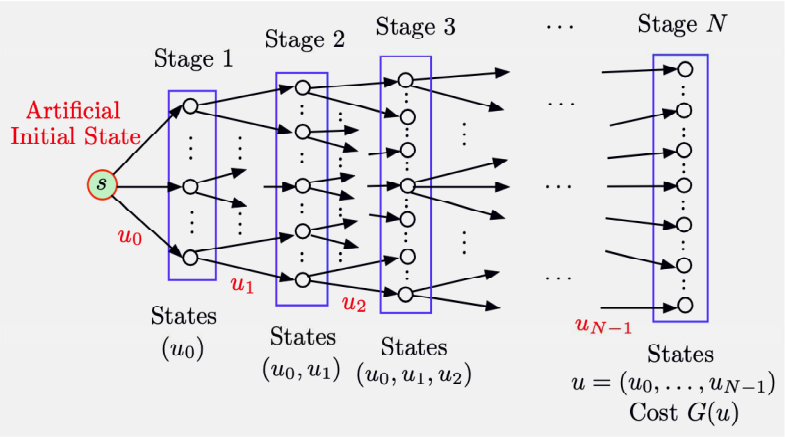

Figure 1.6.2 Formulation of a discrete optimization problem as a DP problem with N stages. There is a cost $G(u)$ only at the terminal stage on the arc connecting an N-solution $u = (u_0, \ldots, u_{N-1})$ upon reaching the terminal state. Note that there is only one incoming arc at each node.

Let us now extend the ideas of the examples just noted to the general discrete optimization problem:

$$\begin{aligned} &\text{minimize} \quad G(u) \\ &\text{subject to} \quad u \in U, \end{aligned} \tag{1.61}$$

where U is a finite set of feasible solutions and $G(u)$ is a cost function.

We assume that each solution u has N components; i.e., it has the form

$$u = (u_0, \ldots, u_{N-1}),$$

where N is a positive integer. We can then view the problem as a sequential decision problem, where the components $u_0, \ldots, u_{N-1}$ are selected one-at-a-time. A k-tuple $(u_0, \ldots, u_{k-1})$ consisting of the first k components of a solution is called a *k-solution*. We associate k-solutions with the kth stage of the finite horizon discrete optimal control problem shown in Fig. 1.6.2. In particular, for $k = 1, \ldots, N$, we view as the states of the kth stage all the k-tuples $(u_0, \ldots, u_{k-1})$. For stage $k = 0, \ldots, N-1$, we view u_k as the control. The initial state is an artificial state denoted s. From this state, by applying u_0, we may move to any "state" (u_0), with u_0 belonging to the set

$$U_0 = \big\{\tilde{u}_0 \mid \text{there exists a solution of the form } (\tilde{u}_0, \tilde{u}_1, \ldots, \tilde{u}_{N-1}) \in U\big\}. \tag{1.62}$$

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

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

the set

$$U_k(u_0,\ldots,u_{k-1}) = \big\{u_k \mid \text{for some } \overline{u}_{k+1},\ldots,\overline{u}_{N-1} \text{ we have } (u_0,\ldots,u_{k-1},u_k,\overline{u}_{k+1},\ldots,\overline{u}_{N-1}) \in U\big\}. \tag{1.63}$$

These are the choices of u_k that are consistent with the preceding choices $u_0,\ldots,u_{k-1}$, and are also consistent with feasibility [we do not exclude the possibility that the set (1.63) is empty]. The last stage corresponds to the N-solutions $u = (u_0,\ldots,u_{N-1})$, and the terminal cost is $G(u)$; see Fig. 1.6.2. All other transitions in this DP problem formulation have cost 0.

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

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

with the terminal condition

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

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

$$u_k^* \in \arg\min_{u_k\in U_k(u_0^*,\ldots,u_{k-1}^*)} J_{k+1}^*(u_0^*,\ldots,u_{k-1}^*,u_k), \quad k = 0,\ldots,N-1, \tag{1.64}$$

where U_0 is given by Eq. (1.62), and U_k is given by Eq. (1.63): first compute u_0^*, then u_1^*, and so on up to u_{N-1}^*; cf. Eq. (1.8).

Of course here the number of states typically grows exponentially with N, but we can use the DP minimization (1.64) as a starting point for 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. (1.64). One possibility is to use as

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

the cost generated by a heuristic method that solves the problem suboptimally with the values of the first $k+1$ decision components fixed at $u_0^*,\ldots,u_{k-1}^*,u_k$. This is the *rollout algorithm*, which turns out to be a very simple and effective approach for approximate combinatorial optimization.

Let us finally note that while we have used a general cost function G and constraint set U in our discrete optimization model of this section, in

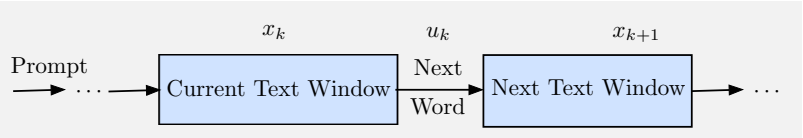

Figure 1.6.3 Schematic visualization of an LLM problem based on N-grams.

many problems G and/or U may have a special (e.g., additive) structure, which is consistent with a sequential decision making process and may be computationally exploited. The traveling salesman Example 1.2.2 is a case in point, where G consists of the sum of N components (the intercity travel costs), one per stage. Our next example deals with a problem of great current interest.

Example 1.6.2 (A Large Language Model Based on N-Grams)

Let us consider an N-gram model, whereby a text string consisting of N words is transformed into another string of N words by adding a word at the front of the string and deleting the word at the back of the string. We view the text strings as states of a dynamic system affected by the added word choice, which we view as the control. We denote by x_k the string obtained at time k, and by u_k the word added at time k. We assume that u_k is chosen from a given set $U(x_k)$. Thus we have a controlled dynamic system, which is deterministic and is described by an equation of the form

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

where f specifies the operation of adding u_k at the front of x_k and removing the word at the back of x_k. The initial string x_0 is assumed given.

If we have a cost function G by which to evaluate a text string, we can pose a DP problem with either a finite or an infinite horizon. For example if the string evolution terminates after exactly N steps, we obtain the finite horizon problem of minimizing the function $G(x_N)$ of the final text string x_N. In this case, x_N is obtained after we have a chance to change successively all the words of the initial string x_0, subject to the constraints $u_k \in U(x_k)$.

Another possibility is to introduce a termination action, whereby addition/deletion of words is optionally stopped at some time and the final text string x is obtained with cost $G(x)$. In such a problem formulation, we may also include an additive stage cost that depends on u. This is an infinite horizon formulation that involves an additional termination state t in the manner of Section 1.6.2.

Note that in both the finite and the infinite horizon formulation of the problem, the initial string x_0 may include a "prompt," which may be subject to optimization through some kind of "prompt engineering." Depending on the context, this may include the use of another optimization or heuristic algorithm, perhaps unrelated to DP, which searches for a favorable prompt from within a given set of choices.

Interesting policies for the preceding problem formulation may be provided by a neural network, such as a Generative Pretrained Transformer

(GPT). In our terms, the GPT can be viewed simply as a policy that generates next words. This policy may be either deterministic, i.e., $u_k = \mu(x_k)$ for some function μ, or it may be a "randomized" policy, which generates u_k according to a probability distribution that depends on x_k. Our DP formulation can also form the basis for policy improvement algorithms such as rollout, which aim to improve the quality of the output generated by the GPT. Another, more ambitious, possibility is to consider an approximate, neural network-based, policy iteration/self-training scheme, such as the ones discussed earlier, based on the AlphaZero/TD-Gammon architecture. Such a scheme generates a sequence of GPTs, with each GPT trained with data provided by the preceding GPT, a form of self-learning in the spirit of the AlphaZero and TD-Gammon policy iteration algorithms, cf. Section 1.1.

It is also possible to provide an infinite horizon formulation of the general disrete optimization problem

$$\begin{aligned} &\text{minimize} \quad G(u) \\ &\text{subject to} \quad u \in U, \end{aligned} \tag{1.65}$$

where U is a finite set of feasible solutions, $G(u)$ is a cost function, and u consists of N components, $u = (u_0, \ldots, u_{N-1})$; cf. Eq. (1.61). To this end, we introduce a termination state t that the system enters after N steps. At step k, the component u_k is selected subject to $u_k \in U_k(u_0, \ldots, u_{k-1}$, where the constraint set $U_k(u_0, \ldots, u_{k-1}$ is given by Eq. (1.63). This is a special case of a general finite to infinite horizon stochastic DP problem reformulation, which we describe in the next section.

1.6.4 General Finite to Infinite Horizon Reformulation

There is a conceptually important reformulation that transforms a finite horizon problem, possibly involving a nonstationary system and cost per stage, to an equivalent infinite horizon problem. It is based on introducing an expanded state space, which is the union of the state spaces of the finite horizon problem plus an artificial cost-free termination state that the system moves into at the end of the horizon. This reformulation is of great conceptual value, as it provides a mechanism to bring to bear ideas that can be most conveniently understood within an infinite horizon context. For example, it helps to understand the synergy of off-line training and on-line play based on Newton's method, and the related insights that explain the good performance of rollout algorithms in practice.

To define the reformulation, let us consider the N-stage horizon stochastic problem of Section 1.3.1, whose system has the form

$$x_{k+1} = f_k(x_k, u_k, w_k), \qquad k = 0, \ldots, N-1, \tag{1.66}$$

and let us denote by X_k, $k = 0, \ldots, N$, and U_k, $k = 0, \ldots, N-1$, the corresponding state spaces and control spaces, respectively. We introduce

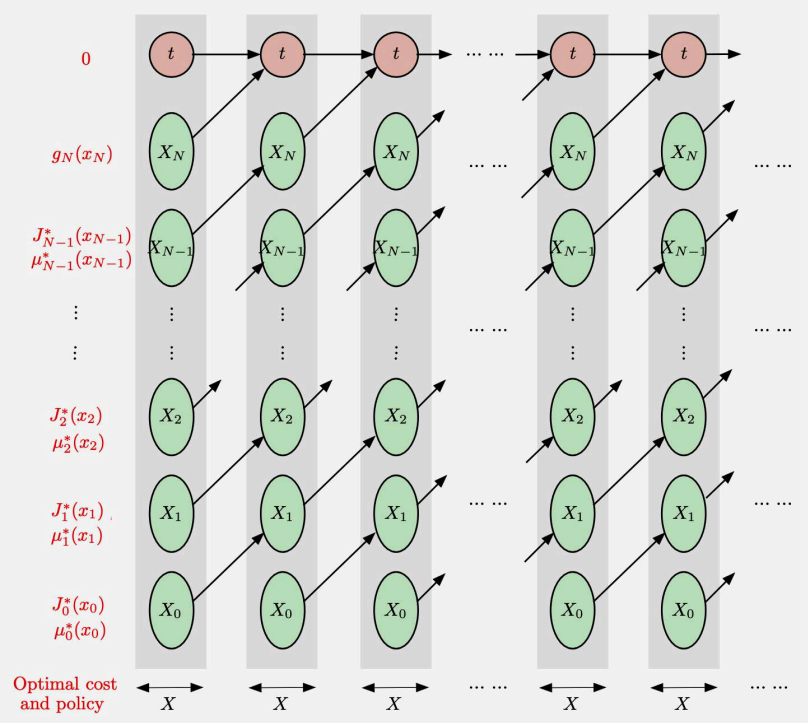

Figure 1.6.4 Illustration of the infinite horizon equivalent of a finite horizon problem. The state space is $X = \left(\cup_{k=0}^{N} X_k\right) \cup \{t\}$, and the control space is $U = \cup_{k=0}^{N-1} U_k$. Transitions from states $x_k \in X_k$ lead to states in $x_{k+1} \in X_{k+1}$ according to the system equation $x_{k+1} = f_k(x_k, u_k, w_k)$, and they are stochastic when they involve the random disturbance w_k. The transition from states $x_N \in X_N$ lead deterministically to the termination state at cost $g_N(x_N)$. The termination state t is cost-free and absorbing.

The infinite horizon optimal cost $J^*(x_k)$ and optimal policy $\mu^*(x_k)$ at state $x_k \in X_k$ of the infinite horizon problem are equal to optimal cost-to-go $J^*_k(x_k)$ and optimal policy $\mu^*_k(x_k)$ of the finite horizon problem.

an artificial termination state t, and we consider an infinite horizon problem with state and control spaces X and U given by

$$X = \left(\cup_{k=0}^{N} X_k\right) \cup \{t\}, \qquad U = \cup_{k=0}^{N-1} U_k; \tag{1.67}$$

see Fig. 1.6.4.

The system equation and the control constraints of this problem are also reformulated so that states in X_k, $k = 0, \ldots, N-1$, are mapped to states in X_{k+1}, according to Eq. (1.66), while states $x_N \in X_N$ are mapped to the termination state t at cost $g_N(x_N)$. Upon reaching t, the state stays at t at no cost. Thus the policies of the infinite horizon problem map states $x_k \in X_k$ to controls in $U_k(x_k) \subset U_k$, and consist of functions $\mu_k(x_k)$ that

are policies of the finite horizon problem. Moreover, the Bellman equation for the infinite horizon problem is identical to the DP algorithm for the finite horizon problem.

It can be seen that the optimal cost and optimal control, $J^*(x_k)$ and $\mu^*(x_k)$, at a state $x_k \in X_k$ in the infinite horizon problem are equal to the optimal cost-to-go $J_k^*(x_k)$ and optimal control $\mu_k^*(x_k)$ of the original finite horizon problem, respectively; cf. Fig. 1.6.4. Moreover approximation in value space and rollout in the finite horizon problem translate to infinite horizon counterparts, and can be understood as Newton steps for solving the Bellman equation of the infinite horizon problem (or equivalently the DP algorithm of the finite horizon problem).

In summary, finite horizon problems can be viewed as infinite horizon problems with a special structure that involves a termination state t, and the state and control spaces of Eq. (1.67), as illustrated in Fig. 1.6.4. The Bellman equation of the infinite horizon problem coincides with the DP algorithm of the finite horizon problem. The PI algorithm for the infinite horizon problem can be translated directly to a PI algorithm for the finite horizon problem, involving repeated policy evaluations and policy improvements. Finally, the Newton step interpretations for approximation in value space and rollout schemes for the infinite horizon problem have straightforward analogs for finite horizon problems, and explain the powerful cost improvement mechanism that underlies the rollout algorithm and its variations.

1.6.5 State Augmentation, Time Delays, Forecasts, and Uncontrollable State Components

In practice, we are often faced with situations where some of the assumptions of our stochastic optimal control problem formulation are violated. For example, the disturbances may involve a complex probabilistic description that may create correlations that extend across stages, or the system equation may include dependences on controls applied in earlier stages, which affect the state with some delay.

Generally, in such cases the problem can be reformulated into our DP problem format through a technique, which is called *state augmentation* because it typically involves the enlargement of the state space. The general intuitive guideline in state augmentation is to *include in the enlarged state at time k all the information that is known to the controller at time k and can be used with advantage in selecting u_k*. State augmentation allows the treatment of time delays in the effects of control on future states, correlated disturbances, forecasts of probability distributions of future disturbances, and many other complications. We note, however, that state augmentation often comes at a price: the reformulated problem may have a very complex state space. We provide some examples.

Time Delays

In some applications the system state x_{k+1} depends not only on the preceding state x_k and control u_k, but also on earlier states and controls. Such situations can be handled by expanding the state to include an appropriate number of earlier states and controls.

As an example, assume that there is at most a single stage delay in the state and control; i.e., the system equation has the form

$$x_{k+1} = f_k(x_k, x_{k-1}, u_k, u_{k-1}, w_k), \qquad k = 1, \ldots, N-1, \tag{1.68}$$

$$x_1 = f_0(x_0, u_0, w_0).$$

If we introduce additional state variables y_k and s_k, and we make the identifications $y_k = x_{k-1}$, $z_k = u_{k-1}$, the system equation (1.68) yields

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \\ z_{k+1} \end{pmatrix} = \begin{pmatrix} f_k(x_k, y_k, u_k, z_k, w_k) \\ x_k \\ u_k \end{pmatrix}. \tag{1.69}$$

By defining $\tilde{x}_k = (x_k, y_k, z_k)$ as the new state, we have

$$\tilde{x}_{k+1} = \tilde{f}_k(\tilde{x}_k, u_k, w_k),$$

where the system function $\tilde{f}_k$ is defined from Eq. (1.69).

By using the preceding equation as the system equation and by expressing the cost function in terms of the new state, the problem is reduced to a problem without time delays. Naturally, the control u_k should now depend on the new state $\tilde{x}_k$, or equivalently a policy should consist of functions μ_k of the current state x_k, as well as the preceding state x_{k-1} and the preceding control u_{k-1}.

When the DP algorithm for the reformulated problem is translated in terms of the variables of the original problem, it takes the form

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

$$\begin{aligned} &J_{N-1}^*(x_{N-1}, x_{N-2}, u_{N-2}) \\ &\qquad = \min_{u_{N-1} \in U_{N-1}(x_{N-1})} E_{w_{N-1}} \Big\{ g_{N-1}(x_{N-1}, u_{N-1}, w_{N-1}) \\ &\qquad\qquad + J_N^*\big(f_{N-1}(x_{N-1}, x_{N-2}, u_{N-1}, u_{N-2}, w_{N-1})\big) \Big\}, \end{aligned}$$

$$\begin{aligned} J_k^*(x_k, x_{k-1}, u_{k-1}) = \min_{u_k \in U_k(x_k)} E_{w_k} \Big\{ & g_k(x_k, u_k, w_k) \\ & + J_{k+1}^*\big(f_k(x_k, x_{k-1}, u_k, u_{k-1}, w_k), x_k, u_k\big) \Big\}, \quad k = 1, \ldots, N-2, \end{aligned}$$

$$J_0^*(x_0) = \min_{u_0 \in U_0(x_0)} E_{w_0}\Big\{g_0(x_0, u_0, w_0) + J_1^*\big(f_0(x_0, u_0, w_0), x_0, u_0\big)\Big\}.$$

Similar reformulations are possible when time delays appear in the cost or the control constraints; for example, in the case where the cost is

$$E\left\{g_N(x_N, x_{N-1}) + g_0(x_0, u_0, w_0) + \sum_{k=1}^{N-1} g_k(x_k, x_{k-1}, u_k, w_k)\right\}.$$

The extreme case of time delays in the cost arises in the nonadditive form

$$E\big\{g_N(x_N, x_{N-1}, \ldots, x_0, u_{N-1}, \ldots, u_0, w_{N-1}, \ldots, w_0)\big\}.$$

Then, the problem can be reduced to the standard problem format, by using as augmented state

$$\tilde{x}_k = \big(x_k, x_{k-1}, \ldots, x_0, u_{k-1}, \ldots, u_0, w_{k-1}, \ldots, w_0\big)$$

and $E\big\{g_N(\tilde{x}_N)\big\}$ as reformulated cost. Policies consist of functions μ_k of the present and past states $x_k, \ldots, x_0$, the past controls $u_{k-1}, \ldots, u_0$, and the past disturbances $w_{k-1}, \ldots, w_0$. Naturally, we must assume that the past disturbances are known to the controller. Otherwise, we are faced with a problem where the state is imprecisely known to the controller, which will be discussed in the next section.

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 the subsequent disturbance w_k and, possibly, 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 state variables 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 and disturbance are

$$\tilde{x}_k = (x_k, y_k), \qquad \tilde{w}_k = (w_k, \xi_k).$$

The probability distribution of $\tilde{w}_k$ 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

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

$$J_k^*(x_k, y_k) = \min_{u_k \in U_k(x_k)} E_{w_k} \Big\{ g_k(x_k, u_k, w_k) + \sum_{i=1}^{m} p_i J_{k+1}^*\big(f_k(x_k, u_k, w_k), i\big) \mid y_k \Big\}, \tag{1.70}$$

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}.

Note 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 may involve several future disturbances. However, the price for these extensions is increased complexity of the corresponding DP algorithm.

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.

As an 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

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

where the 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.

It turns out that we can formulate a DP algorithm that is executed over the controllable component of the state, with the dependence on the uncontrollable component being "averaged out" (see also the parking Example 1.6.1). 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

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

Note that the preceding expression can be interpreted as an "average cost-to-go" at x_k (averaged over the values of the uncontrollable component y_k). Then, similar to the parking Example 1.6.1, a DP algorithm that generates $\hat{J}_k(x_k)$ can be obtained, and has the following form:

$$\hat{J}_k(x_k) = E_{y_k}\Bigg\{\min_{u_k \in U_k(x_k, y_k)} E_{w_k}\Big\{g_k(x_k, y_k, u_k, w_k) + \hat{J}_{k+1}\big(f_k(x_k, y_k, u_k, w_k)\big) \;\Big|\; x_k, y_k, u_k\Big\} \;\Big|\; x_k\Bigg\}. \tag{1.71}$$

This is a consequence of the calculation

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

Note that the minimization in the right-hand side of the preceding equation must still be performed for all values of the full state (x_k, y_k) in order to yield an optimal control law as a function of (x_k, y_k). Nonetheless, the equivalent DP algorithm (1.71) has the advantage that it is executed over a significantly reduced state space. Later, when we consider approximation in value space, we will find that it is often more convenient to approximate $\hat{J}_k(x_k)$ than to approximate $J_k^*(x_k, y_k)$; see the following discussions of forecasts and of the game of tetris.

As an example, consider the augmented state resulting from the incorporation of forecasts, as described earlier. Then, the forecast y_k represents an uncontrolled state component, so that the DP algorithm can be simplified as in Eq. (1.71). In particular, assume that the forecast y_k can take values $i = 1, \ldots, m$ with probability p_i. Then, 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. (1.70),

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

which is executed over the space of x_k rather than x_k *and* y_k. Note that this is a simpler algorithm to approximate than the one of Eq. (1.70).

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. (1.71). Here is an example of this type.

Example 1.6.3 (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. 1.6.5). The squares fill up as blocks of different 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) up to termination of the game, whichever occurs first.

We can model the problem of finding an optimal tetris playing strategy as a finite horizon stochastic DP problem, with very long horizon. 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.

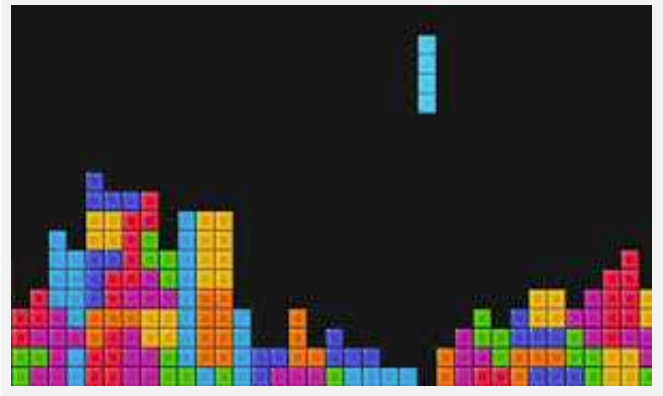

Figure 1.6.5 Illustration of a tetris board.

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

The control, denoted by u, is the horizontal positioning and rotation applied to the falling block. 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 score. Moreover there is a very large amount added to the score when the end of the horizon is reached without the game having terminated.

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 (1.71) is executed over the space of board positions x and has the intuitive form

$$\hat{J}_k(x) = \sum_y p(y) \max_u \Big[g(x,y,u) + \hat{J}_{k+1}\big(f(x,y,u)\big) \Big], \qquad \text{for all } x, \qquad (1.72)$$

where

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

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

when the state is (x,y) and control u is applied, respectively. The DP algorithm (1.72) assumes a finite horizon formulation of the problem.

Alternatively, we may consider an undiscounted infinite horizon formulation, involving a termination state (i.e., a stochastic shortest path problem). The "reduced" form of Bellman's equation, which corresponds to the DP algorithm (1.72), has the form

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

The value $\hat{J}(x)$ can be interpreted as an "average score" at x (averaged over the values of the uncontrollable block shapes y).

Finally, let us note that despite the simplification achieved by eliminating the uncontrollable portion of the state, the number of states x is still enormous, and the problem can only be addressed by suboptimal methods.†

1.6.6 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 of x_k. However, in many practical settings, this assumption is unrealistic because some components of the state may be inaccessible for observation, the sensors used for measuring them may be inaccurate, or the cost of measuring them more accurately may be prohibitive.

Often in such situations, the controller has access to only some of the components of the current state, and the corresponding observations 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 observations 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., [Ber17a], [RuN16]; these problems are also popularly referred to with the acronym POMDP for *partially observed Markovian Decision problem*).

Generally, solving a POMDP exactly is typically intractable, even though there are DP algorithms for doing so. Thus in practice, POMDP are solved approximately, except under very special circumstances.

Despite their inherent computational difficulty, 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, which involves a different and more complicated state, called a *sufficient statistic*. Once this is done, we can state an exact DP algorithm that is defined over the space of the sufficient statistic. Roughly speaking, a sufficient statistic is a quantity that summarizes the content of the information available up to k for the purposes of optimal control. This statement can be made more precise, but we will not elaborate further in this book; see e.g., the DP textbook [Ber17a].

† Tetris is generally considered to be an interesting and challenging stochastic testbed for RL algorithms, and has received a lot of attention over a period spanning 20 years (1995-2015), starting with the papers [TsV96], [BeI96], and the neuro-dynamic programming book [BeT96], and ending with the papers [GGS13], [SGG15], which contain many references to related works in the intervening years. All of these works are based on approximation in value space and various forms of approximate policy iteration.

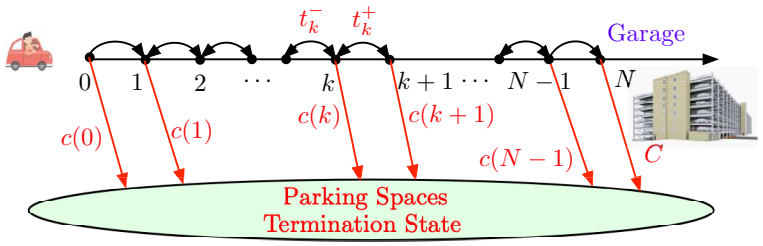

Figure 1.6.6 Cost structure and transitions of the bidirectional parking problem. The driver may park at space $k = 0, 1, \ldots, N-1$ at cost $c(k)$, if the space is free, can move to $k-1$ at cost t_k^- or can move to $k+1$ at cost t_k^+. At space N (the garage) the driver must park at cost C.

A common sufficient statistic is the *belief state*, which we will denote by b_k. It is the probability distribution of x_k given all the observations that have been obtained by the controller and all the controls applied by the controller up to time k, and it can serve as "state" in an appropriate DP algorithm. The belief state can in principle be computed and updated by a variety of methods that are based on Bayes' rule, such as *Kalman filtering* (see e.g., [AnM79], [KuV86], [Kri16], [ChC17]) and *particle filtering* (see e.g., [GSS93], [DoJ09], [Can16], [Kri16]).

Example 1.6.4 (Bidirectional Parking)

Let us consider a more complex version of the parking problem of Example 1.6.1. 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 t_i^- or can move to $i+1$ at a cost t_i^+. Moreover, the driver records and remembers the free/taken status of the spaces previously visited and may return to any of these spaces; see Fig. 1.6.6.

We 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 stage, $p(i)$ 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

$$\big(p(0), \ldots, p(N-1)\big),$$

and it can be updated with a simple algorithm at each time based on the new observation: the free/taken status of the space visited at that time.

We can use the belief state as the basis of an exact DP algorithm for computing an optimal policy. This algorithm is typically intractable computationally, but it is conceptually useful, and it can form the starting point for approximations. It has the form

$$J_k^*(b_k) = \min_{u_k \in U_k} \left[\hat{g}_k(b_k, u_k) + E_{z_{k+1}} \Big\{ J_{k+1}^*\big(F_k(b_k, u_k, z_{k+1})\big) \;\Big|\; b_k, u_k \Big\} \right], \tag{1.73}$$

where:

$J_k^*(b_k)$ denotes the optimal cost-to-go starting from belief state b_k at stage k.

U_k is the control constraint set at time k (since the state x_k is unknown at stage k, U_k must be independent of x_k).

$\hat{g}_k(b_k, u_k)$ denotes the expected stage cost of stage k. It is calculated as the expected value of the stage cost $g_k(x_k, u_k, w_k)$, with the joint distribution of (x_k, w_k) determined by the belief state b_k and the distribution of w_k.

$F_k(b_k, u_k, z_{k+1})$ denotes the belief state at the next stage, given that the current belief state is b_k, control u_k is applied, and observation z_{k+1} is received following the application of u_k:

$$b_{k+1} = F_k(b_k, u_k, z_{k+1}). \tag{1.74}$$

This is the system equation for a perfect state information problem with state b_k, control u_k, "disturbance" z_{k+1}, and cost per stage $\hat{g}_k(b_k, u_k)$. The function F_k is viewed as a sequential *belief estimator*, which updates the current belief state b_k based on the new observation z_{k+1}. It is given by either an explicit formula or an algorithm (such as Kalman filtering or particle filtering) that is based on the probability distribution of z_k and the use of Bayes' rule.

The expected value $E_{z_{k+1}}\{\cdot \mid b_k, u_k\}$ is taken with respect to the distribution of z_{k+1}, given b_k and u_k. Note that z_{k+1} is random, and its distribution depends on x_k and u_k, so the expected value

$$E_{z_{k+1}} \Big\{ J_{k+1}^*\big(F_k(b_k, u_k, z_{k+1})\big) \;\Big|\; b_k, u_k \Big\}$$

in Eq. (1.73) is a function of b_k and u_k.

The algorithm (1.73) is just the ordinary DP algorithm for the perfect state information problem shown in Fig. 1.6.7. It involves the system/belief estimator (1.74) and the cost per stage $\hat{g}_k(b_k, u_k)$. Note that since b_k takes values in a continuous space, the algorithm (1.73) will typically require an approximate implementation, using approximation in value space methods.

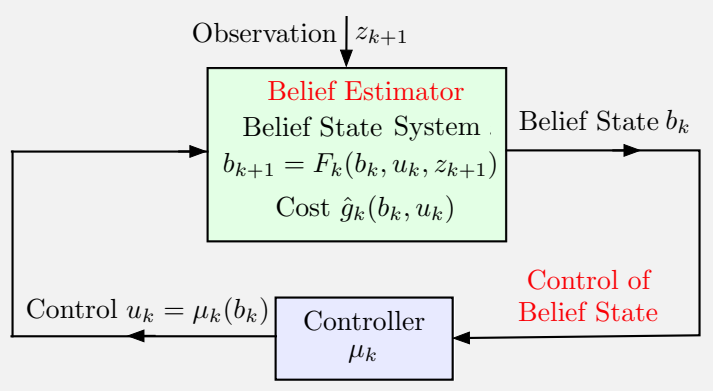

Figure 1.6.7 Schematic illustration of the view of an imperfect state information problem as one of perfect state information, whose state is the belief state b_k, i.e., the conditional probability distribution of x_k given all the observations up to time k. The observation z_{k+1} plays the role of the stochastic disturbance. The function F_k is a sequential estimator that updates the current belief state b_k.

We refer to the textbook [Ber17a], Chapter 4, for a detailed derivation of the DP algorithm (1.73), and to the monograph [BeS78] for a mathematical treatment that applies to infinite-dimensional state and disturbance spaces as well.

An Alternative DP Algorithm for POMDP

The DP algorithm (1.73) is not the only one that can be used for POMDP. There is also an exact DP algorithm that operates in the space of information vectors I_k, defined by

$$I_k = \{z_0, u_0, \ldots, z_{k-1}, u_{k-1}, z_k\},$$

where z_k is the observation received at time k. This is another sufficient statistic, and hence an alternative to the belief state b_k. In particular, we can view I_k as a state of the POMDP, which evolves over time according to the equation

$$I_{k+1} = (I_k, z_{k+1}, u_k).$$

Denoting by $J_k^*(I_k)$ the optimal cost starting at information vector I_k at time k, the DP algorithm takes the form

$$J_k^*(I_k) = \min_{u_k \in U_k(x_k)} E_{w_k, z_{k+1}} \Big\{ g_k(x_k, u_k, w_k) + J_{k+1}^*\big(I_k, z_{k+1}, u_k\big) \mid I_k, u_k \Big\}, \tag{1.75}$$

for $k = 0, \ldots, N-1$, with $J_N^*(I_N) = E\big\{g_N(x_N) \mid I_N\big\}$; see e.g., the DP textbook [Ber17a], Section 4.1.

A drawback of the preceding approach is that the information vector I_k is growing in size over time, thereby leading to a nonstationary system even in the case of an infinite horizon problem with a stationary system and cost function. This difficulty can be remedied in an approximation scheme that uses a finite history of the system (a fixed number of most recent observations) as state, thereby working effectively with a stationary finite-state system; see the paper by White and Scherer [WhS94]. In particular, this approach is used in large language models such as ChatGPT.

Finite-memory approximations for POMDP can be viewed within the context of feature-based approximation architectures, as we will discuss in Chapter 3 (see Example 3.1.6). Moreover, the finite-history scheme can be generalized through the concept of a *finite-state controller*; see the paper by Yu and Bertsekas [YuB08], which also addresses the issue of convergence of the approximation error to zero as the size of the finite-history or finite-state controller is increased.

1.6.7 Multiagent Problems and Multiagent Rollout

In this book, we will view a multiagent system as a collection of decision making entities, called *agents*, which aim to optimally achieve a common goal.† The agents accomplish this by collecting and exchanging information, and otherwise interacting with each other. The agents can be software programs or physical entities such as robots, and they may have different capabilities.

Among the generic challenges of efficient implementation of multiagent systems, one may note issues of limited communication and lack of fully shared information, due to factors such as limited bandwidth, noisy channels, and lack of synchronization. Another important generic issue is that as the number of agents increases, the size of the set of possible joint decisions of the agents increases exponentially, thereby complicating control selection by lookahead minimization. In this section, we will focus on ways to resolve this latter difficulty for problems where the agents fully share information, and in Section 2.9 we will address some of the challenges of problems where the agents may have some autonomy, and act without fully coordinating with each other.

For a mathematical formulation, let us consider the discounted infinite horizon problem and a special structure of the control space, whereby the control u consists of m components, $u = (u^1, \ldots, u^m)$, with a separable control constraint structure $u^\ell \in U^\ell(x)$, $\ell = 1, \ldots, m$. Thus the control constraint set is the Cartesian product

$$U(x) = U^1(x) \times \cdots \times U^m(x), \tag{1.76}$$

† In a more general version of a multiagent system, which is outside our scope, the agents may have different goals, and act in their own self-interest.

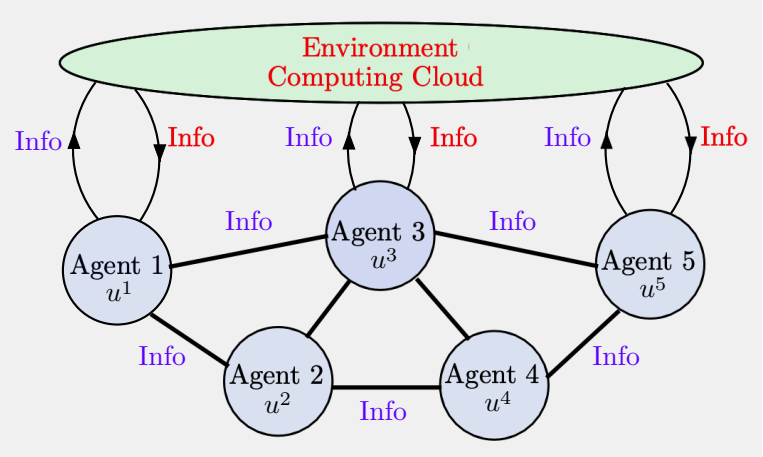

Figure 1.6.8 Schematic illustration of a multiagent problem. There are multiple "agents," and each agent $\ell = 1, \ldots, m$ controls its own decision variable u^ℓ. At each stage, agents exchange new information and also exchange information with the "environment," and then select their decision variables for the stage.

where the sets $U^\ell(x)$ are given. This structure arises in applications involving distributed decision making by multiple agents; see Fig. 1.6.8.

In particular, we will view each component u^ℓ, $\ell = 1, \ldots, m$, as being chosen from within $U^\ell(x)$ by a separate "agent" (a decision making entity). For the sake of the following discussion, we assume that each set $U^\ell(x)$ is finite. Then the one-step lookahead minimization of the standard rollout scheme with base policy μ is given by

$$\tilde{u} \in \arg\min_{u \in U(x)} E_w\Big\{g(x,u,w) + \alpha J_\mu\big(f(x,u,w)\big)\Big\}, \tag{1.77}$$

and involves as many as n^m Q-factors, where n is the maximum number of elements of the sets $U^\ell(x)$ [so that n^m is an upper bound to the number of controls in $U(x)$, in view of its Cartesian product structure (1.76)]. Thus the standard rollout algorithm requires an exponential [order $O(n^m)$] number of Q-factor computations per stage, which can be overwhelming even for moderate values of m.

This potentially large computational overhead motivates a far more computationally efficient rollout algorithm, whereby the one-step lookahead minimization (1.77) is replaced by a sequence of m successive minimizations, *one-agent-at-a-time*, with the results incorporated into the subsequent minimizations. In particular, given a base policy $\mu = (\mu^1, \ldots, \mu^m)$, we perform at state x the sequence of minimizations

$$\tilde{\mu}^1(x) \in \arg\min_{u^1 \in U^1(x)} E_w\Big\{g\big(x,u^1,\mu^2(x),\ldots,\mu^m(x),w\big) + \alpha J_\mu\big(f(x,u^1,\mu^2(x),\ldots,\mu^m(x),w)\big)\Big\},$$

$$\tilde{\mu}^2(x) \in \arg\min_{u^2 \in U^2(x)} E_w\Big\{g\big(x,\tilde{\mu}^1(x),u^2,\mu^3(x)\ldots,\mu^m(x),w\big) + \alpha J_\mu\big(f(x,\tilde{\mu}^1(x),u^2,\mu^3(x),\ldots,\mu^m(x),w)\big)\Big\},$$

$$\ldots \qquad \ldots \qquad \ldots \qquad \ldots$$

$$\tilde{\mu}^m(x) \in \arg\min_{u^m \in U^m(x)} E_w\Big\{g\big(x,\tilde{\mu}^1(x),\tilde{\mu}^2(x),\ldots,\tilde{\mu}^{m-1}(x),u^m,w\big) + \alpha J_\mu\big(f(x,\tilde{\mu}^1(x),\tilde{\mu}^2(x),\ldots,\tilde{\mu}^{m-1}(x),u^m,w)\big)\Big\}.$$

Thus each agent component u^ℓ is obtained by a minimization with the preceding agent components $u^1,\ldots,u^{\ell-1}$ fixed at the previously computed values of the rollout policy, and the following agent components $u^{\ell+1},\ldots,u^m$ fixed at the values given by the base policy. This algorithm requires order $O(nm)$ Q-factor computations per stage, a potentially huge computational saving over the order $O(n^m)$ computations required by standard rollout.

A key idea here is that the computational requirements of the rollout one-step minimization (1.77) are proportional to the number of controls in the set $U(x_k)$ and are independent of the size of the state space. This motivates a reformulation of the problem, first suggested in the book [BeT96], Section 6.1.4, whereby *control space complexity is traded off with state space complexity*, by "unfolding" the control u_k into its m components, which are applied *one agent-at-a-time* rather than all-agents-at-once.

In particular, we can reformulate the problem by breaking down the collective decision u_k into m individual component decisions, thereby reducing the complexity of the control space while increasing the complexity of the state space. The potential advantage is that *the extra state space complexity does not affect the computational requirements of some RL algorithms, including rollout*.

To this end, we introduce a modified but equivalent problem, involving one-at-a-time agent control selection. At the generic state x, we break down the control u into the sequence of the m controls $u^1,u^2,\ldots,u^m$, and between x and the next state $\bar{x} = f(x,u,w)$, we introduce artificial intermediate "states" $(x,u^1),(x,u^1,u^2),\ldots,(x,u^1,\ldots,u^{m-1})$, and corresponding transitions. The choice of the last control component u^m at "state" $(x,u^1,\ldots,u^{m-1})$ marks the transition to the next state $\bar{x} = f(x,u,w)$ according to the system equation, while incurring cost $g(x,u,w)$; see Fig. 1.6.9.

It is evident that this reformulated problem is equivalent to the original, since any control choice that is possible in one problem is also possible in the other problem, while the cost structure of the two problems is the same. In particular, every policy $\mu = (\mu^1,\ldots,\mu^m)$ of the original problem, including a base policy in the context of rollout, is admissible for the reformulated problem, and has the same cost function for the original as well as the reformulated problem.

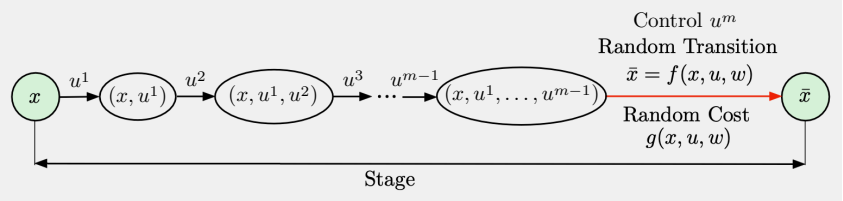

Figure 1.6.9 Equivalent formulation of the stochastic optimal control problem for the case where the control u consists of m components $u^1, u^2, \ldots, u^m$:

$$u = (u^1, \ldots, u^m) \in U(x) = U^1(x) \times \cdots \times U^m(x).$$

The figure depicts the kth stage transitions. Starting from state x, we generate the intermediate states

$$(x, u^1), (x, u^1, u^2), \ldots, (x, u^1, \ldots, u^{m-1}),$$

using the respective controls $u^1, \ldots, u^{m-1}$. The final control u^m leads from $(x, u^1, \ldots, u^{m-1})$ to $\bar{x} = f(x, u, w)$, and the random cost $g(x, u, w)$ is incurred.

The motivation for the reformulated problem is that the control space is simplified at the expense of introducing $m-1$ additional layers of states, and the corresponding $m-1$ cost-to-go functions

$$J^1(x, u^1), J^2(x, u^1, u^2), \ldots, J^{m-1}(x, u^1, \ldots, u^{m-1}).$$

The increase in size of the state space does not adversely affect the operation of rollout, since the Q-factor minimization (1.77) is performed for just one state at each stage.

The major fact that can be proved about multiagent rollout (see Section 2.9 and the end-of-chapter references) is that it *achieves cost improvement*:

$$J_{\tilde{\mu}}(x) \leq J_\mu(x), \qquad \text{for all } x,$$

where $J_\mu(x)$ is the cost function of the base policy $\mu = (\mu^1, \ldots, \mu^m)$, and $J_{\tilde{\mu}}(x)$ is the cost function of the rollout policy $\tilde{\mu} = (\tilde{\mu}^1, \ldots, \tilde{\mu}^m)$, starting from state x. Furthermore, this cost improvement property can be extended to multiagent PI schemes that involve one-agent-at-a-time policy improvement operations, and have sound convergence properties. Moreover, multiagent rollout becomes the starting point for related PI schemes that are well suited for distributed operation in contexts involving multiple autonomous decision makers; see Section 2.9, the book [Ber20a], the papers [Ber20b] and [BKB20], and the tutorial survey [Ber21a].

Example 1.6.5 (Spiders and Flies)

This example is representative of a broad range of practical problems such as multirobot service systems involving delivery, maintenance and repair, search

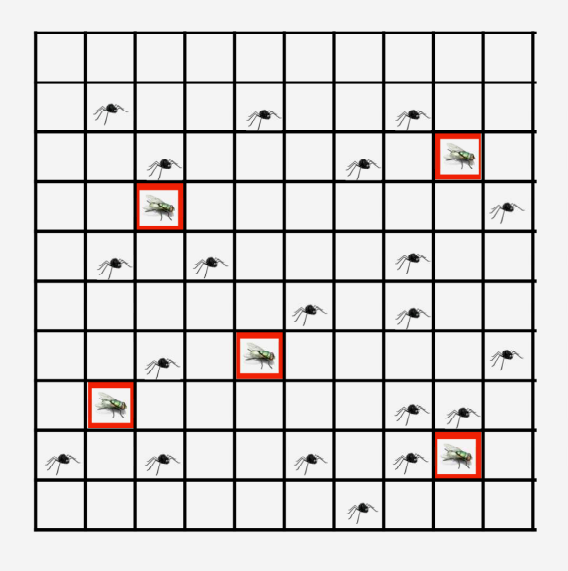

Figure 1.6.10 Illustration of a 2-dimensional spiders-and-fly problem with 20 spiders and 5 flies (cf. Example 1.6.5). The flies moves randomly, regardless of the position of the spiders. During a stage, each spider moves to a neighboring location or stays where it is, so there are 5 moves per spider (except for spiders at the edges of the grid). The total number of possible joint spiders moves is a little less than 5^{20}.

and rescue, firefighting, etc. Here there are m spiders and several flies moving on a 2-dimensional grid; cf. Fig. 1.6.10. The objective is for the spiders to catch all the flies as fast as possible.

During a stage, each fly moves to a some other position according to a given state-dependent probability distribution. Each spider learns the current state (the vector of spiders and fly locations) at the beginning of each stage, and either moves to a neighboring location or stays where it is. Thus each spider has as many as 5 choices at each stage. The control is $u = (u^1, \ldots, u^m)$, where u^ℓ is the choice of the ℓth spider, so there are about 5^m possible values of u.

To apply multiagent rollout, we need a base policy. A simple possibility is to use the policy that directs each spider to move on the path of minimum distance to the closest fly position. According to the multiagent rollout formalism, the spiders choose their moves one-at-time in the order from 1 to m, taking into account the current positions of the flies and the earlier moves of other spiders, and assuming that future moves will be chosen according to the base policy, which is a tractable computation.

In particular, at the beginning at the typical stage, spider 1 selects its best move (out of the no more than 5 possible moves), assuming the other spiders $2, \ldots, m$ will move towards their closest surviving fly during the current stage, and all spiders will move towards their closest surviving fly during the following stages, up to the time where no surviving flies remain.

Spider 1 then broadcasts its selected move to all other spiders. Then spider 2 selects its move taking into account the move already chosen by spider 1, and assuming that spiders $3, \ldots, m$ will move towards their closest surviving fly during the current stage, and all spiders will move towards their closest surviving fly during the following stages, up to the time where no surviving flies remain. Spider 2 then broadcasts its choice to all other spiders. This process of one-spider-at-a-time move selection is repeated for the remaining spiders $3, \ldots, m$, marking the end of the stage.

Note that while standard rollout computes and compares 5^m Q-factors (actually a little less to take into account edge effects), multiagent rollout computes and compares ≤ 5 moves per spider, for a total of less than $5m$. Despite this tremendous computational economy, experiments with this type of spiders and flies problems have shown that multiagent rollout achieves a comparable performance to the one of standard rollout.

1.6.8 Problems with Unknown Parameters - Adaptive Control

Our discussion so far dealt with problems with a known mathematical model, i.e., one where the system equation, cost function, control constraints, and probability distributions of disturbances are perfectly known. The mathematical model may be available through explicit mathematical formulas and assumptions, or through a computer program that can emulate all of the mathematical operations involved in the model, including Monte Carlo simulation for the calculation of expected values.

It is important to note here that from our point of view, *it makes no difference whether the mathematical model is available through closed form mathematical expressions or through a computer simulator*: the methods that we discuss are valid either way, only their suitability for a given problem may be affected by the availability of mathematical formulas.

In practice, however, it is common that the system parameters are either not known exactly or can change over time, and this introduces potentially enormous complications.† As an example consider our oversimplified cruise control system that we noted in Example 1.3.1 or its infinite horizon version. The state evolves according to

$$x_{k+1} = x_k + bu_k + w_k, \tag{1.78}$$

where x_k is the deviation $v_k - \bar{v}$ of the vehicle's velocity v_k from the nominal $\bar{v}$, u_k is the force that propels the car forward, and w_k is the disturbance

† The difficulties of decision and control within a changing environment are often underestimated. Among others, they complicate the balance between offline training and on-line play, which we discussed in Section 1.1 in connection the AlphaZero. It is worth keeping in mind that as much as learning to play high quality chess is a great challenge, the rules of play are stable and do not change unpredictably in the middle of a game! Problems with changing system parameters can be far more challenging!

that has nonzero mean. However, the coefficient b and the distribution of w_k change frequently, and cannot be modeled with any precision because they depend on unpredictable time-varying conditions, such as the slope and condition of the road, and the weight of the car (which is affected by the number of passengers). Moreover, the nominal velocity $\bar{v}$ is set by the driver, and when it changes it may affect the parameter b in the system equation, and other parameters.†

In this section, we will briefly review some of the most commonly used approaches for dealing with unknown parameters in optimal control theory and practice. We should note also that unknown problem environments are an integral part of the artificial intelligence view of RL. In particular, to quote from the popular book by Sutton and Barto [SuB18], RL is viewed as "a computational approach to learning from interaction," and "learning from interaction with the environment is a foundational idea underlying nearly all theories of learning and intelligence."

The idea of learning from interaction with the environment is often connected with the idea of exploring the environment to identify its characteristics. In control theory this is often viewed as part of the *system identification* methodology, which aims to construct mathematical models of dynamic systems. The system identification process is often combined with the control process to deal with unknown or changing problem parameters, in a framework that is sometimes called *dual control*. This is one of the most challenging areas of stochastic optimal and suboptimal control, and has been studied intensively since the early 1960s.

Robust and Adaptive Control

Given a controller design that has been obtained assuming a nominal DP problem model, one possibility is to simply ignore changes in problem parameters. We may then try to investigate the performance of the current design for a suitable range of problem parameter values, and ensure that it is adequate for the entire range. This is sometimes called a *robust controller design*. For example, consider the oversimplified cruise control system of Eq. (1.78) with a linear controller of the form $\mu(x) = Lx$ for some scalar L. Then we check the range of parameters b for which the current controller is stable (this is the interval of values b for which $|1 + bL| < 1$), and ensure that b remains within that range during the system's operation.

The more general class of methods where the controller is modified in response to problem parameter changes is part of a broad field known as *adaptive control*, i.e., control that adapts to changing parameters. This is a rich methodology with many and diverse applications. Our discussion of

† Adaptive cruise control, which can also adapt the car's velocity based on its proximity to other cars, has been studied extensively and has been incorporated in several commercially sold car models.

adaptive control in this book will be limited. Let us just mention for the moment a simple time-honored adaptive control approach for continuous-state problems called *PID (Proportional-Integral-Derivative) control*, for which we refer to the control literature, including the books by Aström and Hagglund [AsH95], [AsH06], and the end-of-chapter references on adaptive control (also the discussion in Section 5.7 of the RL textbook [Ber19a]).

In particular, PID control aims to maintain the output of a single-input single-output dynamic system around a set point or to follow a given trajectory, as the system parameters change within a relatively broad range. In its simplest form, the PID controller is parametrized by three scalar parameters, which may be determined by a variety of methods, some of them manual/heuristic. PID control is used widely and with success, although its range of application is mainly restricted to single-input, single-output continuous-state control systems.

Dealing with Unknown Parameters Through System Identification

In PID control, no attempt is made to maintain a mathematical model and to track unknown model parameters as they change. An alternative and apparently reasonable form of suboptimal control is to separate the control process into two phases, a *system identification phase* and a *control phase*. In the first phase the unknown parameters are estimated, while the control takes no account of the interim results of estimation. The final parameter estimates from the first phase are then used to implement an optimal or suboptimal policy in the second phase. This alternation of estimation and control phases may be repeated several times during any system run in order to take into account subsequent changes of the parameters. Moreover, it is not necessary to introduce a hard separation between the identification and the control phases. They may be going on simultaneously, with new parameter estimates being introduced into the control process, whenever this is thought to be desirable; see Fig. 1.6.11.

One drawback of this approach is that it is not always easy to determine when to terminate one phase and start the other. A second difficulty, of a more fundamental nature, is that the control process may make some of the unknown parameters invisible to the estimation process. This is known as the problem of *parameter identifiability*, which is discussed in the context of optimal control in several sources, including [BoV79] and [Kum83]; see also [Ber17a], Section 6.7.

Example 1.6.6 (Parameter Identifiability Under Closed-Loop Control)

For a simple example, consider the scalar system

$$x_{k+1} = ax_k + bu_k, \qquad k = 0, \ldots, N-1,$$

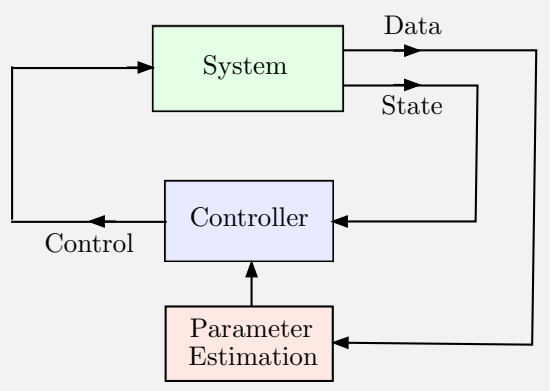

Figure 1.6.11 Schematic illustration of concurrent parameter estimation and system control. The system parameters are estimated on-line and the estimates are periodically passed on to the controller.

and the quadratic cost

$$\sum_{k=1}^{N}(x_k)^2.$$

Assuming perfect state information, if the parameters a and b are known, it can be seen that the optimal control law is

$$\mu_k^*(x_k) = -\frac{a}{b}x_k,$$

which sets all future states to 0. Assume now that the parameters a and b are unknown, and consider the two-phase method. During the first phase the control law

$$\tilde{\mu}_k(x_k) = \gamma x_k \tag{1.79}$$

is used (γ is some scalar; for example, $\gamma = -\frac{\overline{a}}{\overline{b}}$, where $\overline{a}$ and $\overline{b}$ are some a priori estimates of a and b, respectively). At the end of the first phase, the control law is changed to

$$\overline{\mu}_k(x_k) = -\frac{\hat{a}}{\hat{b}}x_k,$$

where $\hat{a}$ and $\hat{b}$ are the estimates obtained from the estimation process. However, with the control law (1.79), the closed-loop system is $x_{k+1} = (a+b\gamma)x_k$, so the estimation process can at best yield the value of $(a + b\gamma)$ but not the values of both a and b. In other words, the estimation process cannot discriminate between pairs of values (a_1, b_1) and (a_2, b_2) such that

$$a_1 + b_1\gamma = a_2 + b_2\gamma.$$

Therefore, a and b are not identifiable when feedback control of the form (1.79) is applied.

On-line parameter estimation algorithms, which address among others the issue of identifiability, have been discussed extensively in the control theory literature, but the corresponding methodology is complex and beyond our scope in this book. However, assuming that we can make the estimation phase work somehow, we are free to revise the controller using the newly estimated parameters in a variety of ways, in an on-line replanning process.

Unfortunately, there is still another difficulty with this type of on-line replanning: it may be hard to recompute an optimal or near-optimal policy on-line, using a newly identified system model. In particular, it may be impossible to use time-consuming methods that involve for example the training of a neural network or discrete/integer control constraints. A simpler possibility is to use rollout, which we discuss next.†

Adaptive Control by Rollout and On-Line Replanning

We will now consider an approach for dealing with unknown or changing parameters, which is based on on-line replanning. We have discussed this approach in the context of rollout and multiagent rollout, where we stressed the importance of fast on-line policy improvement.

Let us assume that some problem parameters change and the current controller becomes aware of the change "instantly" (i.e., very quickly before the next stage begins). The method by which the problem parameters are recalculated or become known is immaterial for the purposes of the following discussion. It may involve a limited form of parameter estimation, whereby the unknown parameters are "tracked" by data collection over a few time stages, with due attention paid to issues of parameter identifiability; or it may involve new features of the control environment, such as a changing number of servers and/or tasks in a service system (think of new spiders and/or flies appearing or disappearing unexpectedly in the spiders-and-flies Example 1.6.5).

We thus assume away/ignore issues of parameter estimation, and focus on revising the controller by on-line replanning based on the newly obtained parameters. This revision may be based on any suboptimal method, but rollout with the current policy used as the base policy is particularly

† Another possibility is to deal with this difficulty by precomputation. In particular, assume that the set of problem parameters may take a known finite set of values (for example each set of parameter values may correspond to a distinct maneuver of a vehicle, motion of a robotic arm, flying regime of an aircraft, etc). Then we may precompute a separate controller for each of these values. Once the control scheme detects a change in problem parameters, it switches to the corresponding predesigned current controller. This is sometimes called a *multiple model control design* or *gain scheduling*, and has been applied with success in various settings over the years.

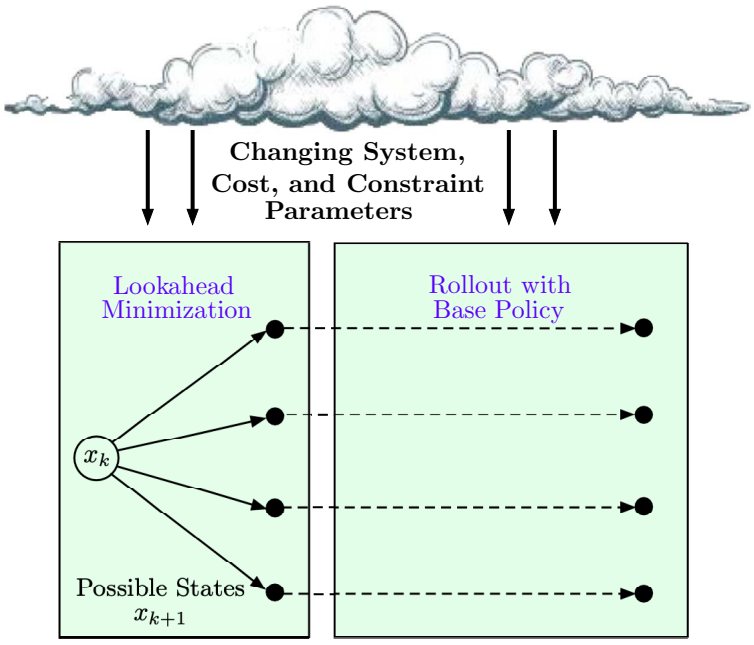

Figure 1.6.12 Schematic illustration of adaptive control by rollout. One-step lookahead is followed by simulation with the base policy, which stays fixed. The system, cost, and constraint parameters are changing over time, and the most recent values are incorporated into the lookahead minimization and rollout operations. For the discussion in this section, we may assume that all the changing parameter information is provided by some computation and sensor "cloud" that is beyond our control. The base policy may also be revised based on various criteria.

attractive. Here the advantage of rollout is that it is simple and reliable. In particular, it does not require a complicated training procedure to revise the current policy, based for example on the use of neural networks or other approximation architectures, so *no new policy is explicitly computed in response to the parameter changes*. Instead the current policy is used as the base policy for rollout, and the available controls at the current state are compared by a one-step or mutistep minimization, with cost function approximation provided by the base policy (cf. Fig. 1.6.12).

Note that *over time the base policy may also be revised* (on the basis of an unspecified rationale), in which case the rollout policy will be revised both in response to the changed current policy and in response to the changing parameters. This is necessary in particular when the constraints of the problem change.

The principal requirement for using rollout in an adaptive control context is that the rollout control computation should be fast enough to be performed between stages. Note, however, that accelerated/truncated

versions of rollout, as well as parallel computation, can be used to meet this time constraint.

The following example considers on-line replanning with the use of rollout in the context of the simple one-dimensional linear quadratic problem that we discussed earlier in this chapter. The purpose of the example is to illustrate analytically how rollout with a policy that is optimal for a nominal set of problem parameters works well when the parameters change from their nominal values. This property is not practically useful in linear quadratic problems because when the parameter change, it is possible to calculate the new optimal policy in closed form, but it is indicative of the performance robustness of rollout in other contexts. Generally, adaptive control by rollout and on-line replanning makes sense in situations where the calculation of the rollout controls for a given set of problem parameters is faster and/or more convenient than the calculation of the optimal controls for the same set of parameter values. These problems include cases involving nonlinear systems and/or difficult (e.g., integer) constraints.

Example 1.6.7 (On-Line Replanning for Linear Quadratic Problems Based on Rollout)

Consider the deterministic undiscounted infinite horizon linear quadratic problem. It involves the linear system

$$x_{k+1} = x_k + bu_k,$$

and the quadratic cost function

$$\lim_{N\to\infty} \sum_{k=0}^{N-1} (x_k^2 + ru_k^2).$$

The optimal cost function is given by

$$J^*(x) = K^* x^2,$$

where K^* is the unique positive solution of the Riccati equation

$$K = \frac{rK}{r + b^2 K} + 1. \tag{1.80}$$

The optimal policy has the form

$$\mu^*(x) = L^* x, \tag{1.81}$$

where

$$L^* = -\frac{bK^*}{r + b^2 K^*}. \tag{1.82}$$

As an example, consider the optimal policy that corresponds to the nominal problem parameters $b = 2$ and $r = 0.5$: this is the policy (1.81)-(1.82), with K obtained as the positive solution of the quadratic Riccati Eq. (1.80) for $b = 2$ and $r = 0.5$. In particular, we can verify that

$$K = \frac{2 + \sqrt{6}}{4}.$$

From Eq. (1.82) we then obtain

$$L = -\frac{2 + \sqrt{6}}{5 + 2\sqrt{6}}. \tag{1.83}$$

We will now consider changes of the values of b and r while keeping L constant, and we will compare the quadratic cost coefficient of the following three cost functions as b and r vary:

(a) The optimal cost function K^*x^2, where K^* is given by the positive solution of the Riccati Eq. (1.80).

(b) The cost function K_Lx^2 that corresponds to the base policy

$$\mu_L(x) = Lx,$$

where L is given by Eq. (1.83). From our earlier discussion, we have

$$K_L = \frac{1 + rL^2}{1 - (1 + bL)^2}.$$

(c) The cost function $\tilde{K}_Lx^2$ that corresponds to the rollout policy

$$\tilde{\mu}_L(x) = \tilde{L}x,$$

obtained by using the policy μ_L as base policy. Using the formulas given earlier, we have

$$\tilde{L} = -\frac{bK_L}{r + b^2K_L},$$

and

$$\tilde{K}_L = \frac{1 + r\tilde{L}^2}{1 - (1 + b\tilde{L})^2}.$$

Figure 1.6.13 shows the coefficients K^*, K_L, and $\tilde{K}_L$ for a range of values of r and b. We have

$$K^* \leq \tilde{K}_L \leq K_L.$$

The difference $K_L - K^*$ is indicative of the robustness of the policy μ_L, i.e., the performance loss incurred by ignoring the values of b and r, and continuing

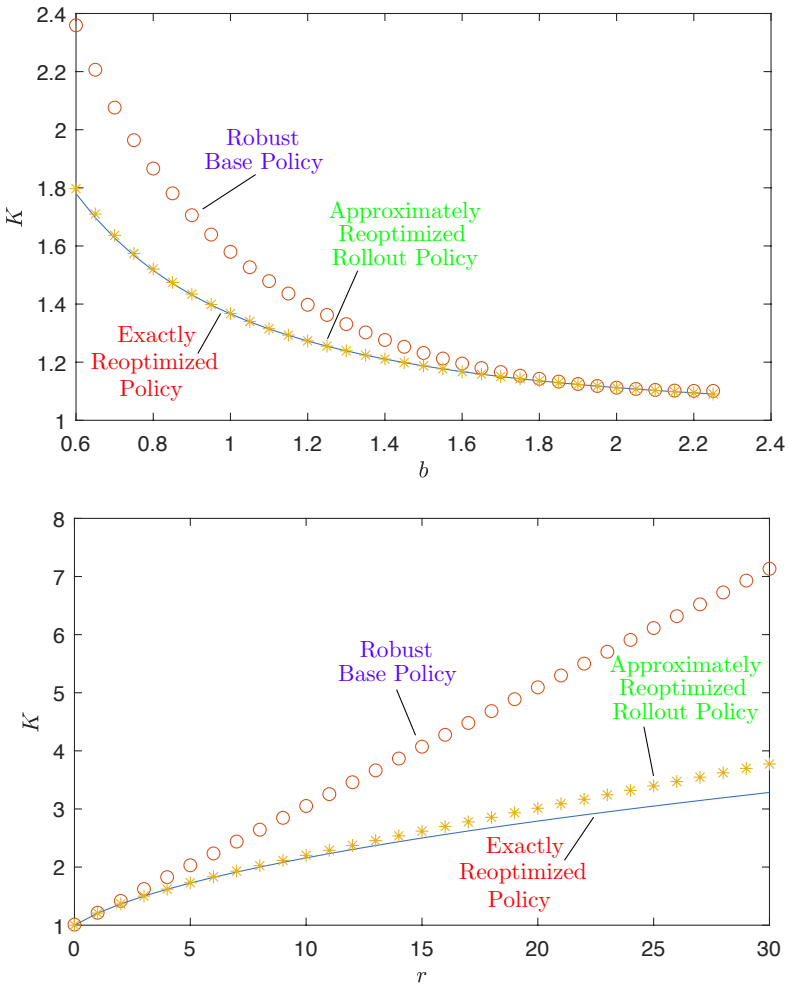

Figure 1.6.13 Illustration of adaptive control by rollout under changing problem parameters. The quadratic cost coefficients K^* (optimal, denoted by solid line), K_L (base policy, denoted by circles), and $\tilde{K}_L$ (rollout policy, denoted by asterisks) for the two cases where $r = 0.5$ and b varies, and $b = 2$ and r varies. The value of L is fixed at the value that is optimal for $b = 2$ and $r = 0.5$ [cf. Eq. (1.83)].

The rollout policy performance is very close to the one of the exactly reoptimized policy, while the base policy yields much worse performance. This is a consequence of the quadratic convergence rate of Newton's method that underlies rollout:

$$\lim_{J \to J^*} \frac{\tilde{J} - J^*}{J - J^*} = 0,$$

where for a given initial state, $\tilde{J}$ is the rollout performance, J^* is the optimal performance, and J is the base policy performance.

to use the policy μ_L, which is optimal for the nominal values $b = 2$ and $r = 0.5$, but suboptimal for other values of b and r. The difference $\tilde{K}_L - K^*$ is indicative of the performance loss due to using on-line replanning by rollout rather than using optimal replanning. Finally, the difference $K_L - \tilde{K}_L$ is indicative of the performance improvement due to on-line replanning using rollout rather than keeping the policy μ_L unchanged.

Note that Fig. 1.6.13 illustrates the behavior of the error ratio

$$\frac{\tilde{J} - J^*}{J - J^*},$$

where for a given initial state, $\tilde{J}$ is the rollout performance, J^* is the optimal performance, and J is the base policy performance. This ratio approaches 0 as $J - J^*$ becomes smaller because of the quadratic convergence rate of Newton's method that underlies the rollout algorithm.

Adaptive Control as POMDP

The preceding adaptive control formulation strictly separates the dual objective of estimation and control: first parameter identification and then controller reoptimization (either exact or rollout-based). In an alternative adaptive control formulation, the parameter estimation and the application of control are done simultaneously, and indeed part of the control effort may be directed towards improving the quality of future estimation. This alternative (and more principled) approach is based on a view of adaptive control as a partially observed Markovian decision problem (POMDP) with a special structure. We will see in Section 2.11 that this approach is well-suited for approximation in value space schemes, including forms of rollout.

To describe briefly the adaptive control reformulation as POMDP, we introduce a system whose state consists of two components:

(a) A perfectly observed component x_k that evolves over time according to a discrete-time equation.

(b) A component θ which is unobserved but stays constant, and is estimated through the perfect observations of the component x_k.

We view θ as a parameter in the system equation that governs the evolution of x_k. Thus we have

$$x_{k+1} = f_k(x_k, \theta, u_k, w_k), \tag{1.84}$$

where u_k is the control at time k, selected from a set $U_k(x_k)$, and w_k is a random disturbance with given probability distribution that depends on (x_k, θ, u_k). For convenience, we will assume that θ can take one of m known values $\theta^1, \ldots, \theta^m$.

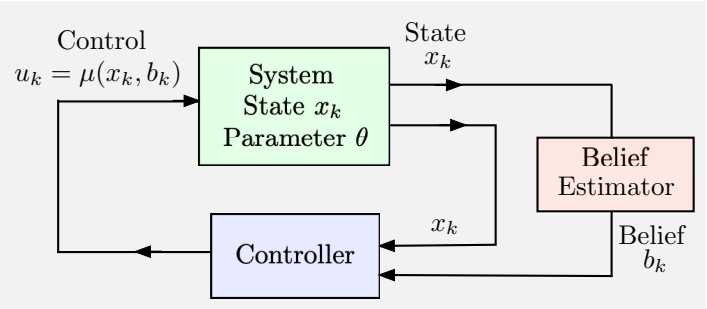

Figure 1.6.14 Schematic illustration of simultaneous control and belief estimation for the unknown system parameter θ. The control applied is a function of the current belief state (x_k, b_k), where b_k is the conditional probability distribution of θ given the observations accumulated up to time k (the current and past states $x_k, \ldots, x_0$, and the past controls $u_{k-1}, \ldots, u_0$).

The a priori probability distribution of θ is given and is updated based on the observed values of the state components x_k and the applied controls u_k. In particular, the information vector

$$I_k = \{x_0, \ldots, x_k, u_0, \ldots, u_{k-1}\}$$

is available at time k, and is used to compute the conditional probabilities

$$b_{k,i} = P\{\theta = \theta^i \mid I_k\}, \qquad i = 1, \ldots, m.$$

These probabilities form a vector

$$b_k = (b_{k,1}, \ldots, b_{k,m}),$$

which together with the perfectly observed state x_k, form the pair (x_k, b_k), which is the *belief state* of the POMDP at time k. The overall control scheme takes the form illustrated in Fig. 1.6.14.

As discussed in Section 1.6.4, an exact DP algorithm can be written for the equivalent POMDP, and this algorithm is suitable for the use of approximation in value space and rollout. We will describe this approach in some detail in Section 2.11. Related ideas will also be discussed in the context of Bayesian estimation and sequential estimation in Section 2.10.

Note that the case of a deterministic system

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

is particularly interesting, because we can then typically expect that the true parameter θ^* will be identified in a finite number of stages. The reason is that at each stage k, we are receiving a noiseless observation relating to θ, namely the state x_k. Once the true parameter θ^* is identified, the problem becomes one of perfect state information.

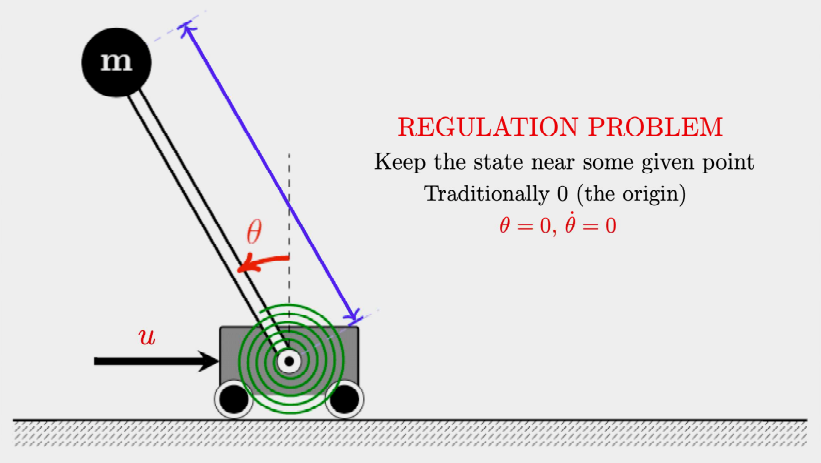

Figure 1.6.15 Illustration of a classical regulation problem, known as the "cart-pole problem" or "inverse pendulum problem." The state is the two-dimensional vector of angular position and angular velocity. We aim to keep the pole at the upright position (state equal to 0) by exerting horizontal force u on the cart.

1.6.9 Model Predictive Control

In this section, we will provide a brief summary of the model predictive control (MPC) methodology for control system design, with a view towards its connection with approximation in value space and rollout schemes.† We will focus on classical control problems, where the objective is to keep the state of a deterministic system close to the origin of the state space (see Fig. 1.6.15). Another type of classical control problem is to keep the system close to a given trajectory (see Fig. 1.6.16). It can also be treated by forms of MPC, but will not be discussed in this book.

We discussed earlier the linear quadratic approach, whereby the system is represented by a linear model, the cost is quadratic in the state and the control, and there are no state and control constraints. The linear quadratic and other approaches based on state variable system representations and optimal control became popular, starting in the late 50s and early 60s. Unfortunately, however, the analytically convenient linear quadratic problem formulations 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

† An extensive overview on the connections and applications of the conceptual framework of this book with model predictive and adaptive control is given in the author's paper [Ber24].

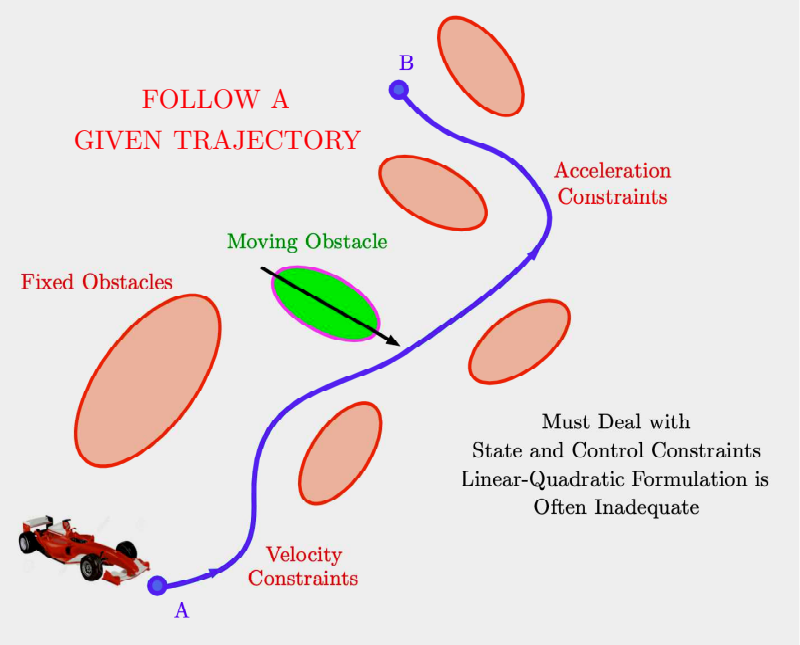

Figure 1.6.16 Illustration of constrained motion of a car from point A to point B. There are state (position/velocity) constraints, and control (acceleration) constraints. When there are mobile obstacles, the state constraints may change unpredictably, necessitating on-line replanning.

trajectory. Moreover, some of the control variables may be naturally discrete, and this is incompatible with the linear system viewpoint.

(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 car may be constrained by the presence of obstacles and hardware limitations (see Fig. 1.6.16). 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 formulation have motivated MPC, which combines elements of several ideas that we have discussed so far, such as multistep lookahead, rollout with a base policy, and certainty equivalence. Aside from dealing adequately with state and control constraints, MPC is well-suited for on-line replanning, like all rollout methods.

Note that the ideas of MPC were developed independently of the approximate DP/RL methodology. However, the two fields are closely related, and there is much to be gained from understanding one field within

the context of the other, as the subsequent development will aim to show. A major difference between MPC and finite-state stochastic control problems that are popular in the RL/artificial intelligence literature is that in MPC the state and control spaces are continuous/infinite, such as for example in self-driving cars, the control of aircraft and drones, or the operation of chemical processes.

In this section, we will primarily focus on the undiscounted infinite horizon deterministic problem, which involves the system

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

whose state x_k and control u_k are finite-dimensional vectors. The cost per stage is assumed nonnegative

$$g(x_k, u_k) \geq 0, \qquad \text{for all } (x_k, u_k),$$

(e.g., a positive definite quadratic cost). There are control constraints $u_k \in U(x_k)$, and to simplify the following discussion, we will initially consider no state constraints. We assume that the system can be kept at the origin at zero cost, i.e.,

$$f(0, \overline{u}_k) = 0, \quad g(0, \overline{u}_k) = 0 \qquad \text{for some control } \overline{u}_k \in U(0).$$

For a given initial state x_0, we want to obtain a sequence $\{u_0, u_1, \ldots\}$ that satisfies the control constraints, while minimizing the total cost.

This is a classical problem in control system design, known as the *regulation problem*, where the aim is to keep the state of the system near the origin (or more generally some desired set point), in the face of disturbances and/or parameter changes. In an important variant of the problem, there are additional state constraints of the form $x_k \in X$, and there arises the issue of maintaining the state within X, not just at the present time but also in future times. We will address this issue later in this section.

The Classical Form of MPC - View as a Rollout Algorithm

We will first focus on a classical form of the MPC algorithm, proposed in the form given here by Keerthi and Gilbert [KeG88]. In this algorithm, at each encountered state x_k, we apply a control $\tilde{u}_k$ that is computed as follows; see Fig. 1.6.17:

(a) We solve an ℓ-stage optimal control problem involving the same cost function and the requirement that the state after ℓ steps is driven to 0, i.e., $x_{k+\ell} = 0$. This is the problem

$$\min_{u_t,\, t=k,\ldots,k+\ell-1} \sum_{t=k}^{k+\ell-1} g(x_t, u_t), \tag{1.85}$$

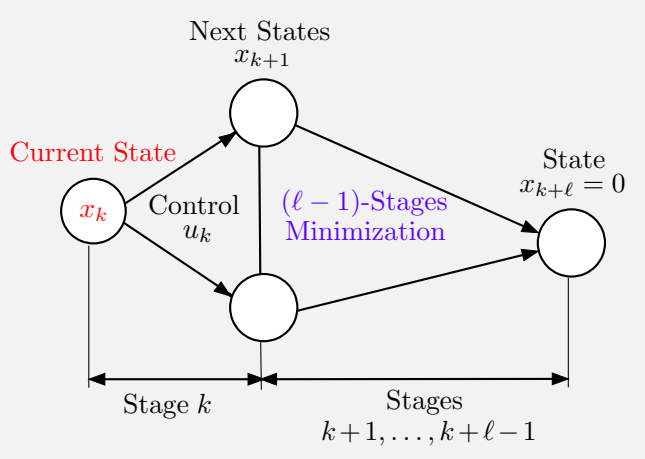

Figure 1.6.17 Illustration of the problem solved by a classical form of MPC at state x_k. We minimize the cost function over the next ℓ stages while imposing the requirement that $x_{k+\ell}=0$. We then apply the first control of the optimizing sequence. In the context of rollout, the minimization over u_k is the one-step lookahead, while the minimization over $u_{k+1},\ldots,u_{k+\ell-1}$ that drives $x_{k+\ell}$ to 0 is the base heuristic.

subject to the system equation constraints

$$x_{t+1}=f(x_t,u_t),\qquad t=k,\ldots,k+\ell-1, \tag{1.86}$$

the control constraints

$$u_t\in U(x_t),\qquad t=k,\ldots,k+\ell-1, \tag{1.87}$$

and the terminal state constraint

$$x_{k+\ell}=0. \tag{1.88}$$

Here ℓ is an integer with $\ell>1$, which is chosen in some largely empirical way.

(b) If $\{\tilde{u}_k,\ldots,\tilde{u}_{k+\ell-1}\}$ is the optimal control sequence of this problem, we apply $\tilde{u}_k$ and we discard the other controls $\tilde{u}_{k+1},\ldots,\tilde{u}_{k+\ell-1}$.

(c) At the next stage, we repeat this process, once the next state x_{k+1} is revealed.

To make the connection of the preceding MPC algorithm with rollout, we note that *the one-step lookahead function $\tilde{J}$ implicitly used by MPC [cf. Eq. (1.85)] is the cost function of a certain stable base policy*. This is the policy 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. This rollout view of MPC was first discussed in the author's paper [Ber05]. It is useful for making a connection with the approximate DP/RL, rollout, and its interpretation in terms of Newton's method. In particular, an important consequence is that *the MPC policy is stable*, since rollout with a stable base policy can be shown to yield a stable policy under very general conditions, as we have noted earlier for the special case of linear quadratic problems in Section 1.5; cf. Fig. 1.5.10.

We may also equivalently view the preceding MPC algorithm as rollout with $\bar{\ell}$-step lookahead, where $1 < \bar{\ell} < \ell$, with the base policy that drives to 0 the state after $\ell - \bar{\ell}$ stages and keeps the state at 0 thereafter. This suggests variations of MPC that involve truncated rollout with terminal cost function approximation, which we will discuss shortly.

Terminal Cost Approximation - Stability Issues

In a common variant of MPC, the requirement of driving the system state to 0 in ℓ steps in the ℓ-stage MPC problem (1.85), is replaced by a terminal cost $G(x_{k+\ell})$, which is positive everywhere except at 0. Thus at state x_k, we solve the problem

$$\min_{u_t,\, t=k,\ldots,k+\ell-1} \left[G(x_{k+\ell}) + \sum_{t=k}^{k+\ell-1} g(x_t, u_t) \right], \tag{1.89}$$

instead of problem (1.85) where we require that $x_{k+\ell} = 0$. This variant can be viewed as rollout with one-step lookahead, and a base policy, which at state x_{k+1} applies the first control $\tilde{u}_{k+1}$ of the sequence $\{\tilde{u}_{k+1}, \ldots, \tilde{u}_{k+\ell-1}\}$ that minimizes

$$G(x_{k+\ell}) + \sum_{t=k+1}^{k+\ell-1} g(x_t, u_t).$$

It can also be viewed outside the context of rollout, as approximation in value space with ℓ-step lookahead minimization and terminal cost approximation given by G. Thus the cost function of the preceding MPC controller may be much closer to J^* than G is.

An important question is to choose the terminal cost approximation so that the resulting MPC controller is stable. Our discussion of Section 1.5 on the region of stability of approximation in value space schemes applies here. In particular, under the nonnegative cost assumption of this section, the MPC controller can be proved to be stable if a single value iteration (VI) starting from G produces a function that takes uniformly smaller values than G:

$$\min_{u \in U(x)} \Big\{ g(x,u) + G\big(f(x,u)\big) \Big\} \leq G(x), \qquad \text{for all } x. \tag{1.90}$$

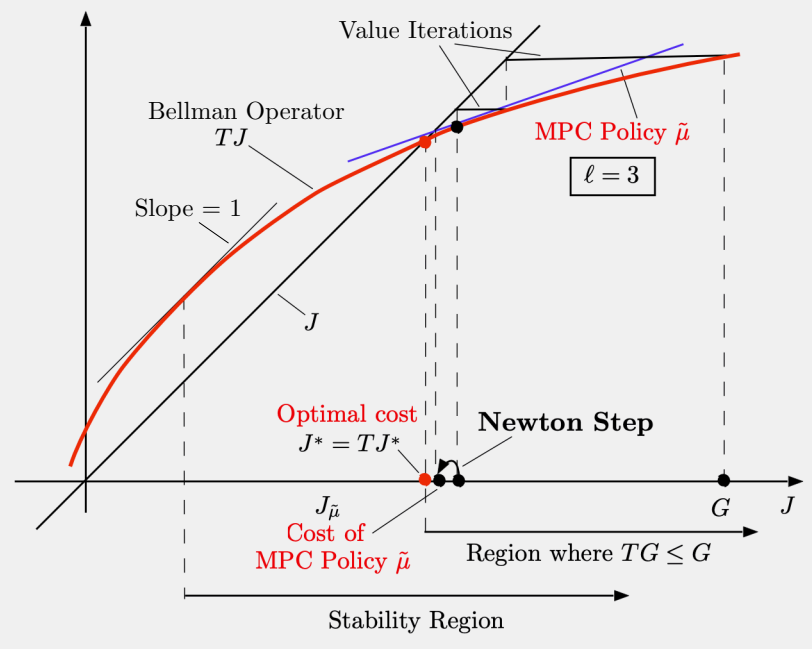

Figure 1.6.18 Illustration of the Bellman operator, defined by

$$(TJ)(x) = \min_{u \in U(x)} \Big\{ g(x,u) + J\big(f(x,u)\big) \Big\}, \qquad \text{for all } x.$$

The condition in (1.90) can be written compactly as $(TG)(x) \leq G(x)$ for all x. When satisfied by the terminal cost function G, it guarantees stability of the MPC policy $\tilde{\mu}$ with ℓ-step lookahead minimization. In this figure, $\ell = 3$.

Figure 1.6.18 provides a graphical illustration. It shows that this condition guarantees that successive iterates of value iteration, as implemented through multistep lookahead, lie within the region of stability, so that the policy produced by MPC is stable.

We also expect that as the length ℓ of the lookahead minimization is increased, the stability properties of the MPC controller are improved. In particular, *given $G \geq 0$, the resulting MPC controller is likely to be stable for ℓ sufficiently large*, since the VI algorithm ordinarily converges to J^*, which lies within the region of stability. Results of this type are known within the MPC framework under various conditions (see the papers by Mayne at al. [MRR00], Magni et al. [MDM01], the MPC book [RMD17], and the author's book [Ber20a], Section 3.1.2). Our discussion of stability in Section 1.5 is also relevant within this context; cf. Fig. 1.5.9.

In another variant of MPC, in addition to the terminal cost function approximation G, we use truncated rollout, which involves running some

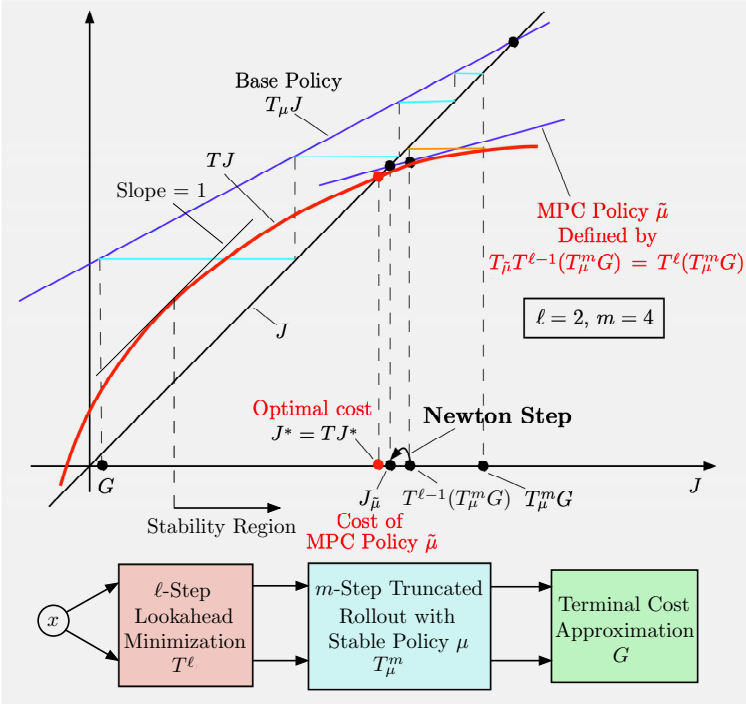

Figure 1.6.19 An MPC scheme with ℓ-step lookahead minimization, m-step truncated rollout with a stable base policy μ, and a terminal cost function approximation G, together with its interpretation as a Newton step. In this figure, $\ell = 2$ and $m = 4$. The truncated rollout with base policy μ consists of m value iterations with the Bellman operator corresponding to μ, which is given by

$$(T_\mu J)(x) = g\Big(x, \mu(x)\Big) + J\Big(f\Big(x, \mu(x)\Big)\Big).$$

Thus, truncated rollout applies m value iterations with base policy μ, starting with the function G and yielding the function $T_\mu^m G$. Then $\ell - 1$ value iterations are applied to $T_\mu^m G$ through the $(\ell - 1)$-step minimization. Finally, the Newton step is applied to

$$T^{\ell-1}(T_\mu^m G)$$

to yield the cost function of the MPC policy $\tilde{\mu}$. As m increases, the starting point for the Newton step moves closer to J_μ, which lies within the region of stability.

stable base policy μ for a number of steps m; see Fig. 1.6.19. This is quite similar to standard truncated rollout, except that the computational solution of the lookahead minimization problem (1.89) may become complicated when the control space is infinite. As discussed earlier in Section 1.5, *increasing the length of the truncated rollout enlarges the region of stability of the MPC controller*. The reason is that by increasing the length

of the truncated rollout, we push the start of the Newton step towards of the cost function J_μ of the stable policy, which lies within the region of stability. The base policy may also be used to address state constraints; see the papers by Rosolia and Borelli [RoB17], [RoB19], Li et al. [LJM21], and the discussions in the author's RL books [Ber20a], [Ber22a].

Finally, let is note that when faced with changing problem parameters, it is natural to consider on-line replanning as per our earlier adaptive control discussion. In this context, once new estimates of system and/or cost function parameters become available, MPC can adapt accordingly by introducing the new parameter estimates into the ℓ-stage optimization problem in (a) above.

State Constraints, Invariant Sets, and Off-Line Training

Our discussion so far has skirted a major issue in MPC, which is that there may be additional state constraints of the form $x_k \in X$, for all k, where X is some subset of the true state space. Indeed much of the original work on MPC was motivated by control problems with state constraints, imposed by the physics of the problem, which could not be handled effectively with the nice unconstrained framework of the linear quadratic problem that we have discussed in Section 1.5.

To deal with additional state constraints of the form $x_k \in X$, where X is some subset of the state space, the MPC problem to be solved at the kth stage [cf. Eq. (1.89)] must be modified. Assuming that the current state x_k belongs to the constraint set X, the MPC problem should take the form

$$\min_{u_t,\, t=k,\ldots,k+\ell-1} \left[G(x_{k+\ell}) + \sum_{t=k}^{k+\ell-1} g(x_t, u_t) \right], \tag{1.91}$$

subject to the control constraints

$$u_t \in U(x_t), \qquad t = k, \ldots, k+\ell-1, \tag{1.92}$$

and the state constraints

$$x_t \in X, \qquad t = k+1, \ldots, k+\ell. \tag{1.93}$$

The control $\tilde{u}_k$ thus obtained will generate a state

$$x_{k+1} = f(x_k, \tilde{u}_k)$$

that will belong to X, and similarly the entire state trajectory thus generated will satisfy the state constraint $x_t \in X$ for all t, assuming that the initial state does.

However, there is an important difficulty with the preceding MPC scheme, namely *there is no guarantee that the problem (1.91)-(1.93) has a feasible solution for all initial states* $x_k \in X$. Here is a simple example.

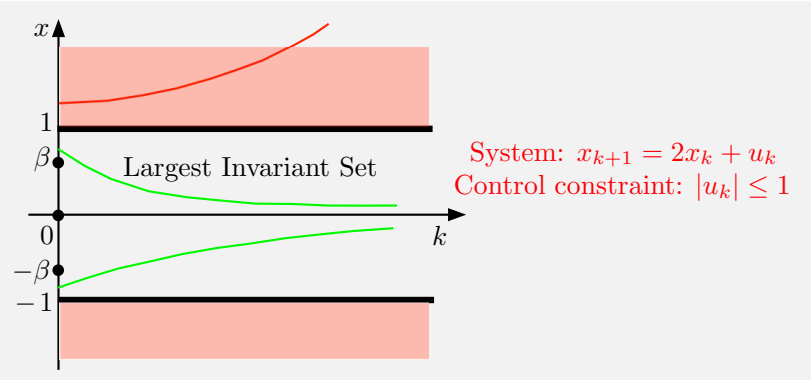

Figure 1.6.20 An illustration of invariance of a state constraint set X. Here the sets of the form $X = \{x_k \mid |x_k| \leq \beta\}$ are invariant for $\beta \leq 1$. For $\beta = 1$, we obtain the largest invariant set (the one that contains all other invariant sets). The figure shows some state trajectories produced by MPC. Note that starting with an initial condition x_0 with $|x_0| > 1$ (or $|x_0| < 1$) the closed-loop system obtained by MPC is unstable (or stable respectively); cf. the red and green trajectories shown.

Example 1.6.8 (State Constraints in MPC)

Consider the scalar system

$$x_{k+1} = 2x_k + u_k,$$

with control constraint

$$|u_k| \leq 1,$$

and state constraints of the form $x_k \in X$, for all k, where

$$X = \{x_k \mid |x_k| \leq \beta\}. \tag{1.94}$$

Then if $\beta > 1$, the state constraint cannot be satisfied for all initial states $x_0 \in X$. In particular, if we take $x_0 = \beta$, then $2x_0 > 2$ and $x_1 = 2x_0 + u_0$ will satisfy $x_1 > x_0 = \beta$ for any value of u_0 with $|u_0| \leq 1$. Similarly the entire sequence of states $\{x_k\}$ generated by any set of feasible controls will satisfy

$$x_{k+1} > x_k \quad \text{for all } k, \qquad x_k \uparrow \infty.$$

The state constraint can be satisfied only for initial states x_0 in the set $\hat{X}$ given by

$$\hat{X} = \{x_k \mid |x_k| \leq 1\};$$

see Fig. 1.6.20, which also illustrates the trajectories generated by the MPC scheme of Eq. (1.89), which does not involve state constraints.

The preceding example illustrates a fundamental point in state-constrained MPC: *the state constraint set X must be invariant in the sense that starting from any one of its points x_k there must exist a control $u_k \in U(x_k)$ for which the next state $x_{k+1} = f(x_k, u_k)$ must belong to X.* Mathematically, X is invariant if

$$\text{for every } x \in X, \text{ there exists } u \in U(x) \text{ such that } f(x,u) \in X.$$

In particular, it can be seen that the set X of Eq. (1.94) is invariant if and only if $\beta \leq 1$.

Given an MPC calculation of the form (1.91)-(1.93), we must make sure that the set X is invariant, or else it should be replaced by an invariant subset $\hat{X} \subset X$. Then the MPC calculation (1.91)-(1.93) will be feasible provided the initial state x_0 belongs to $\hat{X}$.

This brings up the question of how we compute an invariant subset of a given constraint set, which is typically an off-line calculation that cannot be performed during on-line play. It turns out that given X there exists a largest possible invariant subset of X, which can be computed in the limit with an algorithm that resembles value iteration. In particular, starting with $X_0 = X$, we obtain a nested sequence of subsets through the recursion

$$X_{k+1} = \big\{x \in X_k \mid f(x,u) \text{ belongs to } X_k \text{ for some } u \in U(x)\big\}, \quad k \geq 0. \tag{1.95}$$

Clearly, we have $X_{k+1} \subset X_k$ for all k, and under mild conditions it can be shown that the intersection set $\hat{X} = \cap_{k=0}^{\infty} X_k$, is the largest invariant subset of X; see the author's PhD thesis [Ber71] and subsequent paper [Ber72a], which introduced the concept of invariance and its use in satisfying state constraints in control over a finite and an infinite horizon.†

As an example, it can be verified that the sequence of value iterates (1.95) starting with a set $X_0 = \{x \mid |x| \leq \beta\}$ with $\beta > 1$ is given by

$$X_k = \{x \mid |x| \leq \beta_k\}, \quad \text{with } \beta_0 = \beta \text{ and } \beta_{k+1} = \tfrac{\beta_k + 1}{2} \text{ for all } k \geq 0.$$

It can thus be seen that we have $\beta_{k+1} < \beta_k$ for all k and $\beta_k \downarrow 1$, so that the intersection $\hat{X} = \cap_{k=0}^{\infty} X_k$ yields the largest invariant set

$$\hat{X} = \big\{x_k \mid |x_k| \leq 1\big\}.$$

There are several ways to compute invariant subsets of constraint sets X, for which we refer to the aforementioned author's work and the MPC literature; see e.g., the book by Rawlings, Mayne, and Diehl [RMD17], and

† The term used in [Ber71] and [Ber72a] is *reachability of a target tube* $\{X, X, \ldots\}$, which is synonymous to invariance of X.

the survey by Mayne [May14], which give additional references. An important point here is that the computation of an invariant subset of the given constraint set X must be done off-line with one of several available algorithmic approaches, so it becomes part of the off-line training (in addition to the terminal cost function G). A relatively simple possibility is to compute an invariant subset $\hat{X}$ that corresponds to some nominal policy $\hat{\mu}$ [i.e., starting from any point $x \in \hat{X}$, the state $f\big(x, \hat{\mu}(x)\big)$ belongs to $\hat{X}$]. Such an invariant subset may be obtained by some form of simulation using the policy $\hat{\mu}$. Moreover, $\hat{\mu}$ can also be used for truncated rollout and also provide a terminal cost function approximation.

Given an off-line training process, which provides an invariant set $\hat{X}$, a terminal cost function G, and possibly a base policy for truncated rollout, MPC becomes an on-line play algorithm for which our earlier discussion applies. Note, however, that in an adaptive control context, where a model is estimated on-line as it is changing, it may be difficult to recompute on-line an invariant set that can be used to enforce the state constraints of the problem. This is particularly so if the state constraints change themselves as part of the changing problem data.

Stochastic MPC by Certainty Equivalence

Let us finally mention that while in this section we have focused on deterministic problems, there are variants of MPC, which include the treatment of uncertainty. The books and papers cited earlier contain several ideas along these lines; see e.g. the books by Kouvaritakis and Cannon [KoC16], Rawlings, Mayne, and Diehl [RMD17], and the survey by Mesbah [Mes16].

In this connection, it is also worth mentioning the *certainty equivalence approach* that we discussed briefly earlier. In particular, upon reaching state x_k we may perform the MPC calculations after replacing the uncertain quantities $w_{k+1}, w_{k+2}, \ldots$ with deterministic quantities $\overline{w}_{k+1}, \overline{w}_{k+2}, \ldots$, *while allowing for the stochastic character of the disturbance w_k of just the current stage k*. Note that only the first step of this MPC calculation is stochastic. Thus the calculation needed per stage is not much more difficult than the one for deterministic problems, while still implementing a Newton step for solving the associated Bellman equation; see our earlier discussion, and also Section 2.5.3 of the RL book [Ber19a] and Section 3.2 of the book [Ber22a].

1.7 REINFORCEMENT LEARNING AND DECISION/CONTROL

The current state of RL has greatly benefited from the cross-fertilization of ideas from decision and control, and from artificial intelligence; see Fig. 1.7.1. The strong connections between these two fields are now widely recognized. Still, however, there are cultural differences, including the

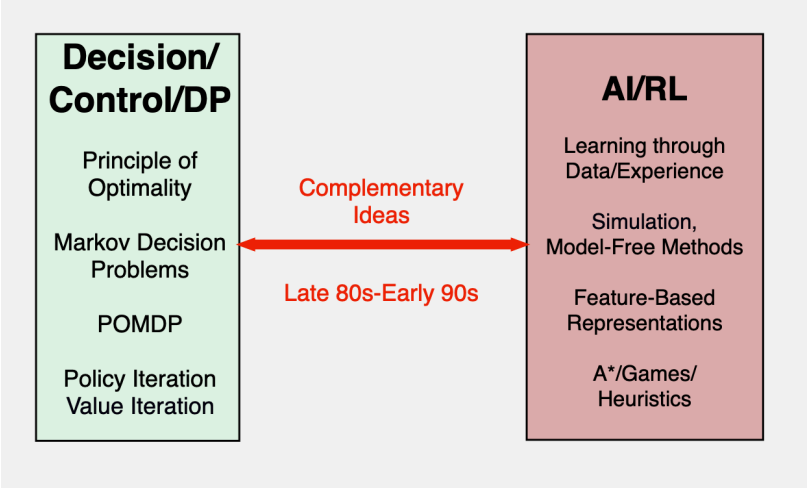

Figure 1.7.1 A schematic illustration of the synergy of ideas between artificial intelligence on one hand, and decision and control on the other.

traditional reliance on mathematical analysis for the decision and control field, and the emphasis on challenging problem implementations in the artificial intelligence field. Moreover, substantial differences in language and emphasis remain between RL-based discussions (where artificial intelligence-related terminology is used) and DP-based discussions (where optimal control-related terminology is used).

1.7.1 Differences in Terminology

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 AI or the optimal control terminology, we provide a list of terms commonly used in RL, and their optimal control counterparts.

(a) **Environment** = System.

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

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

(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) 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. (Q-value is also used often in RL.)

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

(j) **Learning** = Solving a DP problem without using an explicit mathematical model. (This is the principal meaning of the term "learning" in RL. Other meanings are also common.)

(k) **Self-learning** (or self-play in the context of games) = Solving a DP problem using some form of 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** = State aggregation.

(p) **Temporal abstraction** = Time aggregation.

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

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

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

(t) **Experience replay** = Reuse of samples in a simulation process.

(u) **Bellman operator** = DP mapping or operator.

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

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

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

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

(z) **Ground truth** = Empirical evidence or information provided by direct observation.

Some of the preceding terms will be introduced in future chapters; see also the RL textbook [Ber19a]. The reader may then wish to return to this section as an aid in connecting with the relevant RL literature.

1.7.2 Differences in Notation

Unfortunately, the confusion arising from different terminology has been exacerbated by the use of different notations. This book roughly follows the "standard" notation of the Bellman/Pontryagin optimal control era; see e.g., the books by Athans and Falb [AtF66], Bellman [Bel67], and Bryson and Ho [BrH75]. This notation is consistent with the author's other DP books and is the most appropriate for a unified treatment of the subject, which simultaneously addresses discrete and continuous spaces problems.

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) w: stochastic disturbance.

(g) i: discrete state.

(h) $p_{xy}(u)$: transition probability from state x to state y under control u.

(i) α: discount factor in discounted problems.

The x-u-J notation is standard in optimal control textbooks (e.g., the classical books [AtF66] and [BrH75], noted earlier, as well as the more recent books by Stengel [Ste94], Kirk [Kir04], and 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 (unfortunately the more natural symbol "c" has not been used much in place of "g" for the cost per stage). The discrete system notations i and $p_{ij}(u)$ are common in the discrete-state Markov decision problem and operations research literature, where discrete-state problems have been treated extensively [sometimes the alternative notation $p(j \mid i, u)$ is used for the transition probabilities].

The artificial intelligence literature addresses for the most part finite-state Markov decision problems, most frequently the discounted and stochastic shortest path infinite horizon problems. The most commonly used notation is s for state, a for action, $r(s, a, s')$ for reward per stage, $p(s' \mid s, a)$ or $p(s, a, s')$ for transition probability from s to s' under action a, and γ for discount factor. However, this type of notation is not well suited for continuous spaces models, which are of major interest in this book. The reason is that it requires the use of transition probability distributions defined over continuous spaces, and it leads to more complex and less intuitive mathematics. Moreover the transition probability notation is cumbersome for deterministic problems, which involve no probabilistic structure at all.

1.7.3 A Few Words about Machine Learning and Mathematical Optimization

Machine learning and optimization are closely intertwined fields, as they focus on related mathematical models and computational algorithms.† How-

† Both machine learning and optimization are also closely connected with the field of statistical analysis. However, in this section, we will not focus on this connection, as it is less relevant to the content of this book.

ever, they involve different cultures and application contexts, so it is worth reflecting on their similarities and differences.

Machine learning can be broadly categorized into three main types of methods, all of which involve the collection and use of data in some form:

(a) *Supervised learning*: Here a dataset of many input-output pairs is collected. An optimization algorithm is used to create a parametrized function that fits well the data, as well as make accurate predictions on new, unseen data. Supervised learning problems are typically formulated as optimization problems, examples of which we will see in Chapter 3. A common algorithmic approach is to use a gradient-type algorithm to minimize a loss function that measures the difference between the actual outputs of the dataset and the predicted outputs of the parametrized model.

(b) *Unsupervised learning*: Here the dataset is "unlabeled" in the sense that the data are not separated into input and matching output pairs. Unsupervised learning algorithms aim to identify patterns and structures in the data, in applications such as clustering, dimensionality reduction, and density estimation. The main objective is to extract meaningful insights and features from the data. Some unsupervised learning techniques can be approached by DP, but the connection is not strong. Generally speaking, unsupervised learning does not seem to connect well with the types of sequential decision making applications of this book.

(c) *Reinforcement learning*: RL differs in an important way from supervised and unsupervised learning. *It does not use a dataset as a starting point*. Instead, it generates data on-line or off-line as dictated by the needs of the optimization algorithm it uses, be it multistep lookahead minimization, approximate policy iteration and rollout, or approximation in policy space.†

Optimization problems and algorithms on the other hand may or may not involve the collection and use of data. They involve data only in the context of special applications, most of which are related to machine learning. In theoretical terms, optimization problems are categorized in terms of their mathematical structure, which is the primary determinant of the suitability of particular types of methods for their solution. In particular, it is common to distinguish between *static optimization problems* and *dynamic optimization problems*. The latter problems involve sequential decision making, with feedback between decisions, while the former problems involve a single decision. Stochastic problems with perfect or imperfect state observations are dynamic (unless they involve open-loop

† A variant of RL called *offline RL* or *batch RL*, starts from a historical dataset, and does not explore the environment to collect new data.

decision making without the use of any feedback), and they require the use of DP for their optimal solution. Deterministic problems can be formulated as static problems, but they can also be formulated as dynamic problems for reasons of algorithmic expediency. In this case, the decision making process is (sometimes artificially) broken down into stages, as is often done in this book in the context of discrete optimization and other contexts.

Another important categorization of optimization problems is based on whether their search space is *discrete* or is *continuous*. Discrete problems include deterministic problems such as integer and combinatorial optimization problems, and can be addressed by formal methods of integer programming as well as by DP. Also, because they tend to be difficult, they are often addressed (suboptimally) with the use of heuristics. Continuous problems are usually addressed with very different methods, which are based on calculus and convexity, such as Lagrange multiplier theory and duality, and the computational machinery of linear, nonlinear, and convex programming. Special cases of discrete problems that involve the use of graphs, such as matching, transportation, and transhipment, may also be addressed with network optimization methods, which involve the use of continuous optimization approaches that are based on linear programming and duality. Hybrid problems, which involve both continuous and discrete variables, usually require the use of discrete optimization methods, although they can often be addressed with the use of convex duality methods, which fundamentally have a continuous character.

The DP methodology, generally speaking, applies to just about any kind of optimization problem, deterministic or stochastic, static or dynamic, discrete or continuous, *as long as it is formulated as a sequential decision problem*, in the manner described in Sections 1.2-1.4. In terms of its algorithmic structure, DP is very different from other optimization methodologies, particularly the ones that are based on calculus and convexity. Among others, this is evident from the fact that DP can deal with discrete problems as well as continuous, and does not address issues of local minima (it aims exclusively at global minima).

Notice a qualitative difference between optimization and machine learning: *the former is mostly organized around mathematical structures and the analysis of the foundational issues of the corresponding algorithms, while the latter is mostly organized around how data is collected, used, and analyzed, often with a strong emphasis on statistical issues*. This is an important distinction, which affects profoundly the perspectives of researchers in the two fields.

Relations Between RL and DP Methodologies, and their Applications

In comparing the RL and DP methodologies, we should note that they are fundamentally connected through their corresponding problem formu-

lations: they both involve sequential decision making. Thus any problem addressed by DP can in principle be addressed by RL, and reversely.

However, the RL algorithmic methodology is broader than DP, and includes the use of optimization algorithms of the gradient descent and random search type, simulation-based methodologies, statistical methods of sampling and performance evaluation, and neural network design and training ideas.

Moreover, in the artificial intelligence view of RL, a machine learns through trial and error by interacting with an environment.† In practical terms, this is more or less the same as what DP aims to do, but in RL there is often an emphasis on the presence of uncertainty and exploration of the environment. This is different from DP, which in addition to stochastic problems, it is often applied to deterministic problems that do not involve uncertainty or exploration (adaptive control is the only decision and control problem type, where uncertainty and exploration arise in a significant way). We may also add that RL has brought into the field of sequential decision making a fresh and ambitious spirit that has made possible the solution of problems thought to be well outside the capabilities of DP.

On the other hand, a substantial portion of the decision, control, and optimization community views the RL methodology essentially as an approximate form of DP, which can be applied to difficult problems that are beyond the reach of exact optimization. In the context of this view, there is a lot of interest in using RL methods to address intractable problems, including deterministic discrete/integer optimization, which need not involve data collection, interaction with the environment, uncertainty, and learning.

In terms of applications, DP was originally developed in the 1950s and 1960s as part of the then emerging methodologies of operations research and optimal control. These methodologies are now mature and provide important tools and perspectives, as well as a rich variety of applications, such as robotics, autonomous transportation, and aerospace, which can benefit from the use of RL. Moreover, DP has been used in a broad range of applications in industrial engineering, economics, and finance, so these applications can also benefit from the use of RL methods and perspectives. At the same time, RL and machine learning have ushered opportunities for the application of DP techniques in new domains, such as machine translation, image recognition, knowledge representation, database organization, large language models, and automated planning, where they can have a significant practical impact.

---

† A common description it is that "the machine learns sequentially how to make decisions that maximize a reward signal, based on the feedback received from the environment."

The Use of Mathematics in Optimization and Machine Learning

Let us now discuss some differences between the research cultures of optimization and machine learning, as they pertain to the use of mathematics. In optimization, the emphasis is often on general purpose methods that offer broad and mathematically rigorous performance guarantees, for a wide variety of problems. In particular, it is widely believed that a solid mathematical foundation for a given optimization methodology enhances its reliability and clarifies the boundaries of its applicability. Furthermore, it is recognized that formulating practical problems and matching them to the right algorithms is greatly enhanced by one's understanding of the mathematical structure of the underlying optimization methodology.

Machine learning research includes important lines of analysis that have a strongly mathematical character, particularly relating to theoretical computer science, complexity theory, and statistical analysis. At the same time, in machine learning there are eminently useful algorithmic structures, such as neural networks, large language models, and image generative models, which are not well-understood mathematically and defy to a large extent mathematical analysis.† This can add to a perception that focusing on rigorous mathematics, as opposed to practical implementation, may be a low payoff investment in many practical machine learning contexts.

Moreover, as we have mentioned earlier, the starting point in machine learning is often a type of dataset or a specialized type of training problem (e.g., language translation or image recognition), so what is needed is a method that works well on that dataset or type of problem, and not necessarily on other datasets or problems. Thus specialized approximation architectures, implementation techniques, and heuristics, which perform well for the given problem and dataset type, may be perfectly acceptable in a machine learning context, even if they do not provide rigorous and generally applicable performance guarantees.

In conclusion, both optimization and machine learning use mathematical models and rigorous analysis in important ways, and often overlap in the techniques and tools that they use, as well as in the practical applications that they address. However, depending on the type of problem considered, there may be differences in the emphasis and priority placed on mathematical analysis, insight, and generality versus practical effectiveness and problem-specific efficiency. This is particularly true in certain special-

† As an illustration, the paper by He et al., "Deep Residual Learning for Image Recognition," published in Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition, 2016, has been cited over 162,000 times as of May 2023, and contains only two equations. The famous neural network architecture paper by Vaswani et al., "Attention is all you Need," published in NIPS, 2017, which laid the foundation for GPT, has been cited over 73,000 times as of May 2023, and contains only six equations.

ized contexts, and can lead to some tension, as different fields may not fully appreciate each other's perspective.

1.8 NOTES, SOURCES, AND EXERCISES

We will now summarize this chapter and describe how it can be used flexibly as a foundation for a few different courses. We will also provide a selective overview of the DP and RL literature, and also give a few exercises that have been used in ASU classes.

Chapter Summary

In this chapter, we have aimed to provide an overview of the approximate DP/RL landscape, which can serve as the foundation for a deeper in-class development of other RL topics. In particular, we have described in varying levels of depth the following:

(a) The algorithmic foundation of exact DP in all its major forms: deterministic and stochastic, discrete and continuous, finite and infinite horizon.

(b) Approximation in value space with one-step and multistep lookahead, the workhorse of RL, which underlies its major success stories, including AlphaZero. We contrasted approximation in value space with approximation in policy space, and discussed how the two may be combined.

(c) The important division between off-line training and on-line play in the context of approximation in value space. We highlighted how their synergy can be intuitively explained in terms of Newton's method.

(d) The fundamental methods of policy iteration and rollout, the former being primarily an off-line method, and the latter being primarily a less ambitious on-line method. Both methods and their variants bear close relation to Newton's method and draw their effectiveness from this relation.

(e) Some major models with a broad range of applications, such as discrete optimization, POMDP, multiagent problems, adaptive control, and model predictive control. We delineated their principal characteristics and the major RL implementation issues within their contexts.

(f) The use of function approximation, which has been a recurring theme in our presentation. We have touched upon some of the principal schemes for approximation, e.g., neural networks and feature-based architectures.

One of the principal aims of this chapter was to provide a foundational platform for multiple RL courses that explore at a deeper level various algorithmic methodologies, such as:

(1) Rollout and policy iteration.

(2) Neural networks and other approximation architectures for off-line training.

(3) Aggregation, which can be used for cost function approximation in the context of approximation in value space.

(4) A broader discussion of sequential decision making in contexts involving changing system parameters, sequential estimation, and simultaneous system identification and control.

(5) Stochastic algorithms, such as temporal difference methods and Q-learning, which can be used for off-line policy evaluation in the context of approximate policy iteration.

(6) Sampling methods to collect data for off-line training in the context of cost and policy approximations.

(7) Statistical estimates and efficiency enhancements of various sampling methods used in simulation-based schemes. This includes confidence intervals and computational complexity estimates.

(8) On-line methods for specially structured contexts, including problems of the multi-armed bandit type.

(9) Simulation-based algorithms for approximation in policy space, including policy gradient and random search methods.

(10) A deeper exploration of control system design methodologies such as model predictive control and adaptive control, and their applications in robotics and automated transportation.

In our course we have focused selectively on the methodologies (1)-(4), with a limited coverage of (9) in Section 3.5. In a different course, other choices from the above list may be made, by building on the content of the present chapter.

Notes and Sources for Individual Sections

In the literature review that follows, we will focus primarily on textbooks, research monographs, and broad surveys, which supplement our discussions, express related viewpoints, and collectively provide a guide to the literature. Inevitably our referencing reflects a cultural bias, and an overemphasis on sources that are familiar to the author and are written in a similar style to the present book (including the author's own works). Thus we wish to apologize in advance for the many omissions of important research references that are somewhat outside our own understanding and view of the field.

Sections 1.1-1.4: 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 textbook [Ber17a] provides an extensive discussion of finite horizon exact DP, and its applications to discrete and continuous spaces problems, using a notation and style that is consistent with the one used here. The books by Puterman [Put94] and the author [Ber12] provide detailed (but substantially different) treatments of infinite horizon finite-state stochastic DP problems. The book [Ber12] also covers continuous/infinite state and control spaces problems, including the linear quadratic problems that we have discussed for one-dimensional problems in this chapter. Continuous spaces problems present special analytical and computational challenges, which are at the forefront of research of the RL methodology.

Some of the more complex 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, including partial state information problems. This monograph provides an extensive treatment of these issues. The followup work by Huizhen Yu and the author [YuB15] resolves the special measurability issues that relate to policy iteration, and provides additional analysis relating to value iteration. The second volume of the author's DP book [Ber12], Appendix A, provides an accessible summary introduction of the measure-theoretic framework of the book [BeS78].† In the RL literature, the mathematical difficulties around measurability are usually neglected (as they are in this book), and this is fine because they do not play an important role in applications. Moreover, measurability issues do not arise for problems involving finite or countably infinite state and control spaces. We note, however, that there are quite a few published works in RL as

† The rigorous mathematical theory of stochastic optimal control, including the development of an appropriate measure-theoretic framework, dates to the 60s and 70s, with the work of Blackwell and other great mathematicians. It culminated in the monograph [BeS78], which provides the now "standard" framework, based on the formalism of Borel spaces, lower semianalytic functions, and universally measurable policies. This development involves daunting mathematical complications, which stem, among others, from Blackwell's observation that when a Borel measurable function $F(x,u)$, of the two variables x and u, is minimized with respect to u, the resulting function $G(x) = \min_u F(x,u)$ need not be Borel measurable (it belongs to the broader class of lower semianalytic functions; see [BeS78]). Moreover, even if the minimum is attained by several functions/policies μ, i.e., $G(x) = F\big(x,\mu(x)\big)$ for all x, it is possible that none of these μ is Borel measurable (however, there does exist a minimizing policy that belongs to the broader class of universally measurable policies). Thus, starting with a Borel measurability framework for cost functions and policies, we quickly get outside that framework when executing DP algorithms, such as value and policy iteration. The broader framework of universal measurability is required to correct this deficiency, in the absence of additional (fairly strong) assumptions.

well as exact DP, which purport to address measurability issues with a mathematical narrative that is either confusing or plain incorrect.

The third edition of the author's abstract DP monograph [Ber22b], expands on the original 2013 first edition, and aims at a unified development of the core theory and algorithms of total cost sequential decision problems. It addresses simultaneously stochastic, minimax, game, risk-sensitive, and other DP problems, through the use of abstract DP operators (or Bellman operators as we call them here). The idea is to gain insight through abstraction. In particular, the structure of a DP model is encoded in its abstract Bellman operator, which serves as the "mathematical signature" of the model. Thus, characteristics of this operator (such as monotonicity and contraction) largely determine the analytical results and computational algorithms that can be applied to that model. Abstract DP ideas are also useful for visualizations and interpretations of RL methods using the Newton method formalism that we have discussed somewhat briefly in this book in the context of linear quadratic problems.

Approximation in value space, rollout, and policy iteration are the principal subjects of this book.† These are very powerful and general techniques: they can be applied to deterministic and stochastic problems, finite and infinite horizon problems, discrete and continuous spaces problems, and mixtures thereof. Moreover, rollout is reliable, easy to implement, and can be used in conjunction with on-line replanning. It is also compatible with new and exciting technologies such as transformer networks and large language models.

As we have noted, rollout with a given base policy is simply the first iteration of the policy iteration algorithm starting from the base policy. Truncated rollout can be interpreted as an "optimistic" form of a single policy iteration, whereby a policy is evaluated inexactly, by using a limited number of value iterations; see the books [Ber20a], [Ber22a].‡

† The name "rollout" (also called "policy rollout") was introduced by Tesauro and Galperin [TeG96] in the context of rolling the dice in the game of backgammon. In Tesauro's proposal, a given backgammon position is evaluated by "rolling out" many games starting from that position to the end of the game. To quote from the paper [TeG96]: "In backgammon parlance, the expected value of a position is known as the "equity" of the position, and estimating the equity by Monte-Carlo sampling is known as performing a "rollout." This involves playing the position out to completion many times with different random dice sequences, using a fixed policy to make move decisions for both sides."

‡ Truncated rollout was also proposed in the context of backgammon in the paper [TeG96]. To quote from this paper: "Using large multi-layer networks to do full rollouts is not feasible for real-time move decisions, since the large networks are at least a factor of 100 slower than the linear evaluators described previously. We have therefore investigated an alternative Monte-Carlo algorithm, using so-called "truncated rollouts." In this technique trials are not played out

Policy iteration, which will be viewed here as the repeated use of rollout, is more ambitious and challenging than rollout. It requires off-line training, possibly in conjunction with the use of neural networks. Together with its neural network and distributed implementations, it will be discussed in more detail later. Note that rollout does not require any off-line training, once the base policy is available; this is its principal advantage over policy iteration.

Section 1.5: There is a vast literature on linear quadratic problems. The connection of policy iteration with Newton's method within this context and its quadratic convergence rate was first derived by Kleinman [Kle68] for continuous-time linear quadratic problems (the corresponding discrete-time result was given by Hewer [Hew71]). For followup work, which relates to policy iteration with approximations, see Feitzinger, Hylla, and Sachs [FHS09], and Hylla [Hyl11].

The general relation of approximation in value space with Newton's method, beyond policy iteration, and its connections with MPC and adaptive control was first presented in the author's book [Ber20a], the papers [Ber21b], [Ber22c], and in the book [Ber22a], which contains an extensive discussion. This relation provides the starting point for an in-depth understanding of the synergy between the off-line training and the on-line play components of the approximation in value space architecture, including the role of multistep lookahead in enhancing the starting point of the Newton step. The monograph [Ber22a] also provides convergence analysis of variants of Newton's method applied to the solution of nondifferentiable fixed point problems.

Note that in approximation in value space, we are applying Newton's method to the solution of a system of equations (the Bellman equation). This context has no connection with the "gradient descent" methods that are popular for the solution of special types of optimization problems in RL, arising for example in neural network training problems (see Chapter 3). In particular, there are no gradient descent methods that can be used for the solution of systems of equations such as the Bellman equation. There are, however, "first order" deterministic algorithms such as the Gauss-Seidel and Jacobi methods (and stochastic asynchronous extensions) that can

---

to completion, but instead only a few steps in the simulation are taken, and the neural net's equity estimate of the final position reached is used instead of the actual outcome. The truncated rollout algorithm requires much less CPU time, due to two factors: First, there are potentially many fewer steps per trial. Second, there is much less variance per trial, since only a few random steps are taken and a real-valued estimate is recorded, rather than many random steps and an integer final outcome. These two factors combine to give at least an order of magnitude speed-up compared to full rollouts, while still giving a large error reduction relative to the base player." Analysis and computational experience with truncated rollout since 1996 are consistent with the preceding assessment.

be applied to the solution of systems of equations with special structure, including Bellman equations. Such methods include various Q-learning algorithms, which are discussed in the neuro-dynamic programming book by Bertsekas and Tsitsiklis [BeT89], as well as the recent book by Meyn [Mey22]. They are generally far slower than Newton's method, and have limited value in on-line play contexts.

Section 1.6: Many applications of DP are discussed in the 1st volume of the author's DP book [Ber17a]. This book also covers a broad variety of state augmentation and problem reformulation techniques, including the mathematics of how problems with imperfect state information can be transformed to perfect state information problems. In Section 1.6 we have aimed to provide an overview, with an emphasis on the use of approximations. In what follows we provide some related historical notes.

Multiagent problems: This subject has a long history (Marschak [Mar55], Radner [Rad62], Witsenhausen [Wit68], [Wit71a], [Wit71b]), and was researched extensively in the 70s; see the review paper by Ho [Ho80] and the references cited there. The names used for the field at that time were *team theory* and *decentralized control*. For a sampling of subsequent works in team theory and multiagent optimization, we refer to the papers by Krainak, Speyer, and Marcus [KLM82a], [KLM82b], and de Waal and van Schuppen [WaS00]. For more recent works, see Nayyar, Mahajan, and Teneketzis [NMT13], Nayyar and Teneketzis [NaT19], Li et al. [LTZ19], Qu and Li [QuL19], Gupta [Gup20], the book by Zoppoli, Sanguineti, Gnecco, and Parisini [ZSG20], and the references quoted there. In addition to the aforementioned works, surveys of multiagent sequential decision making from an RL perspective were given by Busoniu, Babuska, and De Schutter [BBD08], [BBD10b].

We note that the term "multiagent" has been used with several different meanings in the literature. For example, some authors place emphasis on the case where the agents do not have common information when selecting their decisions. This gives rise to sequential decision problems with "nonclassical information patterns," which can be very complex, partly because they cannot be addressed by exact DP. Other authors adopt as their starting point a problem where the agents are "weakly" coupled through the system equation, the cost function, or the constraints, and consider methods whereby the weak coupling is exploited to address the problem through (suboptimal) decoupled computations.

Agent-by-agent minimization in multiagent approximation in value space and rollout was proposed in the author's paper [Ber19c], which also discusses extensions to infinite horizon policy iteration algorithms, and explores connections with the concept of person-by-person optimality from team theory; see also the textbook [Ber20a], the papers [Ber19d], [Ber20b]. The papers by Bhattacharya et al. [BKB20], Garces et al. [GBG22], and Weber at al. [WGP23] present computational studies with challenging prob-

lems, where several of the multiagent algorithmic ideas were adapted, tested, and validated. These papers consider large-scale multi-robot and vehicle routing problems, involving partial state information, and explore some of the attendant implementation issues, including autonomous multiagent rollout, through the use of policy neural networks and other precomputed signaling policies. The papers also provide a comparison with alternative approaches to multiagent problems, some of which are based on policy gradient methods.

A different type of distributed computation and multiagent optimization, whereby each agent has a partial/local model of the system within part of the state space and relies on aggregate information from other agents to execute a DP computation is proposed in the author's DP book [Ber12], Section 6.5.4; see also Section 3.5.8 of the present book.

Adaptive control: The research on adaptive control has a long history and its literature is very extensive; see the books by Aström and Wittenmark [AsW94], Aström and Hagglund [AsH06], Bodson [Bod20], Goodwin and Sin [GoS84], Ioannou and Sun [IoS96], Jiang and Jiang [JiJ17], Krstic, Kanellakopoulos, and Kokotovic [KKK95], Kumar and Varaiya [KuV86], Liu, et al. [LWW17], Lavretsky and Wise [LaW13], Narendra and Annaswamy [NaA12], Sastry and Bodson [SaB11], Slotine and Li [SlL91], and Vrabie, Vamvoudakis, and Lewis [VVL13]. These books describe a vast array of methods spanning 60 years, and ranging from adaptive and PID model-free approaches, to simultaneous or sequential control and identification, to time series models, to extremum-seeking methods, to simulation-based RL techniques, etc.

The ideas of PID control have been applied widely to adaptive and robust control contexts, and have a long history; see the books by Aström and Hagglund [AsH95], [AsH06], which provide many references. According to Wikipedia, "a formal control law for what we now call PID or three-term control was first developed using theoretical analysis, by Russian American engineer Nicolas Minorsky" in 1922 [Min22].

The DP framework for adaptive control was introduced in a series of papers by Feldbaum, starting in 1960 with [Fel60], under the name *dual control theory*. These papers emphasized the division of effort between system estimation and control, now more commonly referred to as the *exploration-exploitation tradeoff*. In the last paper of the series [Fel63], Feldbaum prophetically concluded as follows: "At the present time, the most important problem for the immediate future is the development of approximate solution methods for dual control theory problems, the formulation of sub-optimal strategies, the determination of the numerical value of risk in quasi-optimal systems and its comparison with the value of risk in existing systems."

The research on problems involving unknown models and using data for model identification simultaneously with control was rekindled with the

advent of the artificial intelligence side of RL and its focus on the active exploration of the environment. Here there is emphasis on "learning from interaction with the environment" [SuB18] through the use of (possibly hidden) Markov decision models, machine learning, and neural networks, in a wide array of methods that are under active development at present. This is more or less the same as the classical problems of dual and adaptive control that have been discussed since the 60s from a control theory perspective.

The formulation of adaptive and dual control problems as POMDP (cf. Section 2.11) is classical. The use of rollout within this context was first suggested in the author's book [Ber22a], Section 6.7.

Model predictive control: The literature on the theory and applications of MPC is voluminous. Some early widely cited papers are Clarke, Mohtadi, and Tuffs [CMT87a], [CMT87b], and Keerthi and Gilbert [KeG88]. For surveys, which give many of the early references, see Morari and Lee [MoL99], Mayne et al. [MRR00], and Findeisen et al. [FIA03], and for a more recent review, see Mayne [May14]. Textbooks on MPC include Maciejowski [Mac02], Goodwin, Seron, and De Dona [GSD06], Camacho and Bordons [CaB07], Kouvaritakis and Cannon [KoC16], Borrelli, Bemporad, and Morari [BBM17], and Rawlings, Mayne, and Diehl [RMD17]. The connections between MPC, approximation in value space and rollout were discussed in the author's surveys [Ber05a] and [Ber24].

Reinforcement Learning Sources

The first DP/RL books were written 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 is culturally different and 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], [Ber17a] for a broader discussion of some of the topics of this book, including algorithmic convergence issues and additional DP models, such as those based on average cost and semi-Markov problem optimization. Note that both of these books deal with finite-state Markovian decision models and use a transition probability notation, as they do not address continuous spaces problems, which are one of the major focal points of this book.

More recent books are by Gosavi [Gos15] (a much expanded 2nd edition of his 2003 monograph), which emphasizes simulation-based optimization and RL algorithms, Cao [Cao07], which focuses on a sensitivity approach to simulation-based methods, Chang, Fu, Hu, and Marcus [CFH13] (a 2nd edition of their 2007 monograph), which emphasizes finite-horizon/multistep lookahead schemes and adaptive sampling, Busoniu, Babuska, De Schutter, and Ernst [BBD10a], which focuses on function

approximation methods for continuous space systems and includes a discussion of random search methods, Szepesvari [Sze10], which is a short monograph that selectively treats some of the major RL algorithms such as temporal differences, armed bandit methods, and Q-learning, Powell [Pow11], which emphasizes resource allocation and operations research applications, Powell and Ryzhov [PoR12], which focuses on specialized topics in learning and Bayesian optimization, Vrabie, Vamvoudakis, and Lewis [VVL13], which discusses neural network-based methods and on-line adaptive control, Kochenderfer et al. [KAC15], which selectively discusses applications and approximations in DP and the treatment of uncertainty, Jiang and Jiang [JiJ17], which addresses adaptive control and robustness issues within an approximate DP framework, Liu, Wei, Wang, Yang, and Li [LWW17], which deals with forms of adaptive dynamic programming, and topics in both RL and optimal control, and Zoppoli, Sanguineti, Gnecco, and Parisini [ZSG20], which addresses neural network approximations in optimal control as well as multiagent/team problems with nonclassical information patterns. The book by Meyn [Mey22] focuses on the connections of RL and optimal control, similar to the present book, but is more mathematically oriented, and treats stochastic problems and algorithms in far more detail.

There are also several books that, while not exclusively focused on DP and/or RL, touch upon several of the topics of the present book. 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 discusses some of the popular approximate DP/RL algorithms. The book by Haykin [Hay08] discusses approximate DP in the broader context of neural network-related subjects. The book by Krishnamurthy [Kri16] focuses on partial state information problems, with a discussion of both exact DP, and approximate DP/RL methods. The textbooks by Kouvaritakis and Cannon [KoC16], Borrelli, Bemporad, and Morari [BBM17], and Rawlings, Mayne, and Diehl [RMD17] collectively provide a comprehensive view of the MPC methodology. The book by Lattimore and Szepesvari [LaS20] is focused on multiarmed bandit methods. The book by Brandimarte [Bra21] is a tutorial introduction to DP/RL that emphasizes operations research applications and includes MATLAB codes. The book by Hardt and Recht [HaR21] focuses on broader subjects of machine learning but covers selectively approximate DP and RL topics as well.

The present book is similar in style, terminology, and notation to the author's recent RL textbooks [Ber19a], [Ber20a], [Ber22a], and the 3rd edition of the abstract DP monograph [Ber22b], which collectively provide a fairly comprehensive and more mathematical account of the subject. In particular, the 2019 RL textbook includes a broader coverage of approximation in value space methods, including certainty equivalent control and

aggregation methods. It also addresses approximation in policy space in greater detail than the present book. The 2020 book focuses more closely on rollout, policy iteration, and multiagent problems. The 2022 book focuses on the connection of approximation in value space with Newton's method, relying on analysis first provided in the book [Ber20a] and the paper [Ber22c]. The abstract DP monograph [Ber22b] (a 3rd edition of the original 2013 1st edition) is an advanced treatment of exact DP, which provides the mathematical framework of Bellman operators that are central for some of the Newton method visualizations presented in the present book and in the books [Ber20a], [Ber22a].

In addition to textbooks, there are many surveys and short research monographs relating to our subject, which are rapidly multiplying in number. 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 RL). The volume by White and Sofge [WhS92] also contains surveys describing early work in the field.

Several overview papers in the volume by Si, Barto, Powell, and Wunsch [SBP04] describe some approximation methods that we will not be covering in much detail in this book: linear programming 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]). Updated accounts of these and other related topics are given in the survey collections by Lewis, Liu, and Lendaris [LLL08], and Lewis and Liu [LeL13].

Recent extended 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), Szepesvari [Sze10] (which discusses 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), Geffner and Bonet [GeB13] (which deals with problems in search and automated planning), Schmidhuber [Sch15], Arulkumaran et al. [ADB17], Li [Li17], Busoniu et al. [BDT18], and Caterini and Chang [CaC18] (which deal with reinforcement learning schemes that are based on the use of deep neural networks), Recht [Rec18a] (which discusses continuous spaces optimal control), and the author's [Ber05a] (which focuses on rollout algorithms and MPC), [Ber11a] (which focuses on approximate policy iteration), [Ber18a] (which focuses on aggregation methods), [Ber20b] (which focuses on multiagent problems), and [Ber24] (which focuses on the relations between RL and MPC).

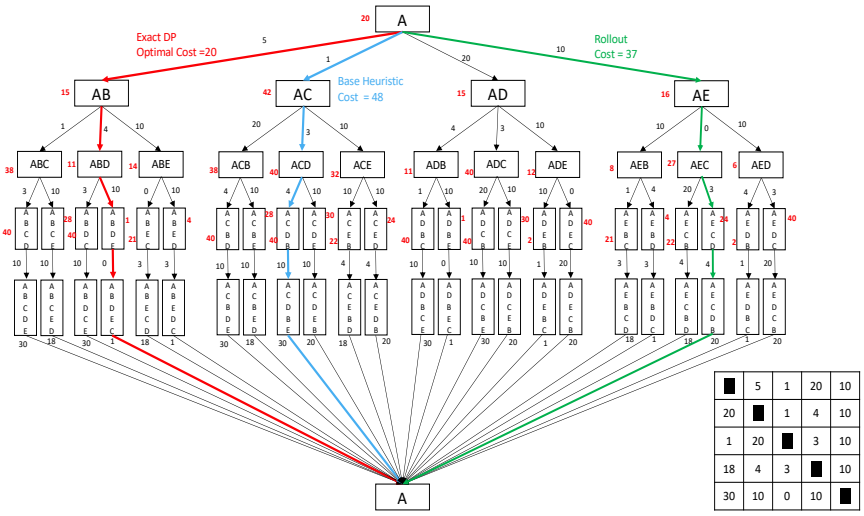

Figure 1.8.1 Solution of parts (a), (b), and (c) of Exercise 1.1. A 5-city traveling salesman problem illustration of rollout with the nearest neighbor base heuristic.

E X E R C I S E S

1.1 (Computational Exercise - Traveling Salesman Problem)

Consider a modified version of the four-city traveling salesman problem of Example 1.2.3, where there is a fifth city E. The intercity travel costs are shown in Fig. 1.8.1, which also gives the solutions to parts (a), (b), and (c).

(a) Use exact DP with starting city A to verify that the optimal tour is ABDECA with cost 20.

(b) Verify that the nearest neighbor heuristic starting with city A generates the tour ACDBEA with cost 48.

(c) Apply rollout with one-step lookahead minimization, using as base heuristic the nearest neighbor heuristic. Show that it generates the tour AECDBA with cost 37.
Illustration of the algorithm: At city A, the nearest neighbor heuristic generates the tour ACDBEA with cost 48, as per part (b). At city A, the rollout algorithm considers the four options of moving to cities B, C, D, E, or equivalently to states AB, AC, AD, AE, and it computes the nearest neighbor-generated tours corresponding to each of these states. These tours are ABCDEA with cost 49, ACDBEA with cost 48, ADCEBA with cost

63, and AECDBA with cost 37. The tour AECDBA has the least cost, so the rollout algorithm moves to city E or equivalently to state AE.

At AE, the rollout algorithm considers the three options of moving to cities B, C, D, or equivalently to states AEB, AEC, AED, and it computes the nearest neighbor-generated tours corresponding to each of these states. These tours are AEBCDA with cost 42, AECDBA with cost 37, AEDCBA with cost 63. The tour AECDBA has the least cost, so the rollout algorithm moves to city C or equivalently to state AEC.

At AEC, the rollout algorithm considers the two options of moving to cities B, D, and compares the nearest neighbor-generated tours corresponding to each of these. These tours are AECBDA with cost 52 and AECDBA with cost 37. The tour AECDBA has the least cost, so the rollout algorithm moves to city D or equivalently to state AECD. Then the rollout algorithm has only one option and generates the tour AECDBA with cost 37.

(d) Apply rollout with two-step lookahead minimization, using as base heuristic the nearest neighbor heuristic. This rollout algorithm operates as follows. For $k = 1, 2, 3$, it starts with a k-city partial tour, it generates every possible two-city addition to this tour, uses the nearest neighbor heuristic to complete the tour, and selects as next city to add to the k-city partial tour the city that corresponds to the best tour thus obtained (only one city is added to the current tour at each step of the algorithm, not two). Show that this algorithm generates the optimal tour.

(e) Estimate roughly the complexity of the computations in parts (a), (b), (c), and (d), assuming a generic N-city traveling salesman problem. *Answer*: The exact DP algorithm requires $O(N^N)$ computation, since there are

$$(N-1)+(N-1)(N-2)+\cdots+(N-1)(N-2)\cdots 2+(N-1)(N-2)\cdots 2\cdot 1$$

arcs in the DP graph to consider, and this number can be estimated as $O(N^N)$. The nearest neighbor heuristic that starts at city A performs $O(N)$ comparisons at each of N stages, so it requires $O(N^2)$ computation. The rollout algorithm at stage k runs the nearest neighbor heuristic $N-k$ times, so it must run the heuristic $O(N^2)$ times for a total computation of $O(N^4)$. Thus the rollout algorithm's complexity involves a low order polynomial increase over the complexity of the base heuristic, something that is generally true for practical discrete optimization problems. Note that even though this may represent a substantial increase in computation over the base heuristic, it is a potentially enormous improvement over the complexity of the exact DP algorithm.

1.2 (Computational Exercise - Linear Quadratic Problem)

In this problem we focus on the one-dimensional linear quadratic problem of Section 1.5 and the interpretation of approximation space as a Newton step for solving the Riccati equation. Consider the undiscounted linear quadratic problem with parameters $a = 2$, $b = 1$, $q = 1$, $r = 5$.

(a) Plot and solve graphically the Riccati equation as in Fig. 1.5.1.

(b) Plot and solve graphically the Riccati equation corresponding to the linear policy $\mu(x) = -(3/2)x$.

(c) Plot graphically the numerical solution Riccati equation by value iteration as in Fig. 1.5.3 using a starting point $K_0 < K^*$ and a starting point $K_0 > K^*$.

(d) Interpret graphically approximation in value space with one-step, two-step, and three-step lookahead as a Newton step in the manner of Figs. 1.5.7 and 1.5.9. Use cost function approximations $\tilde{K}x^2$ with $\tilde{K} < K^*$ and $\tilde{K} > K^*$. What is the region of stability, i.e., the set of $\tilde{K}$ for which approximation in value space produces a stable policy under one-step, two-step, and three-step lookahead.

(e) Plot the performance error $|K_{\tilde{\mu}} - K^*|$ as a function of $|\tilde{K} - K^*|$ for one-step, two-step, and three-step lookahead approximation in value space.

(f) Plot graphical interpretations of rollout and truncated rollout in the manner of Figs. 1.5.10 and 1.5.11 using a stable starting linear policy of your choice.

1.3 (Computational Exercise - Spiders and Flies)

Consider the spiders and flies problem of Example 1.6.5 with two differences: the five flies stay still (rather than moving randomly), and there are only two spiders, both of which start at the fourth square from the right at the top row of the grid of Fig. 1.6.10. The base policy is to move each spider one square towards its nearest fly, with distance measured by the Manhattan metric, and with preference given to a horizontal direction over a vertical direction in case of a tie. Apply the multiagent rollout algorithm of Section 1.6.5, and compare its performance with the one of the ordinary rollout algorithm, and with the one of the base policy. This problem is also discussed in Section 2.9.

1.4 (Computational Exercise - Exercising an Option)

This exercise deals with a computational comparison of the optimal policy, a heuristic policy, and on-line approximation in value space using the heuristic policy, in the context of a problem that involves the timing of the sale of a stock.

An investor has the option to sell a given amount of stock at any one of N time periods. The initial price of the stock is an integer x_0. The price x_k, if it is positive and it is less than a given positive integer value $\bar{x}$, it evolves according to

$$x_{k+1} = \begin{cases} x_k + 1 & \text{with probability } p^+, \\ x_k & \text{with probability } 1 - p^+ - p^-, \\ x_k - 1 & \text{with probability } p^-, \end{cases}$$

where p^+ and p^- have known values with

$$0 < p^- \leq p^+, \qquad p^+ + p^- < 1.$$

If $x_k = 0$, then x_{k+1} moves to 1 with probability p^+, and stays unchanged at 0 with probability $1 - p^+$. If $x_k = \bar{x}$, then x_{k+1} moves to $\bar{x} - 1$ with probability p^-, and stays unchanged at $\bar{x}$ with probability $1 - p^-$.

At each period $k = 0, \ldots, N-1$ for which the stock has not yet been sold, the investor (with knowledge of the current price x_k), can either sell the stock at the current price x_k or postpone the sale for a future period. If the stock has not been sold at any of the periods $k = 0, \ldots, N-1$, it must be sold at period N at price x_N. The investor wants to maximize the expected value of the sale. For the following computations, use reasonable values of your choice for N, p^+, p^-, $\bar{x}$, and x_0 (you should choose x_0 between 0 and $\bar{x}$). You are encouraged to experiment with different sets of values. A set of values that you may try first is

$$N = 14, \quad x_0 = 3, \quad \bar{x} = 7, \quad p^+ = p^- = 0.25.$$

(a) Formulate the problem as a finite horizon DP problem by identifying the state, control, and disturbance spaces, the system equation, the cost function, and the probability distribution of the disturbance. Write the corresponding exact DP algorithm, and use it to compute the optimal policy and the optimal cost as a function of x_0.
Solution: The optimal reward-to-go is generated by the following DP algorithm:

$$J_N^*(x_N) = x_N, \tag{1.96}$$

and for $k = 0, \ldots, N-1$, if $x_k = 0$, then

$$J_k^*(0) = p^+ J_{k+1}^*(1) + (1 - p^+) J_{k+1}^*(0), \tag{1.97}$$

if $x_k = \bar{x}$, then

$$J_k^*(\bar{x}) = \bar{x}, \tag{1.98}$$

(since the price cannot go higher than $\bar{x}$, once at $\bar{x}$, but can go lower), and if $0 < x_k < \bar{x}$, then

$$J_k^*(x_k) = \max\left\{x_k,\, p^+ J_{k+1}^*(x_k{+}1) + (1{-}p^+{-}p^-) J_{k+1}^*(x_k) + p^- J_{k+1}^*(x_k{-}1)\right\}. \tag{1.99}$$

The optimal policy is to sell at $x_k = 1, \ldots, \bar{x}-1$, if x_k attains the maximum in the above equation, and not to sell otherwise. When $x_k = 0$, it is optimal not to sell, while when $x_k = \bar{x}$, it is optimal to sell.

The values of $J_k^*(x_k)$ and the optimal policy are tabulated as shown in Fig. 1.8.2. For this figure, all the calculations are done for the following special case:

$$N = 10, \quad x_0 = 2, \quad \bar{x} = 10, \quad p^+ = p^- = 0.25.$$

These values are also used for parts (b) and (c). However, you are asked to solve the problem for different values as noted earlier. Note that for the problem to have an interesting solution, the problem data must be chosen so that the problem's policies are materially affected by the presence of the upper and lower bounds on the price x_k. As an example consider the case where

$$N = 10, \quad x_0 = 20, \quad \bar{x} = 40, \quad p^+ = p^- = 0.25.$$

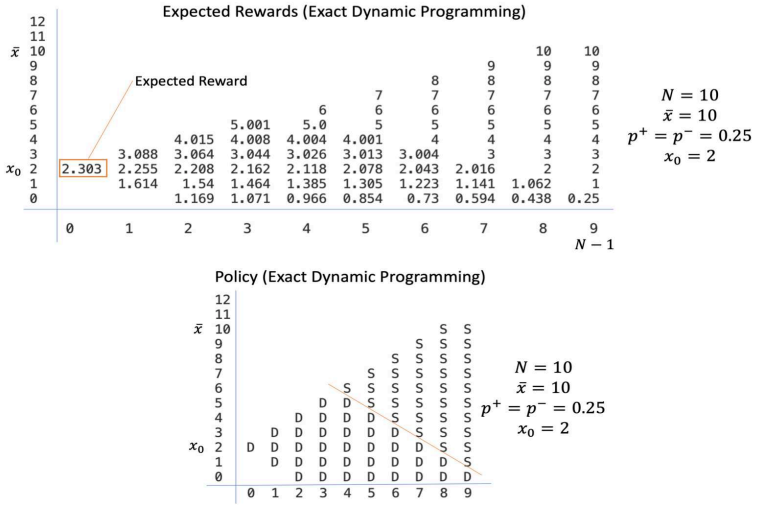

Figure 1.8.2 Table of values of optimal reward-to-go, obtained by exact DP, and corresponding optimal policy [cf. the algorithm (1.96)-(1.99). Only the states x_k that are reachable from x_0 at time k are considered (this is the state space for time k).

Then the bounds $0 \leq x_k$ and $x_k \leq \bar{x}$ never become "active," and it can be verified that the optimal expected reward is $J^*(x_0) = x_0$, while all policies are optimal and attain this optimal expected reward.

(b) Suppose the investor adopts a heuristic, referred to as base heuristic, whereby he/she sells the stock if its price is greater or equal to βx_0, where β is some number with $\beta > 1$. Write an exact DP algorithm to compute the expected value of the sale under this heuristic.
Solution: The reward-to-go for the base heuristic starting from state x_k, denoted $J_k^{x_k}(x_k)$, can be generated by the following (exact) DP algorithm. (Note here the use of superscript x_k in the quantities $J_n^{x_k}(x_n)$ computed by the algorithm. The reason is that the computed values $J_n^{x_k}(x_n)$ depend on x_k, which incidentally implies that base heuristic is not sequentially consistent, as defined later in Section 2.3.2 of this book.) The algorithm is given by

$$J_N^{x_k}(x_N) = x_N, \tag{1.100}$$

and for $n = k, \ldots, N-1$, if $0 < x_n < \beta x_k$, then

$$J_n^{x_k}(x_n) = p^+ J_{n+1}^{x_k}(x_n + 1) + (1 - p^+ - p^-) J_{n+1}^{x_k}(x_n) + p^- J_{n+1}^{x_k}(x_n - 1), \tag{1.101}$$

if $x_n = 0$, then

$$J_n^{x_k}(0) = p^+ J_{n+1}^{x_k}(1) + (1 - p^+) J_{n+1}^{x_k}(0), \tag{1.102}$$

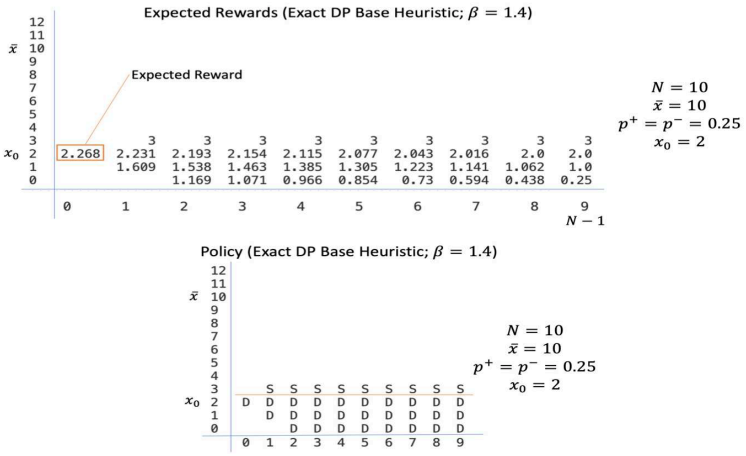

Figure 1.8.3 Table of rewards-to-go for the base policy with $\beta = 1.4$, starting from x_0 [cf. the algorithm (1.100)-(1.103) for $k = 0$].

and if $x_n \geq \beta x_k$, then

$$J_n^{x_k}(x_n) = x_n. \tag{1.103}$$

The values of $J_k^{x_k}(x_k)$ computed by this algorithm are shown in Fig. 1.8.3, together with the decisions applied by the base heuristic.

While the reward-to-go for the base heuristic starting from state x_k is very simple to compute for our problem, in order to apply the rollout algorithm only the values $J_{k+1}^{x_k+1}(x_k+1)$, $J_{k+1}^{x_k}(x_k)$, and $J_{k+1}^{x_k-1}(x_k-1)$ need to be calculated for each state x_k encountered during on-line operation. Moreover, the base heuristic's reward-to-go $J_k^{x_k}(x_k)$ can also be computed on-line by Monte Carlo simulation for the relevant states x_k. This would be the principal option in a more complicated problem where the exact DP algorithm is too time-consuming.

(c) Apply approximation in value space with one-step lookahead minimization and with function approximation that is based on the heuristic of part (b). In particular, use $\tilde{J}_N(x_N) = x_N$, and for $k = 1, \ldots, N-1$, use $\tilde{J}_k(x_k)$ that is equal to the expected value of the sale when starting at x_k and using the heuristic that sells the stock when its price exceeds βx_k. Use exact DP as well as Monte Carlo simulation to compute/approximate on-line the needed values $\tilde{J}_k(x_k)$. Compare the expected values of sale price computed with the optimal, heuristic, and approximation in value space methods.
Solution: The rollout policy $\tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$ is determined by the base heuristic, where for every possible state x_k, and stage $k = 0, \ldots, N-1$, the rollout decision $\tilde{\mu}_k(x_k)$ is

$$\tilde{\mu}_k(x_k) = \text{sell at } x_k,$$

if

$$p^+ J_{k+1}^{x_k+1}(x_k+1) + (1 - p^+ - p^-) J_{k+1}^{x_k}(x_k) + p^- J_{k+1}^{x_k-1}(x_k-1) \leq x_k,$$

and

$$\tilde{\mu}_k(x_k) = \text{don't sell at } x_k,$$

otherwise. The sell or don't sell decision of the rollout algorithm is made on-line according to the preceding criterion, at each state x_k encountered during on-line operation.

Figure 1.8.4 shows the rollout policy, which is computed by the preceding equations using the rewards-to-go of the base heuristic $J_k^{x_k}(x_k)$, as given in Fig. 1.8.3. Once the rollout policy is computed, the corresponding reward function $\tilde{J}_k(x_k)$ can be calculated similar to the case of the base heuristic. Of course, during on-line operation, the rollout decision need only be computed for the states x_k encountered on-line.

The important observation when comparing Figs. 1.8.3 and 1.8.4 is that the rewards-to-go of the rollout policy are greater or equal to the ones for the base heuristic. In particular, starting from x_0, the rollout policy attains reward 2.269, and the base heuristic attains reward 2.268. The optimal policy attains reward 2.4. The rollout policy reward is slightly closer to the optimal than the base heuristic reward.

The rollout reward-to-go values shown in Fig. 1.8.4 are "exact," and correspond to the favorable case where the heuristic rewards needed at x_k, $J_{k+1}^{x_k+1}(x_k+1)$, $J_{k+1}^{x_k}(x_k)$, and $J_{k+1}^{x_k-1}(x_k-1)$, are computed exactly by DP or by infinite-sample Monte Carlo simulation.

When finite-sample Monte Carlo simulation is used to approximate the needed base heuristic rewards at state x_k, i.e., $J_{k+1}^{x_k+1}(x_k+1)$, $J_{k+1}^{x_k}(x_k)$, and $J_{k+1}^{x_k-1}(x_k-1)$, the performance of the rollout algorithm will be degraded. In particular, by using a computer program to implement rollout with Monte Carlo simulation, it can be shown that when $J_{k+1}^{x_k+1}(x_k+1)$, $J_{k+1}^{x_k}(x_k)$, and $J_{k+1}^{x_k-1}(x_k-1)$ are approximated using a 20-sample Monte-Carlo simulation per reward value, the rollout algorithm achieves reward 2.264 starting from x_0. This reward is evaluated by (almost exact) 400-sample Monte Carlo simulation of the rollout algorithm.

When $J_{k+1}^{x_k+1}(x_k+1)$, $J_{k+1}^{x_k}(x_k)$, and $J_{k+1}^{x_k-1}(x_k-1)$ were approximated using a 200-sample Monte-Carlo simulation per reward value, the rollout algorithm achieves reward 2.273 [as evaluated by (almost exact) 400-sample Monte Carlo simulation of the rollout algorithm]. Thus with 20-sample simulation, the rollout algorithm performs worse than the base heuristic starting from x_0. With the more accurate 200-sample simulation, the rollout algorithm performs better than the base heuristic starting from x_0, and performs nearly as well as the optimal policy (but still somewhat worse than in the case where exact values of the needed base heuristic rewards are used (based on an "infinite" number Monte Carlo samples).

It is worth noting here that the heuristic is not a legitimate policy because at any state x_n is makes a decision that depends on the state x_k where it started. Thus the heuristic's decision at x_n depends not just on x_n, but also on the starting state x_k. However, the rollout algorithm is always an approximation in value space scheme with approximation reward $\tilde{J}_k(x_k)$ defined by the heuristic, and it provides a legitimate policy.

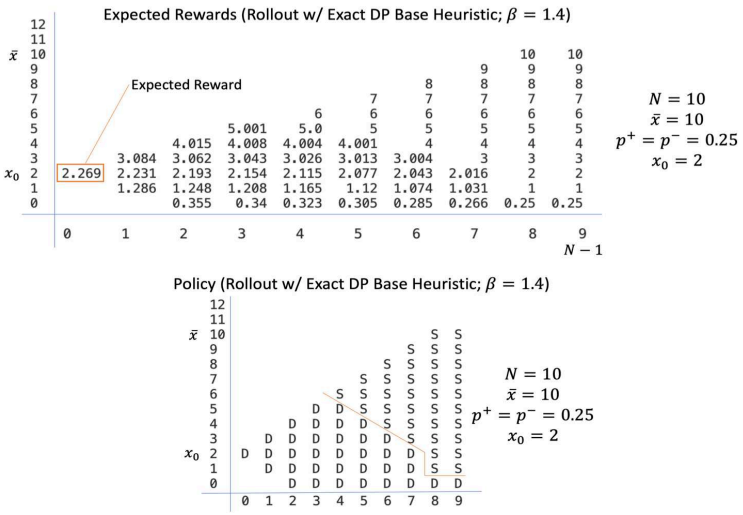

Figure 1.8.4 Table of values of reward-to-go and decisions applied by the rollout policy that corresponds to the base heuristic with $\beta = 1.4$.

(d) Repeat part (c) but with two-step instead of one-step lookahead minimization.

Answer: The implementation is very similar to the one-step lookahead case. The main difference is that at state x_k, the rollout algorithm needs to calculate the base heuristic reward values $J_{k+2}^{x_k+2}(x_k+2)$, $J_{k+2}^{x_k+1}(x_k+1)$, $J_{k+2}^{x_k}(x_k)$, $J_{k+2}^{x_k-1}(x_k-1)$, and $J_{k+2}^{x_k-2}(x_k-2)$. Thus the on-line Monte Carlo simulation work is accordingly increased. Generally the simulation work per stage of the rollout algorithm is proportional to $2\ell+1$, when ℓ-stage lookahead minimization is used, since the number of leafs at the end of the lookahead tree is $2\ell+1$.

1.5 (Computational Exercise - Linear Quadratic Problem)

In a more realistic version of the cruise control system of Example 1.3.1, the system has the form

$$x_{k+1} = ax_k + bu_k + w_k,$$

where the coefficient a satisfies $0 < a \leq 1$, and the disturbance w_k has zero mean and variance σ^2. The cost function has the form

$$(x_N - \bar{x}_N)^2 + \sum_{k=0}^{N-1} \big((x_k - \bar{x}_k)^2 + ru_k^2\big),$$

where $\bar{x}_0, \ldots, \bar{x}_N$ are given nonpositive target values (a velocity profile) that serve to adjust the vehicle's velocity, in order to maintain a safe distance from

the vehicle ahead, etc. In a practical setting, the velocity profile is recalculated by using on-line radar measurements.

Design an experiment to compare the performance of a fixed linear policy π, derived for a fixed nominal velocity profile, and the performance of the algorithm that uses on-line replanning, whereby the optimal policy π^* is recalculated each time the velocity profile changes. Compare with the performance of the rollout policy $\tilde{\pi}$ that uses π as the base policy and on-line replanning.

1.6 (Computational Exercise - Parking Problem)

In reference to Example 1.6.4, a driver aims to park at an inexpensive space on the way to his destination. There are L parking spaces available and a garage at the end. The driver can move in either direction. For example if he is in space i he can either move to $i-1$ with a cost $t-i$, or to $i+1$ with a cost $t+i$, or he can park at a cost $c(i)$ (if the parking space i is free). The only exception is when he arrives at the garage (indicated by index N) and he has to park there at a cost C. Moreover, after the driver visits a parking space he remembers its free/taken status and has an option to return to any parking space he has already visited. However, the driver must park within a given number of stages N, so that the problem has a finite horizon. The initial probability of space i being free is given, and the driver can only observe the free/taken status of a parking only after he/she visits the space. Moreover, the free/taken status of a parking visited so far does not change over time.

Write a program to calculate the optimal solution using exact dynamic programming over a state space that is as small as possible. Try to experiment with different problem data, and try to visualize the optimal cost/policy with suitable graphical plots. Comment on run-time as you increase the number of parking spots L.

1.7 (Newton's Method for Solving the Riccati Equation)

The classical form of Newton's method applied to a scalar equation of the form $H(K) = 0$ takes the form

$$K_{k+1} = K_k - \left(\frac{\partial H(K_k)}{\partial K} \right)^{-1} H(K_k), \tag{1.104}$$

where $\frac{\partial H(K_k)}{\partial K}$ is the derivative of H, evaluated at the current iterate K_k. This exercise shows algebraically (rather than graphically), within the context of linear quadratic problems, that in approximation in value space with quadratic cost approximation, the cost function of the corresponding one-step lookahead policy is the result of a Newton step for solving the Riccati equation. To this end, we will apply Newton's method to the solution of the Riccati Eq. (1.42), which we write in the form $H(K) = 0$, where

$$H(K) = K - \frac{a^2 r K}{r + b^2 K} - q. \tag{1.105}$$

(a) Show that the operation that generates K_L starting from K is a Newton iteration of the form (1.104). In other words, show that for all K that lead to a stable one-step lookahead policy, we have

$$K_L = K - \left(\frac{\partial H(K)}{\partial K}\right)^{-1} H(K), \tag{1.106}$$

where we denote by

$$K_L = \frac{q + rL^2}{1 - (a + bL)^2} \tag{1.107}$$

the quadratic cost coefficient of the one-step lookahead linear policy $\mu(x) = Lx$ corresponding to the cost function approximation $J(x) = Kx^2$:

$$L = -\frac{abK}{r + b^2K}. \tag{1.108}$$

Proof: Our approach for showing the Newton step formula (1.106) is to express each term in this formula in terms of L, and then show that the formula holds as an identity for all L. To this end, we first note from Eq. (1.108) that K can be expressed in terms of L as

$$K = -\frac{rL}{b(a + bL)}. \tag{1.109}$$

Furthermore, by using Eqs. (1.108) and (1.109), $H(K)$ as given in Eq. (1.105) can be expressed in terms of L as follows:

$$H(K) = -\frac{rL}{b(a + bL)} + \frac{arL}{b} - q. \tag{1.110}$$

Moreover, by differentiating the function H of Eq. (1.105), we obtain after a straightforward calculation

$$\frac{\partial H(K)}{\partial K} = 1 - \frac{a^2r^2}{(r + b^2K)^2} = 1 - (a + bL)^2, \tag{1.111}$$

where the second equation follows from Eq. (1.108). Having expressed all the terms in the Newton step formula (1.106) in terms of L through Eqs. (1.107), (1.109), (1.110), and (1.111), we can write this formula in terms of L only as

$$\frac{q + rL^2}{1 - (a + bL)^2} = -\frac{rL}{b(a + bL)} - \frac{1}{1 - (a + bL)^2}\left(-\frac{rL}{b(a + bL)} + \frac{arL}{b} - q\right),$$

or equivalently as

$$q + rL^2 = -\frac{rL\big(1 - (a + bL)^2\big)}{b(a + bL)} + \frac{rL}{b(a + bL)} - \frac{arL}{b} + q.$$

A straightforward calculation now shows that this equation holds as an identity for all L.

(b) What happens when K lies outside the region of stability?

(c) Show that in the case of ℓ-step lookahead, the analog of the quadratic convergence rate estimate has the form

$$|K_{\tilde{L}} - K^*| \leq c \left| F^{\ell-1}(\tilde{K}) - K^* \right|^2,$$

where $F^{\ell-1}(\tilde{K})$ is the result of the $(\ell-1)$-fold application of the mapping F to $\tilde{K}$. Thus a stronger bound for $|K_{\tilde{L}} - K^*|$ is obtained.

1.8 (Region of Stability and the Role of Multistep Lookahead)

In Section 1.5, we discussed the concept of the region of stability in the context of linear quadratic problems. The concept extends to far more general infinite horizon problems (see e.g., the book [Ber22a], Section 3.3). The idea is to call a stationary policy μ unstable if $J_\mu(x) = \infty$ for some states x, and call it stable otherwise. For $\ell \geq 1$, the ℓ-step region of stability is the set of $\tilde{J}$ for which the corresponding ℓ-step lookahead policy is stable.

Generally, the ℓ-step region of stability expands as ℓ increases. Note also that in finite-state discounted problems all policies are stable, so all $\tilde{J}$ belong to the region of stability. However, for SSP this is not so: there are policies, called *improper*, that do not terminate with positive probability for some initial states (see the books [Ber12] and [Ber22b] for extensive discussions). Such policies can be unstable. In the following example the region of instability includes functions that are very close to J^*, even with large ℓ. This example involves small stage costs, a class of problems that pose challenges for approximation in value space; see Section 2.6.

Consider a shortest path problem with a single state 1, plus the termination state t. At state 1 we can either stay at that state at cost $\epsilon > 0$ or move to the state t at cost 1. Thus the optimal policy at state 1 is to move to t, the optimal cost $J^*(1) = 1$, and is the unique solution of Bellman's equation

$$J^*(1) = \min\left\{1,\, \epsilon + J^*(1)\right\}.$$

(In SSP the optimal cost at t is 0 by assumption, and Bellman's equation involves only the costs of the states other than t.)

(a) Show that the one-step region of stability is the set of all $\tilde{J}(1) > 1 - \epsilon$. What happens in the case where $\tilde{J}(1) = 1 - \epsilon$? Show also that the ℓ-step region of stability is the set of all $\tilde{J}(1) > 1 - \ell\epsilon$. *Note*: The ℓ-step region of stability becomes arbitrarily large for sufficiently large ℓ. However, the boundary of the ℓ-step region of stability is arbitrarily close to $J^*(1)$ for sufficiently small ϵ.

(b) What happens in the case where there are additional states $i = 2, \ldots, n$, and for each of these states i there is the option of staying at i at cost ϵ or moving to $i-1$ at cost 0? *Partial answer*: The one-step region of stability consists of all $\tilde{J} = \big(\tilde{J}(n), \ldots, \tilde{J}(1)\big)$ such that $\epsilon + \tilde{J}(i) > \tilde{J}(i-1)$ for all $i \geq 2$ and $\epsilon + \tilde{J}(1) > 1$.

2

Approximation in Value Space - Rollout Algorithms

Contents

In this chapter, we discuss various aspects of approximation in value space and rollout algorithms, focusing primarily on the case where the state and control spaces are finite. In Sections 2.1-2.6, we consider finite horizon deterministic problems, which in addition to arising often in practice, offer some important advantages in the context of RL. In particular, a finite horizon is well suited for the use of rollout, while the deterministic character of the problem eliminates the need for costly on-line Monte Carlo simulation.

An interesting aspect of our methodology for discrete deterministic problems is that it admits extensions that we have not discussed so far. The extensions include multistep lookahead variants, as well as variants that apply to constrained forms of DP, which involve constraints on the entire system trajectory, and also allow the use of heuristic algorithms that are more general than policies within the context of rollout. These variants rely on the problem's deterministic structure, and do not extend to stochastic problems.

Another interesting aspect of finite state deterministic problems is that they can serve as a framework for an important class of commonly encountered discrete optimization problems, including integer programming and combinatorial optimization problems such as scheduling, assignment, routing, etc. This brings to bear the methodology of approximation in value space, rollout, adaptive control, and MPC, and provides effective suboptimal solution methods for these problems.

In Sections 2.7-2.11, we consider various problems that involve stochastic uncertainty. In Section 2.12, we consider minimax problems that involve set membership uncertainty. The present chapter draws heavily on Chapters 2 and 3 of the book [Ber20a], and Chapter 6 of the book [Ber22a]. These books may be consulted for more details and additional examples.

While our focus in this chapter will be on finite horizon problems, our discussion applies to infinite horizon problems as well, because approximation in value space and rollout are essentially finite-stages algorithms, while the nature of the original problem horizon (be it finite or infinite) affects only the terminal cost function approximation. Thus in implementing one-step or multistep approximation in value space, it makes little difference whether the original problem has finite or infinite horizon. At the same time, for conceptual purposes, we can argue that finite horizon problems, even when they involve a nonstationary system and cost per stage, can be transformed to infinite horizon problems, by introducing an artificial cost-free termination state that the system moves into at the end of the horizon; see Sections 1.6.3 and 1.6.4. Through this transformation, the synergy of off-line training and on-line play based on Newton's method is brought to bear, and the insights that we discussed in Chapter 1 in the context of an infinite horizon apply and explain the good performance of our methods in practice.

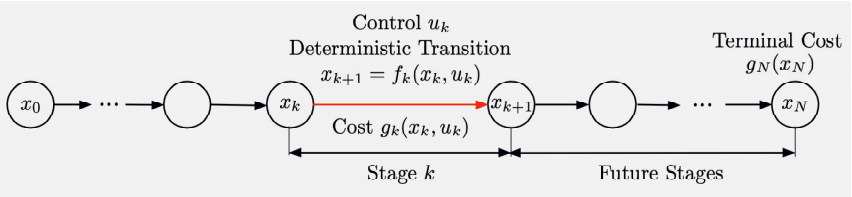

Figure 2.1.1 Illustration of a deterministic N-stage optimal control problem. Starting from state x_k, the next state under control u_k is generated nonrandomly, according to

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

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

2.1 DETERMINISTIC FINITE HORIZON PROBLEMS

We recall from Chapter 1, Section 1.2, that in deterministic finite horizon DP problems, the state is generated nonrandomly over N stages, through a system equation of the form

$$x_{k+1} = f_k(x_k, u_k), \qquad k = 0, 1, \ldots, N-1, \tag{2.1}$$

where k is the time index, and

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

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

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 $k+1$.

The state space X_k and control space U_k are arbitrary sets and may depend on k. Similarly, the system function f_k can be arbitrary and may depend on k. The cost incurred at time k is denoted by $g_k(x_k, u_k)$, and the function g_k 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

$$J(x_0; u_0, \ldots, u_{N-1}) = g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k), \tag{2.2}$$

where $g_N(x_N)$ is a terminal cost incurred at the end of the process. This is a well-defined number, since the control sequence $\{u_0, \ldots, u_{N-1}\}$ together with x_0 determines exactly the state sequence $\{x_1, \ldots, x_N\}$ via the system equation (2.1); see Figure 2.1.1. We want to minimize the cost (2.2) over all sequences $\{u_0, \ldots, u_{N-1}\}$ that satisfy the control constraints, thereby obtaining the optimal value as a function of x_0

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

Notice an important difference from the stochastic case: we optimize over sequences of controls $\{u_0, \ldots, u_{N-1}\}$, rather than over policies that consist of a sequence of functions $\pi = \{\mu_0, \ldots, \mu_{N-1}\}$, where μ_k maps states x_k into controls $u_k = \mu_k(x_k)$, and satisfies the control constraints $\mu_k(x_k) \in U_k(x_k)$ for all x_k. It is well-known that in the presence of stochastic uncertainty, policies are more effective than control sequences, and can result in improved cost. On the other hand for deterministic problems, minimizing over control sequences yields the same optimal cost as over policies, since the cost of any policy starting from a given state determines with certainty the controls applied at that state and the future states, and hence can also be achieved by the corresponding control sequence. This point of view allows more general forms of rollout, which we will discuss in this chapter: instead of using a policy for rollout, we will allow the use of more general heuristics for choosing future controls.

We recall from Chapter 1, Section 1.2, the DP algorithm for finite horizon deterministic problems. It constructs functions

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

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

DP Algorithm for Deterministic Finite Horizon Problems

Start with

$$J_N^*(x_N) = g_N(x_N), \qquad \text{for all } x_N, \tag{2.3}$$

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

$$J_k^*(x_k) = \min_{u_k \in U_k(x_k)} \Big[g_k(x_k, u_k) + J_{k+1}^*\big(f_k(x_k, u_k)\big) \Big], \qquad \text{for all } x_k. \tag{2.4}$$

Note that at stage k, the calculation in Eq. (2.4) 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 was shown in Section 1.2, namely that for all $k = 0, 1, \ldots, N-1$, and all states x_k at time k, we have

$$J_k^*(x_k) = \min_{\substack{u_m \in U_m(x_m) \\ m=k,\ldots,N-1}} J(x_k; u_k, \ldots, u_{N-1}), \tag{2.5}$$

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

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

Thus, $J_k^*(x_k)$ is the optimal cost for an $(N-k)$-stage tail subproblem that starts at state x_k and time k, and ends at time N. Based on this interpretation of $J_k^*(x_k)$, we call it the *optimal cost-to-go* from state x_k at stage k, and refer to J_k^* as the *optimal cost-to-go function* or *optimal cost function* at time k.

We have also discussed in Section 1.2 the construction of an optimal control sequence. Once the functions $J_0^*, \ldots, J_N^*$ have been obtained, we can use a forward algorithm to construct an optimal control sequence $\{u_0^*, \ldots, u_{N-1}^*\}$ and state trajectory $\{x_1^*, \ldots, x_N^*\}$ for a 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

$$u_k^* \in \arg \min_{u_k \in U_k(x_k^*)} \Big[g_k(x_k^*, u_k) + J_{k+1}^*\big(f_k(x_k^*, u_k)\big) \Big], \qquad (2.6)$$

and

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

Note an interesting conceptual division of the optimal control sequence construction: there is *off-line training* to obtain J_k^* by precomputation [cf. the DP Eqs. (2.3)-(2.4)], which is followed by *on-line play* to obtain u_k^* [cf. Eq. (2.6)]. This is analogous to the two algorithmic processes described in Section 1.1 in connection with computer chess and backgammon.

Finite-State Deterministic Problems

For the first five sections of this chapter, we will consider the case where the state and control spaces are discrete and consist of a finite number of

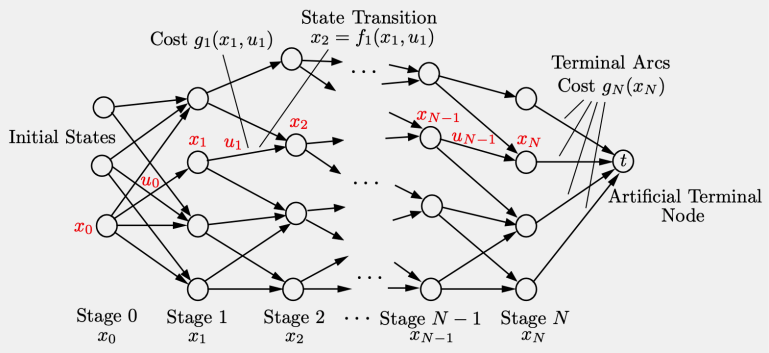

Figure 2.1.2 Illustration of a deterministic finite-state DP problem. Nodes correspond to states x_k. Arcs correspond to state-control pairs (x_k, u_k). An arc (x_k, u_k) has start and end nodes x_k and $x_{k+1} = f_k(x_k, u_k)$, respectively. The cost $g_k(x_k, u_k)$ of the transition is the length of this arc. An artificial terminal node t is connected with an arc of cost $g_N(x_N)$ with each state x_N. The problem is equivalent to finding a shortest path from initial nodes of stage 0 to node t.

elements. As we have noted in Section 1.2, such problems can be described with an acyclic graph specifying for each state x_k the possible transitions to next states x_{k+1}. The nodes of the graph correspond to states x_k and the arcs of the graph correspond to state-control pairs (x_k, u_k). Each arc with start node x_k corresponds to a choice of a single control $u_k \in U_k(x_k)$ and has as end node the next state $f_k(x_k, u_k)$. The cost of an arc (x_k, u_k) is defined as $g_k(x_k, u_k)$; see Fig. 2.1.2. 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 $g_N(x_N)$. The control sequences $\{u_0, \ldots, u_{N-1}\}$ correspond to paths originating at the initial state (a node at stage 0) and terminating at one of the nodes corresponding to the final stage N. With this description it can be seen that *a deterministic finite-state finite-horizon problem is equivalent to finding a minimum-length (or shortest) path from the initial nodes of the graph (stage 0) to the terminal node t*, as we have discussed in Section 1.2.

Shortest path problems arise in a great variety of application domains. While there are quite a few efficient polynomial algorithms for solving them, some practical shortest path problems are extraordinarily difficult because they involve an astronomically large number of nodes. For example deterministic scheduling problems of the type discussed in Example 1.2.1 can be formulated as shortest path problems, but with a number of nodes that grows exponentially with the number of tasks. For such problems neither exact DP nor any other shortest path algorithm can compute an exact optimal solution in practice. In what follows, we will aim to show that suboptimal solution methods, and rollout algorithms in particular, offer a viable alternative.

Many types of search problems involving games and puzzles also ad-

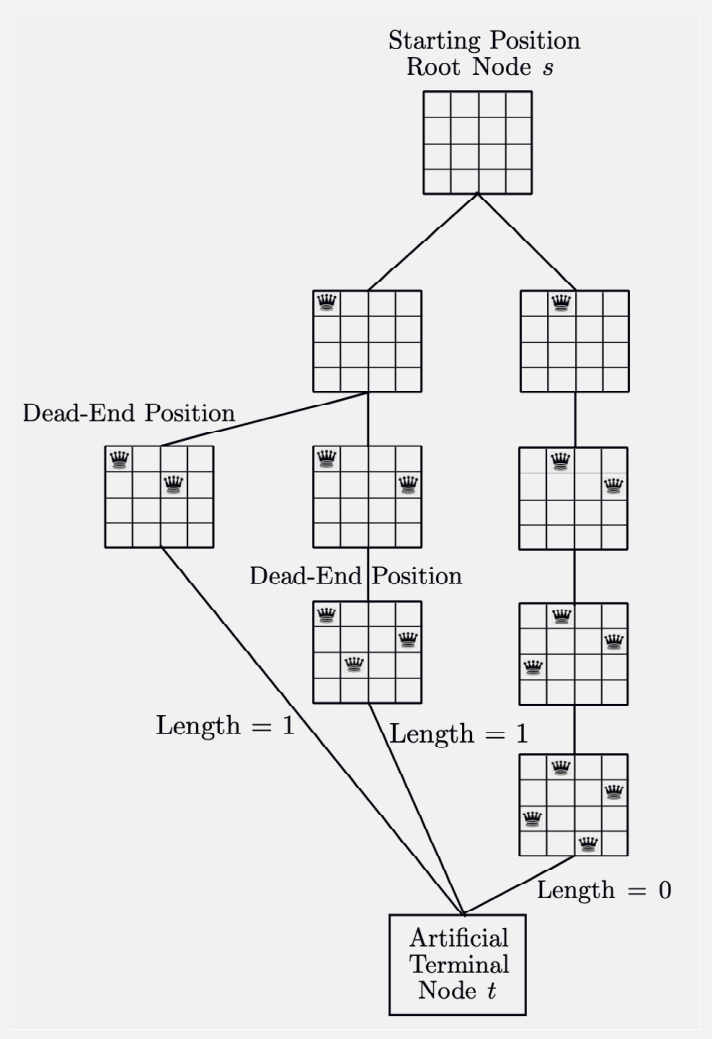

Figure 2.1.3 A finite horizon deterministic DP formulation of the four queens problem. Symmetric positions resulting from placing a queen in one of the rightmost squares in the top row have been ignored. Squares containing a queen have been darkened. All arcs have length zero except for those connecting dead-end positions to the artificial terminal node.

mit in principle exact solution by DP, but have to be solved by suboptimal methods in practice. The following is a characteristic example.

Example 2.1.1 (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. 2.1.3) 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 the N queens problem for all $N \geq 4$ (for $N = 2$ and $N = 3$, clearly there is no solution). Moreover effective (non-DP) search algorithms have been devised for its solution up to very large values of N.

The preceding example illustrates some of the difficulties of applying exact DP to discrete/combinatorial problems with the type of formulation that we have described. The state space typically becomes very large, particularly as k increases. In the preceding example, to start a backward DP algorithm, we need to consider all the possible terminal positions, which are too many when N is large. There is an alternative exact DP algorithm for deterministic problems, which proceeds forwards from the initial state. It is simply the backward DP algorithm applied to an equivalent shortest path problem, derived form one of Fig. 2.1.2 by reversing the directions of all the arcs, and exchanging the roles of the origin and the destination. It will be discussed in Section 2.4; see also [Ber17a], Chapter 2. Still, however, this forward DP algorithm cannot overcome the difficulty with a very large state space.

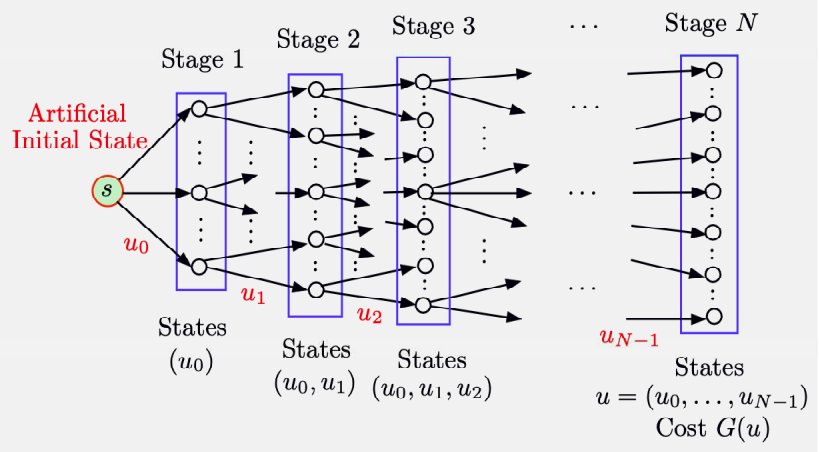

Figure 2.1.4 A DP formulation of the discrete optimization problem of minimizing a cost function $G(u)$ over all u within a finite set U, for the case where u consists of N components (cf. Section 1.6.3).

Discrete Optimization Problems

A major class of deterministic problems that can be formulated as DP problems involves the minimization of a cost function $G(u)$ over all u within a constraint set U. For the purposes of this chapter, we assume that U is finite, although a similar DP formulation is also possible for the more general case of an infinite set U. In Section 1.6.3, we discussed the case where u consists of N components,

$$u = (u_0, \ldots, u_{N-1}),$$

the system equation takes the simple form

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

and there is just a terminal cost $G(x_N) = G(u)$; see Fig. 2.1.4.

The following is a simple but challenging example, which we will further discuss in Section 2.6.

Example 2.1.2 (Constraint Programming)

An interesting special case of the general optimization problem $\min_{u \in U} G(u)$, where $u = (u_0, \ldots, u_{N-1})$, is a *feasibility problem*, where $G(u) \equiv 0$, and the problem reduces to finding a value of u that satisfies the constraint. Generally, the structure of the constraint set U is encoded in a graph representing the problem such as the one of Fig. 2.1.4; cf. Section 1.6.3. This type of feasibility problem is also known as a *constraint programming problem*. The four queens

problem (Example 2.1.1) provides an illustration. Another example is the breakthrough problem to be discussed in Section 2.3.

Constraint programming problems can also be transformed into equivalent unconstrained (or less constrained) problems by using problem-dependent penalty functions that eliminate constraints while quantifying the level of constraint violation. As an illustration, consider the case where the problem is to find a feasible solution of a system of constraints of the form

$$h_k(u_k, u_{k+1}) \leq 0, \qquad k = 0, \ldots, N-1,$$

$$u_k \in U_k, \qquad k = 0, \ldots, N-1.$$

This problem can be transformed into the equivalent DP problem of minimizing

$$\sum_{k=1}^{N} \max\big\{0, h_k(x_k, u_k)\big\},$$

subject to the system equation $x_{k+1} = u_k$, and the control constraints $u_k \in U_k$, $k = 0, \ldots, N-1$. Other penalty functions can also be used, such as a quadratic; see the author's nonlinear programming text [Ber16]. This approach is convenient, but it offers no guarantee that it can find a complete feasible solution $(u_0, \ldots, u_{N-1})$, even if one exists. It simply aims to minimize (suboptimally) a measure of the total constraint violation. However, in the process it may be able to find a complete feasible solution, or an infeasible solution that is adequate for practical purposes.

2.2 APPROXIMATION IN VALUE SPACE - DETERMINISTIC PROBLEMS

The forward optimal control sequence construction of Eq. (2.6) 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. 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 idea of approximation in value space that we discussed in Section 1.2.3. It constructs a suboptimal solution $\{\tilde{u}_0, \ldots, \tilde{u}_{N-1}\}$ in place of the optimal $\{u_0^*, \ldots, u_{N-1}^*\}$, by using $\tilde{J}_k$ in place of J_k^* in the DP procedure (2.6).

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 $k = 1, 2, \ldots, N-1$, set

$$\tilde{u}_k \in \arg\min_{u_k \in U_k(\tilde{x}_k)} \Big[g_k(\tilde{x}_k, u_k) + \tilde{J}_{k+1}\big(f_k(\tilde{x}_k, u_k)\big) \Big], \qquad (2.7)$$

and

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

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 is minimized in approximation in value space [cf. Eq. (2.7)] is known as the (approximate) *Q-factor of* (x_k, u_k). Note that the computation of the suboptimal control (2.7) can be done through the Q-factor minimization

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

This suggests the possibility of using approximate off-line trained Q-factors in place of cost functions in approximation in value space schemes. However, contrary to the cost approximation scheme (2.7) and its multistep counterparts, the performance may be degraded through the errors in the off-line training of the Q-factors (depending on how the training is done).

Exploiting Structure to Expedite the Lookahead Minimization

An important practical idea is to choose the cost function approximation $\tilde{J}_{k+1}$ in Eq. (2.7) in a way that exploits the problem's structure to expedite the computation of the one-step lookahead minimizing control $\tilde{u}_k$. A noteworthy example arises in multiagent problems with decomposable structure, where the use of separable function approximations $\tilde{J}_{k+1}$ leads to decomposition of the minimization of Eq. (2.7) (see the books [Ber19a], Section 2.3.1, and [Ber20a], Section 3.2.4; the author has first encountered this idea through the MIT Ph.D. thesis of J. Kimemia [Kim82], [KGB82], which dealt with multi-machine flexible manufacturing). The following example illustrates a similar idea but in a different context.

Example 2.2.1 (Multidimensional Assignment)

Let us consider multidimensional assignment problems, a class of combinatorial problems that have both a temporal and a spacial allocation structure.

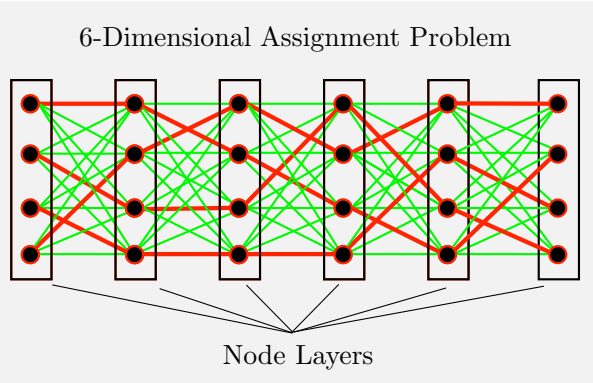

Figure 2.2.1 Illustration of the graph of an $(N+1)$-dimensional assignment problem (here $N = 5$). There are $N+1$ node layers each consisting of m nodes (here $m = 4$). Each grouping consists of $N+1$ nodes, one from each layer, and N corresponding arcs. An $(N+1)$-dimensional assignment consists of m node-disjoint groupings, where each node belongs to one and only one grouping (illustrated in the figure with thick lines). For each grouping, there is an associated cost that depends on the N-tuple of arcs comprising the grouping. The cost of an $(N+1)$-dimensional assignment is the sum of the costs of its m groupings. The difficulty here is that the cost of a grouping does not decompose into a sum of its N arc costs, so the problem cannot be solved by solving N decoupled 2-dimensional assignment problems (for a suboptimal approach based on enforced decoupling, see [Ber20a], Section 3.4.2).

They arise in various settings including scheduling, resource allocation, and data association. The general idea is to group together tasks, or resources, or data points, so as to optimize some objective. An example is when we are given a set of m jobs, m persons, and m machines, and we want to select m nonoverlapping (job, person, machine) triplets that correspond to minimum cost. Another example is data association problems, whereby we have collected data relating to the movement of some entities (e,g., persons, vehicles) sequentially over a number of time periods, and we want to group together data points that correspond to distinct entities for better inference purposes.

Mathematically, multidimensional assignment problems involve graphs consisting of $N+1$ subsets of nodes $\mathcal{N}_0, \mathcal{N}_1, \ldots, \mathcal{N}_N$, and referred to as *layers*. The arcs of the graphs are directed and are of the form (i, j), where i is a node in a layer $\mathcal{N}_k$, $k = 0, 1, \ldots, N-1$, and j is a node in the corresponding next layer $\mathcal{N}_{k+1}$. Thus we have a directed graph with nodes arranged in $N+1$ layers, and arcs connecting the nodes of each layer to the nodes in their adjacent layers; see Fig. 2.2.1.

We assume that $N \geq 2$, so there are at least three layers. For simplicity, we also assume that each of the layers $\mathcal{N}_k$ contains the same number of nodes, say m, and that there is a unique arc connecting each node in a given layer with each of the nodes of the adjacent layers. We will present an approximation in value space approach that can be implemented by solving 2-dimensional assignment problems for which very fast algorithms exist, such as the Hungarian method and the auction algorithm (see e.g., [Ber98]).

We consider subsets of $N+1$ nodes, referred to as *groupings*, which consist of a single node from every layer. For each grouping, there is an associated cost, which depends on the N-tuple of arcs that comprise the grouping. A partition of the set of nodes into m disjoint groupings (so that each node belongs to one and only one grouping) is called an $(N+1)$-*dimensional assignment*. The cost of an $(N+1)$-dimensional assignment is the sum of the costs of its m groupings. The problem is to find an $(N+1)$-dimensional assignment of minimum cost. The exact solution of the problem is very difficult when the cost of a grouping does not decompose into the sum of costs of the N arcs of the grouping (in which case the problem decouples into N easily solvable 2-dimensional assignment problems).

We formulate the problem as an N-stage sequential decision problem where at the kth stage we select the assignment arcs

$$u_k = \big\{(i_n, j_n) \mid n = 1, \ldots, m\big\}, \qquad k = 0, 1, \ldots, N-1, \tag{2.8}$$

which connect the nodes i_n of layer $\mathcal{N}_k$ to nodes j_n of layer $\mathcal{N}_{k+1}$ on a one-to-one basis. The control costraint set U_k is the set of legitimate assignments, namely those involving exactly one incident arc per node $i_n \in \mathcal{N}_k$, and exactly one incident arc per node j_n of layer $\mathcal{N}_{k+1}$.

We assume a cost function that has the general form

$$G(u_0, \ldots, u_{N-1}).$$

We use as state x_k the partial solution up to time k,

$$x_k = (u_0, \ldots, u_{k-1}),$$

so the system equation takes the simple form

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

cf. Fig. 2.1.4 and Section 1.6.3.

The exact DP algorithm takes the form

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

and,

$$J_k^*(x_k) = \min_{u_k \in U_k} J_{k+1}^*(x_k, u_k), \qquad \text{for all } x_k,\ k = 0, \ldots, N-1,$$

cf. Eqs. (2.3) and (2.4).

Since this DP algorithm is intractable, we resort to approximation of J_{k+1}^* by a function $\tilde{J}_{k+1}$. The one-step lookahead minimization becomes

$$\min_{u_k \in U_k} \tilde{J}_{k+1}(x_k, u_k), \qquad \text{for all } x_k,$$

but is still formidable because its search space U_k is very large. However, it can be greatly simplified by using a cost function approximation $\tilde{J}_{k+1}$ with a

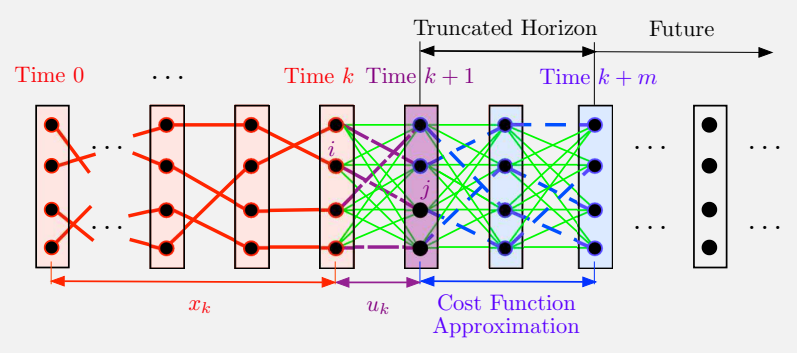

Figure 2.2.2 Illustration of a scheme for approximation in value space in a multiassignment problem. At time k, the next states have the form (x_k, u_k), where $x_k = (u_0, \ldots, u_{k-1})$ is the multiassignment trajectory generated up to time k. The scheme calculates the arc cost $c^{ij}_{k+1}(x_k)$ for each arc (i, j) connecting a node i of the time k layer with a node j of the time $k+1$ layer by extending heuristically the multiassignment trajectory along the "most likely" assignment arcs over some truncated horizon of m steps ($m = 2$ in the figure), and calculating $c^{ij}_{k+1}(x_k)$ as the "cost" of the multiassignment trajectory that starts at time 0, extends to time $k+m$ and passes through arc (i, j).

Once the arc costs $c^{ij}_{k+1}(x_k)$ have been calculated, the assignment u_k and corresponding multiassignment trajectory x_{k+1} are obtained by solving a 2-dimensional assignment problem, using for example the Hungarian method or the auction algorithm.

special structure that is suitable for the use of fast 2-dimensional assignment algorithms. This cost function approximation has the form

$$\tilde{J}_{k+1}(x_k, u_k) = \sum_{(i,j)\in u_k} c^{ij}_{k+1}(x_k),$$

where $\{(i, j) \in u_k\}$ denotes the set of m arcs (i, j) that correspond to the 2-dimensional assignment u_k, cf. Eq. (2.8) [thus the dependence of the right-hand side on u_k comes through the choice of arcs (i, j) specified by u_k]. The arc costs $c^{ij}_{k+1}(x_k)$ in this equation must be calculated for every possible arc (i, j) that connects a node $i \in \mathcal{N}_k$ with a node $j \in \mathcal{N}_{k+1}$; see Fig. 2.2.2.

Note that the arc costs $c^{ij}_{k+1}(x_k)$ may depend in a complicated way on the entire trajectory of previous 2-dimensional assignment choices $x_k = (u_0, \ldots, u_{k-1})$ and the problem data. For problems that involve tracking the movement of people or vehicles over time, the computation of $c^{ij}_{k+1}(x_k)$ relies on sensor data and problem-dependent circumstances. For other problems, enforced decomposition methods may be useful; see [Ber20a], Section 3.4.2. We refer to the data association literature for further details; see e.g., the survey by Emami et al. [EPE20].

Multistep Lookahead Minimization

The approximation in value space algorithm (2.7) involves a one-step lookahead minimization, since it solves a one-stage DP problem for each k. We

may also consider *ℓ-step lookahead*, which involves the solution of an ℓ-step deterministic DP problem, where ℓ is an integer, $1 < \ell < N - k$, with a terminal cost function approximation $\tilde{J}_{k+\ell}$.

As we have noted in Chapter 1, multistep lookahead typically provides better performance over one-step lookahead in approximation in value space schemes. For example in AlphaZero chess, long multistep lookahead is critical for good on-line performance. On the negative side, the solution of the multistep lookahead minimization problem is more time consuming than its one-step lookahead counterpart. However, the deterministic character of the lookahead minimization problem and the fact that it is solved for the single initial state x_k at each time k helps to limit the growth of the lookahead tree and to keep the computation manageable. Moreover, one may try to approximate the solution of the multistep lookahead minimization problem (see Section 2.4).

2.3 ROLLOUT ALGORITHMS FOR DISCRETE OPTIMIZATION

The construction of suitable approximate cost-to-go functions $\tilde{J}_{k+1}$ for approximation in value space can be done in many different ways, including some of the principal RL methods. A method of particular interest for our course is *rollout*, whereby the approximate values $\tilde{J}_{k+1}(x_{k+1})$ in Eq. (2.7) are obtained when needed by running for each $u_k \in U_k(x_k)$ a heuristic control scheme, called *base heuristic*, for a suitably large number of steps, starting from $x_{k+1} = f_k(x_k, u_k)$.

The base heuristic can be any method, which starting from a state x_{k+1} generates a sequence of controls $u_{k+1}, \ldots, u_{N-1}$, the corresponding sequence of states $x_{k+2}, \ldots, x_N$, and the cost of the heuristic starting from x_{k+1}, which we will generically denote by $H_{k+1}(x_{k+1})$ in this chapter:

$$H_{k+1}(x_{k+1}) = g_{k+1}(x_{k+1}, u_{k+1}) + \cdots + g_{N-1}(x_{N-1}, u_{N-1}) + g_N(x_N).$$

This value of $H_{k+1}(x_{k+1})$ is the one used as the approximate cost $\tilde{J}_{k+1}(x_{k+1})$ in the corresponding approximation in value space scheme (2.7).

In this section, we will develop in more detail the theory of rollout with one-step lookahead minimization for deterministic problems, including the important issue of cost improvement. We will also illustrate several variants of the method, and we will consider questions of efficient implementation. We will then discuss examples of discrete optimization applications.

Let us consider a deterministic DP problem with a finite number of controls and a given initial state (so the number of states that can be reached from the initial state is also finite). We first focus on the pure form of rollout that uses one-step lookahead without truncation, and hence no terminal cost approximation. Given a state x_k at time k, this algorithm considers the tail subproblems that start at every possible next state x_{k+1}, and solves them suboptimally with the base heuristic.

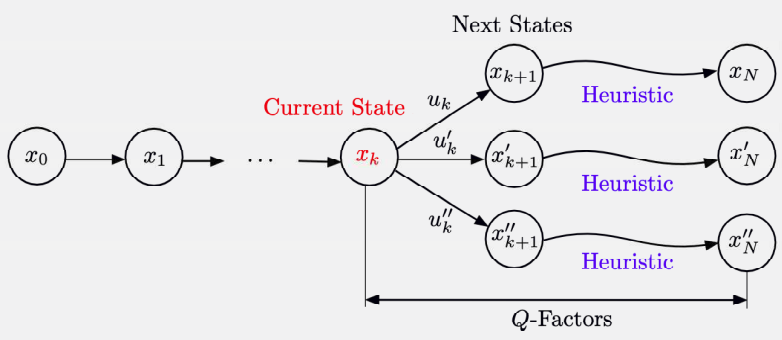

Figure 2.3.1 Schematic illustration of rollout with one-step lookahead for a deterministic problem. At state x_k, for every pair (x_k, u_k), $u_k \in U_k(x_k)$, the base heuristic generates a Q-factor

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

and the rollout algorithm selects the control $\tilde{\mu}_k(x_k)$ with minimal Q-factor.

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

$$x_{t+1} = f_t(x_t, u_t), \qquad t = k+1, \ldots, N-1,$$

and the corresponding cost

$$H_{k+1}(x_{k+1}) = g_{k+1}(x_{k+1}, u_{k+1}) + \cdots + g_{N-1}(x_{N-1}, u_{N-1}) + g_N(x_N).$$

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:

$$g_k(x_k, u_k) + H_{k+1}(x_{k+1}).$$

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

$$\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k), \tag{2.9}$$

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

$$\tilde{Q}_k(x_k, u_k) = g_k(x_k, u_k) + H_{k+1}\big(f_k(x_k, u_k)\big); \tag{2.10}$$

see Fig. 2.3.1. The rollout algorithm thus defines a suboptimal policy $\tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$, referred to as the *rollout policy*, where for each x_k and k, $\tilde{\mu}_k(x_k)$ is the control produced by the Q-factor minimization (2.9).

Note that the rollout algorithm requires running the base heuristic for a number of times that is bounded by Nn, where n is an upper bound on the number of control choices available at each state. Thus if n is small relative to N, the algorithm requires computation equal to a small multiple of N times the computation time for a single application of the base heuristic. Similarly, if n is bounded by a polynomial in N, the ratio of the rollout algorithm computation time to the base heuristic computation time is a polynomial in N.

In Section 1.2 we considered an example of rollout involving the traveling salesman problem and the nearest neighbor heuristic (cf. Examples 1.2.2 and 1.2.3). Let us consider another example, which involves a classical discrete optimization problem.

Example 2.3.1 (Multi-Vehicle Routing)

Consider m vehicles that move along the arcs of a given graph. Some of the nodes of the graph include a task to be performed by the vehicles. Each task will be performed only once, immediately after some vehicle reaches the corresponding node for the first time. We assume a horizon that is large enough to allow every task to be performed. The problem is to find a route for each vehicle so that the tasks are collectively performed by the vehicles in a minimum number of moves. To express this objective, we assume that for each move by a vehicle there is a cost of one unit. These costs are summed up to the point where all the tasks have been performed.

For a large number m of vehicles and a complicated graph, this is a nontrivial combinatorial problem. It can be approached by DP, like any discrete deterministic optimization problem, as we have discussed. In particular, we can view as state at a given stage the m-tuple of current positions of the vehicles together with the list of pending tasks. Unfortunately, however, the number of these states can be enormous (it increases exponentially with the number of tasks and the number of vehicles), so an exact DP solution is intractable.

This motivates an optimization in value space approach based on rollout. For this we need an easily implementable base heuristic that will solve suboptimally the problem starting from any state x_{k+1}, and will provide the cost approximation $\tilde{J}_{k+1}(x_{k+1})$ in Eq. (2.7). One possibility is based on the vehicles choosing their actions selfishly and without coordination, along shortest paths to their nearest pending task.

To illustrate, consider the two-vehicle problem of Fig. 2.3.2. The base heuristic is to move each vehicle one step at a time towards its nearest pending task, until all tasks have been performed.

The rollout algorithm will work as follows. At a given state x_k [involving for example vehicle positions at the node pair $(1, 2)$ and tasks at nodes 7 and 9, as in Fig. 2.3.2], we consider all possible joint vehicle moves (the controls u_k at the state) resulting in the node pairs (3,5), (4,5), (3,4), (4,4), corresponding to the next states x_{k+1} [thus, as an example (3,5) corresponds to vehicle 1 moving from 1 to 3, and vehicle 2 moving from 2 to 5]. We then run the base heuristic starting from each of these node pairs, and accumulate the

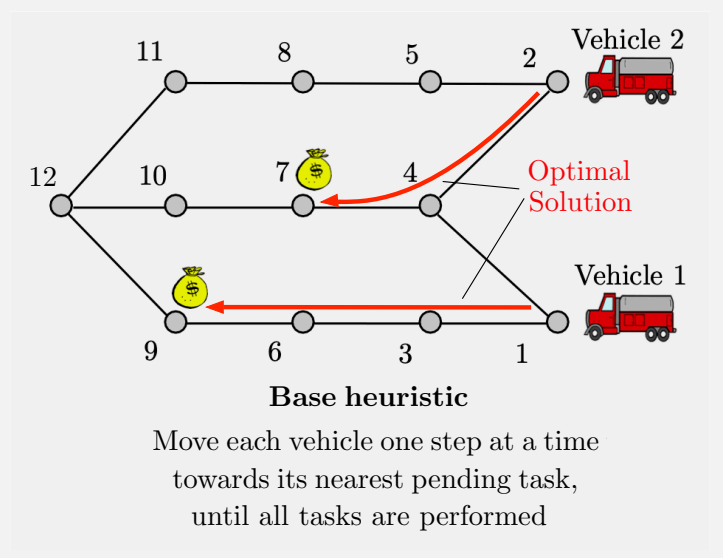

Figure 2.3.2 An instance of the vehicle routing problem of Example 2.3.1. The two vehicles aim to collectively perform the two tasks, at nodes 7 and 9, as fast as possible, by each moving to a neighboring node at each step. The optimal routes are shown.

incurred costs up to the time when both tasks are completed. For example starting from the vehicle positions/next state (3,5), the heuristic will produce the following sequence of moves:

- Vehicles 1 and 2 move from (3,5) to (6,2).
- Vehicles 1 and 2 move from (6,2) to (9,4), and the task at 9 is performed.
- Vehicles 1 and 2 move from (9,4) to (12,7), and the task at 7 is performed.

The two tasks are thus performed in a total of 6 vehicles moves once the move to (3,5) has been made.

The process of running the heuristic is repeated from the other three vehicle position pairs/next states (4,5), (3,4) (4,4), and the heuristic cost (number of moves) is recorded. We then choose the next state that corresponds to minimum cost. In our case the joint move to state x_{k+1} that involves the pair $(3, 4)$ produces the sequence

- Vehicles 1 and 2 move from (3,4) to (6,7), and the task at 7 is performed.
- Vehicles 1 and 2 move from (6,7) to (9,4), and the task at 9 is performed.

and performs the two tasks in a total of 6 vehicle moves. It can be verified that it yields minimum first stage cost plus heuristic cost from the next state, as per Eq. (2.7). Thus, the rollout algorithm will choose to move the vehicles to state (3,4) from state (1,2). At that state the rollout process will be repeated, i.e., consider the possible next joint moves to the node pairs (6,7), (6,2), (6,1),

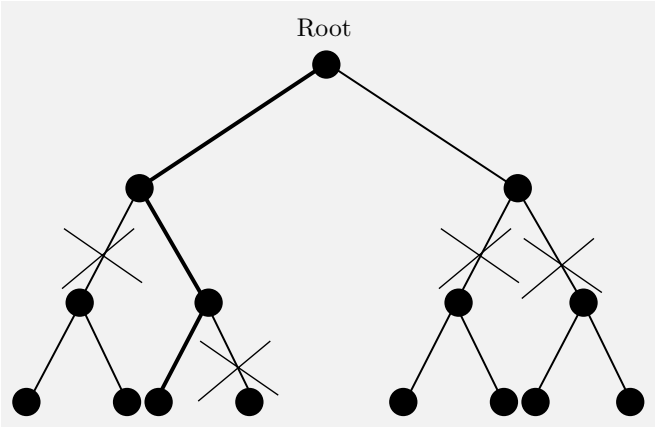

Figure 2.3.3 Binary tree for the breakthrough problem. Each arc is either free or is blocked (crossed out in the figure). The problem is to find a path from the root to one of the leaves, which is free (such as the one shown with thick lines). Note that the problem is related to the constraint programming problem discussed earlier in Section 2.1.

(1,7), (1,2), (1,1), perform a heuristic calculation from each, compare, etc.

It can be verified that the rollout algorithm starting from the state (1,2) shown in Fig. 2.3.2 will attain the optimal cost (a total of 6 vehicle moves). It will perform much better than the heuristic, which starting from state (1,2), will move the two vehicles together to state (4,4), then to (7,7), then to (10,10), then to (12,12), and finally to (9,9), (a total of 10 vehicle moves). This is an instance of the cost improvement property of the rollout algorithm: it performs better than its base heuristic (under appropriate conditions).

Let us finally note that the computation required by in rollout algorithm increases exponentially with the number m of vehicles, since the number of m-tuples of moves at each stage increases exponentially with m. This is the type of problem where multiagent rollout can attain great computational savings; cf. Section 1.6.5, and the subsequent Section 2.9.

Here is an example of a search problem, whose exact solution complexity grows exponentially with the problem size, but can be addressed with a greedy heuristic as well as with the corresponding rollout algorithm.

Example 2.3.2 (The Breakthrough Problem)

Consider a binary tree with N stages as shown in Fig. 2.3.3. Stage k of the tree has 2^k nodes, with the node of stage 0 called *root* and the nodes of stage N called *leaves*. There are two types of tree arcs: *free* and *blocked*. A free (or blocked) arc can (cannot, respectively) be traversed in the direction from the root to the leaves. The objective is to break through the graph with a sequence of free arcs (a free path) starting from the root, and ending at one of the leaves. (A variant of this problem is to introduce a positive cost $c > 0$ for traversing a blocked arc, and 0 cost for traversing a free arc.)

One may use DP to discover a free path (if one exists) by starting from the last stage and by proceeding backwards to the root node. The kth step of the algorithm determines for each node of stage $N-k$ whether there is a free path from that node to some leaf node, by using the results of the preceding step. The amount of calculation at the kth step is $O(2^{N-k})$. Adding the computations for the N stages, we see that the total amount of calculation is $O(N2^N)$, so it increases exponentially with the number of stages. For this reason it is interesting to consider heuristics requiring computation that is linear or polynomial in N, but may sometimes fail to determine a free path, even when a free path exists.

Thus, one may suboptimally use a *greedy* algorithm, which starts at the root node, selects a free outgoing arc (if one is available), and tries to construct a free path by adding successively nodes to the path. At the current node, if one of the outgoing arcs is free and the other is blocked, the greedy algorithm selects the free arc. Otherwise, it selects one of the two outgoing arcs according to some fixed rule that depends only on the current node (and not on the status of other arcs). Clearly, the greedy algorithm may fail to find a free path even if such a path exists, as can be seen from Fig. 2.3.3. On the other hand the amount of computation associated with the greedy algorithm is $O(N)$, which is much faster than the $O(N2^N)$ computation of the DP algorithm. Thus we may view the greedy algorithm as a fast heuristic, which is suboptimal in the sense that there are problem instances where it fails while the DP algorithm succeeds.

One may also consider a rollout algorithm that uses the greedy algorithm as the base heuristic. There is an analysis that compares the probability of finding a breakthrough solution with the greedy and with the rollout algorithm for random instances of binary trees (each arc is independently free or blocked with given probability p). This analysis is given in Section 6.4 of the book [Ber17a], and shows that asymptotically, the rollout algorithm requires $O(N)$ times more computation, but has an $O(N)$ times larger probability of finding a free path than the greedy algorithm.

This tradeoff is qualitatively typical: the rollout algorithm achieves a substantial performance improvement over the base heuristic at the expense of extra computation that is equal to the computation time of the base heuristic times a factor that is a low order polynomial of the problem size.

2.3.1 Cost Improvement with Rollout - Sequential Consistency, Sequential Improvement

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, and others.

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*, introduced in the paper by Bertsekas, Tsitsiklis, and Wu [BTW97], and we later show how to modify the algorithm to deal with the case where these conditions are not met.

Definition 2.3.1: We say that the base heuristic is *sequentially consistent* if it has the property that when it generates the sequence

$$\{x_k, u_k, x_{k+1}, u_{k+1}, \ldots, x_N\}$$

starting from state x_k, it also generates the sequence

$$\{x_{k+1}, u_{k+1}, \ldots, x_N\}$$

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 1.2.3 and the heuristics used in the multivehicle routing Example 2.3.1 are sequentially consistent. Similar examples include the use of various types of greedy/myopic heuristics (Section 6.4 of the book [Ber17a] provides additional examples).† 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.

A sequentially consistent base heuristic can be recognized by the fact that it will apply the same control u_k at a state x_k, no matter what position x_k occupies in a trajectory generated by the base heuristic. Thus *a base heuristic is sequentially consistent if and only if it defines a legitimate DP policy*. This is the policy that moves from x_k to the state x_{k+1} that lies on the state trajectory $\{x_k, x_{k+1}, \ldots, x_N\}$ that the base heuristic generates. Similarly the policy moves from x_n to the state x_{n+1} for $n = k+1, \ldots, N-1$.

† A subtle but important point relates to how one breaks ties while implementing greedy base heuristics. For sequential consistency, one must break ties in a consistent way at various states, i.e., using a fixed rule at each state encountered by the base heuristic. In particular, randomization among multiple controls, which are ranked as equal by the greedy optimization of the heuristic, violates sequential consistency, and can lead to serious degradation of the corresponding rollout algorithm's performance.

We will now show that the rollout algorithm obtained with a sequentially consistent base heuristic has a fundamental cost improvement property: it yields no worse cost than the base heuristic. The amount of cost improvement cannot be easily quantified, but is determined by the performance of the Newton step associated with the rollout policy, so it can be very substantial; cf. the discussion of Chapter 1.

Proposition 2.3.1: (Cost Improvement Under Sequential Consistency) Consider the rollout policy $\tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$ obtained with a sequentially consistent base heuristic, and let $J_{k,\tilde{\pi}}(x_k)$ denote the cost obtained with $\tilde{\pi}$ starting from x_k at time k. Then we have

$$J_{k,\tilde{\pi}}(x_k) \leq H_k(x_k), \qquad \text{for all } x_k \text{ and } k, \tag{2.11}$$

where $H_k(x_k)$ denotes the cost of the base heuristic starting from x_k.

Proof: We prove the inequality (2.11) 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

$$\begin{aligned} J_{k,\tilde{\pi}}(x_k) &= g_k\big(x_k, \tilde{\mu}_k(x_k)\big) + J_{k+1,\tilde{\pi}}\Big(f_k\big(x_k, \tilde{\mu}_k(x_k)\big)\Big) \\ &\leq g_k\big(x_k, \tilde{\mu}_k(x_k)\big) + H_{k+1}\Big(f_k\big(x_k, \tilde{\mu}_k(x_k)\big)\Big) \\ &= \min_{u_k \in U_k(x_k)} \Big[g_k(x_k, u_k) + H_{k+1}\big(f_k(x_k, u_k)\big)\Big] \\ &\leq g_k\big(x_k, \overline{u}_k\big) + H_{k+1}\big(f_k(x_k, \overline{u}_k)\big) \\ &= H_k(x_k), \end{aligned} \tag{2.12}$$

where:

- The first equality is the DP equation for the rollout policy $\tilde{\pi}$.
- The first inequality holds by the induction hypothesis.
- The second equality holds by the definition of the rollout algorithm.
- 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 proof of the cost improvement property (2.11). **Q.E.D.**

Sequential Improvement

We will next 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. Eq. (2.10)], and $H_{k+1}\big(f_k(x_k, u_k)\big)$ denotes the cost of the trajectory of the base heuristic starting from state $f_k(x_k, u_k)$.

Definition 2.3.2: We say that the base heuristic is *sequentially improving* if for all x_k and k, we have

$$\min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k) \leq H_k(x_k). \tag{2.13}$$

In words, the sequential improvement property (2.13) states that

Minimal heuristic Q-factor at $x_k \leq$ Heuristic cost at x_k.

Note that *when the heuristic is sequentially consistent it is also sequentially improving*. This follows from the preceding relation, since for a sequentially consistent heuristic, the heuristic cost at x_k is equal to the Q-factor of the control $\overline{u}_k$ that the heuristic applies at x_k,

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

which is greater or equal to the minimal Q-factor at x_k. This implies Eq. (2.13). A sequentially improving heuristic yields policy improvement as the next proposition shows.

Proposition 2.3.2: (Cost Improvement Under Sequential Improvement) Consider the rollout policy $\tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$ obtained with a sequentially improving base heuristic, and let $J_{k,\tilde{\pi}}(x_k)$ denote the cost obtained with $\tilde{\pi}$ starting from x_k at time k. Then

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

where $H_k(x_k)$ denotes the cost of the base heuristic starting from x_k.

Proof: Follows from the calculation of Eq. (2.12), by replacing the last two steps (which rely on sequential consistency) with Eq. (2.13). **Q.E.D.**

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. In fact *the algorithm has a monotonic improvement property, whereby it discovers a sequence of improved trajectories*. In particular, let us denote the trajectory generated by the base heuristic starting from x_0 by

$$T_0 = (x_0, u_0, \ldots, x_{N-1}, u_{N-1}, x_N),$$

and the final trajectory generated by the rollout algorithm starting from x_0 by

$$T_N = (x_0, \tilde{u}_0, \tilde{x}_1, \tilde{u}_1, \ldots, \tilde{x}_{N-1}, \tilde{u}_{N-1}, \tilde{x}_N).$$

Consider also the intermediate trajectories generated by the rollout algorithm given by

$$T_k = (x_0, \tilde{u}_0, \tilde{x}_1, \tilde{u}_1, \ldots, \tilde{x}_k, u_k, \ldots, x_{N-1}, u_{N-1}, x_N), \quad k = 1, \ldots, N-1,$$

where

$$(\tilde{x}_k, u_k, \ldots, x_{N-1}, u_{N-1}, x_N),$$

is the trajectory generated by the base heuristic starting from $\tilde{x}_k$. Then, by using the sequential improvement condition, it can be proved (see Fig. 2.3.4) that

$$\text{Cost of } T_0 \geq \cdots \geq \text{Cost of } T_k \geq \text{Cost of } T_{k+1} \geq \cdots \geq \text{Cost of } T_N. \quad (2.14)$$

Empirically, it has been observed that the cost improvement obtained by rollout with a sequentially improving heuristic is typically considerable and often dramatic. In particular, many case studies, dating to the middle 1990s, indicate consistently good performance of rollout; see the last section of this chapter for a bibliography. The DP textbook [Ber17a] provides some detailed worked-out examples (Chapter 6, Examples 6.4.2, 6.4.5, 6.4.6, and Exercises 6.11, 6.14, 6.15, 6.16). 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 N. It is generally hard to quantify the amount of performance improvement, but the computational results obtained from the case studies are consistent with the Newton step interpretations that we discussed in Chapter 1.

The books [Ber19a] (Section 2.5.1) and [Ber20a] (Section 3.1) show that the sequential improvement condition is satisfied in the context of MPC, and is the underlying reason for the stability properties of the MPC scheme. On the other hand the base heuristic underlying the classical

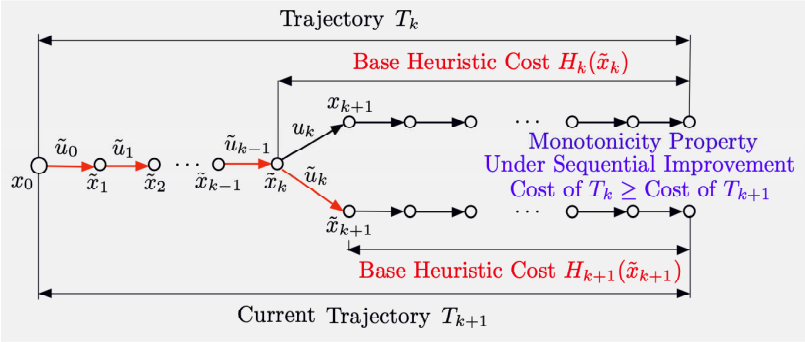

Figure 2.3.4 Proof of the monotonicity property (2.14). At $\tilde{x}_k$, the kth state generated by the rollout algorithm, we compare the "current" trajectory T_k whose cost is the sum of the cost of the current partial trajectory $(x_0, \tilde{u}_0, \tilde{x}_1, \tilde{u}_1, \ldots, \tilde{x}_k)$ and the cost $H_k(\tilde{x}_k)$ of the base heuristic starting from $\tilde{x}_k$, and the trajectory T_{k+1} whose cost is the sum of the cost of the partial rollout trajectory $(x_0, \tilde{u}_0, \tilde{x}_1, \tilde{u}_1, \ldots, \tilde{x}_k)$, and the Q-factor $\tilde{Q}_k(\tilde{x}_k, \tilde{u}_k)$ of the base heuristic starting from $(\tilde{x}_k, \tilde{u}_k)$. The sequential improvement condition guarantees that

$$H_k(\tilde{x}_k) \geq \tilde{Q}_k(\tilde{x}_k, \tilde{u}_k),$$

which implies that

$$\text{Cost of } T_k \geq \text{Cost of } T_{k+1}.$$

If strict inequality holds, the rollout algorithm will switch from T_k and follow T_{k+1}; cf. the traveling salesman Example 1.2.3.

form of the MPC scheme is not sequentially consistent (see the preceding references).

Generally, the sequential improvement condition may not hold for a given base heuristic. This is not surprising since any heuristic (no matter how inconsistent or silly) is in principle admissible to use as base heuristic. Here is an example:

Example 2.3.3 (Sequential Improvement Violation)

Consider the 2-stage problem shown in Fig. 2.3.5, which involves two states at each of stages 1 and 2, and the controls shown. Suppose that the unique optimal trajectory is $(x_0, u_0^*, x_1^*, u_1^*, x_2^*)$, and that the base heuristic produces this optimal trajectory starting at x_0. The rollout algorithm chooses a control at x_0 as follows: it runs the base heuristic to construct a trajectory starting from x_1^* and $\tilde{x}_1$, with corresponding costs $H_1(x_1^*)$ and $H_1(\tilde{x}_1)$. If

$$g_0(x_0, u_0^*) + H_1(x_1^*) > g_0(x_0, \tilde{u}_0) + H_1(\tilde{x}_1), \tag{2.15}$$

the rollout algorithm rejects the optimal control u_0^* in favor of the alternative control $\tilde{u}_0$. The inequality above will occur if the base heuristic chooses $\bar{u}_1$ at x_1^* (there is nothing to prevent this from happening, since the base heuristic

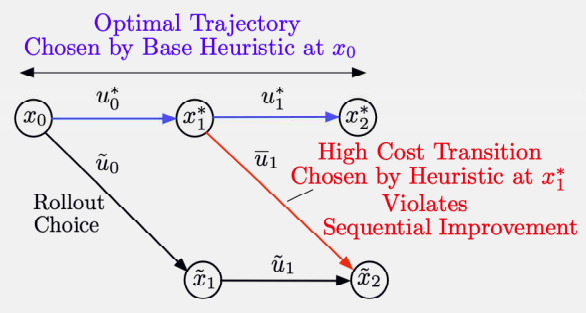

Figure 2.3.5 A 2-stage problem with states $x_1^*, \tilde{x}_1$ at stage 1, and states $x_2^*, \tilde{x}_2$ at stage 2. The controls and corresponding transitions are as shown in the figure. The rollout choice at the initial state x_0 is strictly suboptimal, while the base heuristic choice is optimal. The reason is that the base heuristic is not sequentially improving and makes the suboptimal choice $\overline{u}_1$ at x_1^*, but makes the different (optimal) choice u_1^* when run from x_0.

is arbitrary), and moreover the cost $g_1(x_1^*, \bar{u}_1) + g_2(\tilde{x}_2)$, which is equal to $H_1(x_1^*)$ is high enough.

Let us also verify that if the inequality (2.15) holds then the heuristic is not sequentially improving at x_0, i.e., that

$$H_0(x_0) < \min\big\{g_0(x_0, u_0^*) + H_1(x_1^*),\, g_0(x_0, \tilde{u}_0) + H_1(\tilde{x}_1)\big\}.$$

Indeed, this is true because $H_0(x_0)$ is the optimal cost

$$H_0(x_0) = g_0(x_0, u_0^*) + g_1(x_1^*, u_1^*) + g_2(x_2^*),$$

and must be smaller than both

$$g_0(x_0, u_0^*) + H_1(x_1^*),$$

which is the cost of the trajectory $(x_0, u_0^*, x_1^*, \overline{u}_1, \tilde{x}_2)$, and

$$g_0(x_0, \tilde{u}_0) + H_1(\tilde{x}_1),$$

which is the cost of the trajectory $(x_0, \tilde{u}_0, \tilde{x}_1, \tilde{u}_1, \tilde{x}_2)$.

The preceding example and the monotonicity property (2.14) suggest a simple enhancement to the rollout algorithm, which detects when the sequential improvement condition is violated and takes corrective measures. In this algorithmic variant, called *fortified rollout*, we maintain the best trajectory obtained so far, and keep following that trajectory up to the point where we discover another trajectory that has improved cost (see the next section).

2.3.2 The Fortified Rollout Algorithm

In this section we describe a rollout variant that implicitly enforces the sequential improvement property. 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 we have the trajectory

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

that has been constructed by rollout, called *permanent trajectory*, and we also store a *tentative best trajectory*

$$\overline{T}_k = \{x_0, u_0, \ldots, u_{k-1}, x_k, \overline{u}_k, \overline{x}_{k+1}, \overline{u}_{k+1}, \ldots, \overline{u}_{N-1}, \overline{x}_N\}$$

with corresponding cost

$$C(\overline{T}_k) = \sum_{t=0}^{k-1} g_t(x_t, u_t) + g_k(x_k, \overline{u}_k) + \sum_{t=k+1}^{N-1} g_t(\overline{x}_t, \overline{u}_t) + g_N(\overline{x}_N).$$

The tentative best trajectory $\overline{T}_k$ is the end-to-end trajectory that has minimum cost out of all end-to-end trajectories 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 *discard the suggestion of the rollout algorithm at every state x_k where it produces a trajectory that is inferior to $\overline{T}_k$, and use $\overline{T}_k$ instead* (see Fig. 2.3.6).

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

$$\tilde{T}_k = \{x_0, u_0, \ldots, u_{k-1}, x_k, \tilde{u}_k, \tilde{x}_{k+1}, \tilde{u}_{k+1}, \ldots, \tilde{u}_{N-1}, \tilde{x}_N\}$$

with corresponding cost

$$C(\tilde{T}_k) = \sum_{t=0}^{k-1} g_t(x_t, u_t) + g_k(x_k, \tilde{u}_k) + \sum_{t=k+1}^{N-1} g_t(\tilde{x}_t, \tilde{u}_t) + g_N(\tilde{x}_N).$$

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

$$C(\overline{T}_k) \leq C(\tilde{T}_k),$$

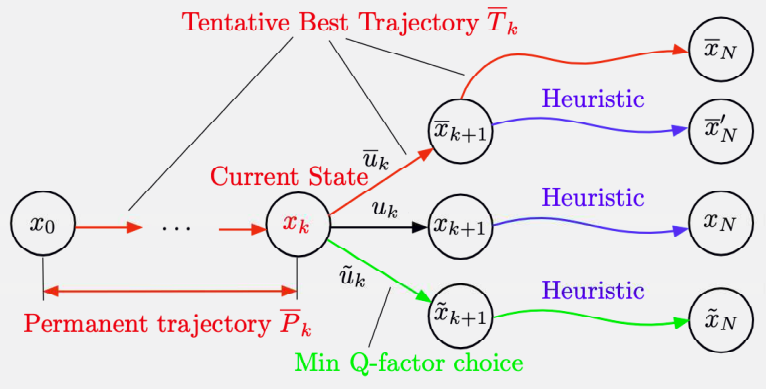

Figure 2.3.6 Schematic illustration of fortified rollout. After k steps, we have constructed the permanent trajectory

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

and the tentative best trajectory

$$\overline{T}_k = \{x_0, u_0, \ldots, u_{k-1}, x_k, \overline{u}_k, \overline{x}_{k+1}, \overline{u}_{k+1}, \ldots, \overline{u}_{N-1}, \overline{x}_N\},$$

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

$$\tilde{T}_k = \{x_0, u_0, \ldots, u_{k-1}, x_k, \tilde{u}_k, \tilde{x}_{k+1}, \tilde{u}_{k+1}, \ldots, \tilde{u}_{N-1}, \tilde{x}_N\}.$$

If the cost of the end-to-end trajectory $\tilde{T}_k$ is lower than the cost of $\overline{T}_k$, we add $(\tilde{u}_k, \tilde{x}_{k+1})$ to the permanent trajectory and set the tentative best trajectory to $\overline{T}_{k+1} = \tilde{T}_k$. Otherwise we add $(\overline{u}_k, \overline{x}_{k+1})$ to the permanent trajectory and keep the tentative best trajectory unchanged: $\overline{T}_{k+1} = \overline{T}_k$.

the algorithm sets the next state and corresponding tentative best trajectory to

$$x_{k+1} = \overline{x}_{k+1}, \qquad \overline{T}_{k+1} = \overline{T}_k,$$

and if

$$C(\overline{T}_k) > C(\tilde{T}_k),$$

it sets the next state and corresponding tentative best trajectory to

$$x_{k+1} = \tilde{x}_{k+1}, \qquad \overline{T}_{k+1} = \tilde{T}_k.$$

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

running the base heuristic from all possible next states x_{k+1}.† It follows that at every state the tentative best trajectory has no larger cost than the initial tentative best 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 often important to use the fortified version if the base heuristic is not known to be sequentially improving. Fortunately, the fortified version involves hardly any additional computational cost.

As expected, when the base heuristic generates an optimal trajectory, the fortified rollout algorithm will also generate the same trajectory. This is illustrated by the following example.

Example 2.3.4

Let us consider the application of the fortified rollout algorithm to the problem of Example 2.3.3 and see how it addresses the issue of cost improvement. The fortified rollout algorithm stores as initial tentative best trajectory the optimal trajectory $(x_0, u_0^*, x_1^*, u_1^*, x_2^*)$ generated by the base heuristic at x_0. Then, starting at x_0, it runs the heuristic from x_1^* and $\tilde{x}_1$, and (despite the fact that the ordinary rollout algorithm prefers going to $\tilde{x}_1$ rather than x_1^*) it discards the control $\tilde{u}_0$ in favor of u_0^*, which is dictated by the tentative best trajectory. It then sets the tentative best trajectory to $(x_0, u_0^*, x_1^*, u_1^*, x_2^*)$.

We finally note that the fortified rollout algorithm can be used in a different setting to restore and maintain the cost improvement property. Suppose in particular that the rollout minimization at each step is performed with approximations. For example the control u_k may have multiple independently constrained components, i.e.,

$$u_k = (u_k^1, \ldots, u_k^m), \qquad U_k(x_k) = U_k^1(x_k) \times \cdots \times U_k^m(x_k).$$

Then, to take advantage of distributed computation, it may be attractive to decompose the optimization over u_k in the rollout algorithm,

$$\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \Big[g_k(x_k, u_k) + H_{k+1}\big(f_k(x_k, u_k)\big) \Big],$$

into an (approximate) parallel optimization over the components u_k^i (or subgroups of these components). However, as a result of approximate optimization over u_k, the cost improvement property may be degraded, even if the sequential improvement assumption holds. In this case by maintaining

† The base heuristic may also be run from a subset of the possible next states x_{k+1}, as in the case where a simplified version of rollout is used; cf. Section 2.3.4. Then fortified rollout will still guarantee a cost improvement property.

the tentative best trajectory, starting with the one produced by the base heuristic at the initial condition, we can ensure that the fortified rollout algorithm, even with approximate minimization, will not produce an inferior solution to the one of the base heuristic.

2.3.3 Using Multiple Base Heuristics - Parallel Rollout

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 out of the trajectories produced by the entire collection of heuristics. The superheuristic can then be used as the base heuristic for a rollout algorithm.†

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

$$\tilde{T}_{k+1}^{\ell} = \{x_{k+1}, \tilde{u}_{k+1}^{\ell}, x_{k+2}, \ldots, \tilde{u}_{N-1}^{\ell}, \tilde{x}_N^{\ell}\},$$

and corresponding cost $C(\tilde{T}_{k+1}^{\ell})$. The superheuristic then produces at x_{k+1} the trajectory $\tilde{T}_{k+1}^{\ell}$ for which $C(\tilde{T}_{k+1}^{\ell})$ is minimum. The rollout algorithm selects at state x_k the control u_k that minimizes the minimal Q-factor:

$$\tilde{u}_k \in \arg \min_{u_k \in U_k(x_k)} \min_{\ell=1,\ldots,m} \tilde{Q}_k^{\ell}(x_k, u_k),$$

where

$$\tilde{Q}_k^{\ell}(x_k, u_k) = g_k(x_k, u_k) + C(\tilde{T}_{k+1}^{\ell})$$

is the cost of the trajectory $(x_k, u_k, \tilde{T}_{k+1}^{\ell})$. Note that the Q-factors of the different heuristics can be computed independently and in parallel. In view of this fact, the rollout scheme just described is sometimes referred to as *parallel rollout.*

An interesting property, which can be readily verified by using the definitions, is that *if all the heuristics are sequentially improving, the same is true for the superheuristic*, something that is also suggested by Fig. 2.3.4. Indeed, let us write the sequential improvement condition (2.13) for each of the base heuristics

$$\min_{u_k \in U_k(x_k)} \tilde{Q}_k^{\ell}(x_k, u_k) \leq H_k^{\ell}(x_k), \qquad \ell = 1, \ldots, m,$$

† A related practically interesting possibility is to introduce a partition of the state space into subsets, and a collection of multiple heuristics that are specially tailored to the subsets. We may then select the appropriate heuristic to use on each subset of the partition. In fact one may use a collection of multiple heuristics tailored to each subset of the state space partition, and at each state, select out of all the heuristics that apply, the one that yields minimum cost.

where $\tilde{Q}_k^\ell(x_k,u_k)$ and $H_k^\ell(x_k)$ are Q-factors and heuristic costs that correspond to the ℓth heuristic. Then by taking minimum over ℓ, we have

$$\min_{\ell=1,\ldots,m}\min_{u_k\in U_k(x_k)}\tilde{Q}_k^\ell(x_k,u_k)\leq\min_{\ell=1,\ldots,m}H_k^\ell(x_k),$$

for all x_k and k. By interchanging the order of the minimizations of the left side, we then obtain

$$\min_{u_k\in U_k(x_k)}\underbrace{\min_{\ell=1,\ldots,m}\tilde{Q}_k^\ell(x_k,u_k)}_{\text{Superheuristic Q-factor}}\leq\underbrace{\min_{\ell=1,\ldots,m}H_k^\ell(x_k)}_{\text{Superheuristic cost}},$$

which is precisely the sequential improvement condition (2.13) for the superheuristic.

2.3.4 Simplified Rollout Algorithms

We will now consider a rollout variant, called *simplified rollout*, which is motivated by problems where the control constraint set $U_k(x_k)$ is either infinite or finite but very large. Then the minimization

$$\tilde{\mu}_k(x_k)\in\arg\min_{u_k\in U_k(x_k)}\tilde{Q}_k(x_k,u_k),\tag{2.16}$$

[cf. Eqs. (2.9) and (2.10)], may be unwieldy, since the number of Q-factors

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

is accordingly infinite or large.

To remedy this situation, we may replace $U_k(x_k)$ with a smaller finite subset $\overline{U}_k(x_k)$:

$$\overline{U}_k(x_k)\subset U_k(x_k).$$

The rollout control $\tilde{\mu}_k(x_k)$ in this variant is one that attains the minimum of $\tilde{Q}_k(x_k,u_k)$ over $u_k\in\overline{U}_k(x_k)$:

$$\tilde{\mu}_k(x_k)\in\arg\min_{u_k\in\overline{U}_k(x_k)}\tilde{Q}_k(x_k,u_k).\tag{2.17}$$

An example is when $\overline{U}_k(x_k)$ results from discretization of an infinite set $U_k(x_k)$. Another possibility is when by using some preliminary approximate optimization, we can identify a subset $\overline{U}_k(x_k)$ of promising controls by using some heuristic method, and to save computation, we restrict attention to this subset. A related possibility is to generate $\overline{U}_k(x_k)$ by some iterative or random search method that explores intelligently the set $U_k(x_k)$ with the aim to minimize $\tilde{Q}_k(x_k,u_k)$ [cf. Eq. (2.16)].

It turns out that the proof of the cost improvement property of Prop. 2.3.2,

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

goes through if the following modified sequential improvement property holds:

$$\min_{u_k \in \overline{U}_k(x_k)} \tilde{Q}_k(x_k, u_k) \leq H_k(x_k). \tag{2.18}$$

This can be seen by verifying that Eq. (2.18) is sufficient to guarantee that the monotone improvement Eq. (2.14) is satisfied. The condition (2.18) is very simple to satisfy if the base heuristic is sequentially consistent, in which case the control $\overline{u}_k$ selected by the base heuristic satisfies

$$\tilde{Q}_k(x_k, \overline{u}_k) = H_k(x_k).$$

In particular, for the property (2.18) to hold, it is sufficient that $\overline{U}_k(x_k)$ contains the base heuristic choice $\overline{u}_k$.

The idea of replacing the minimization (2.16) by the simpler minimization (2.17) can be extended. In particular, by working through the preceding argument, it can be seen that *any policy*

$$\tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$$

such that $\tilde{\mu}_k(x_k)$ *satisfies the condition*

$$\tilde{Q}_k\big(x_k, \tilde{\mu}_k(x_k)\big) \leq H_k(x_k),$$

for all x_k *and* k*, guarantees the modified sequential improvement property (2.18), and hence also the cost improvement property.* A prominent example of such an algorithm arises in the multiagent case where u has m components, $u = (u^1, \ldots, u^m)$, and the minimization over $U_k^1(x_k) \times \cdots \times U_k^m(x_k)$ is replaced by a sequence of single component minimizations, one-component-at-a-time; cf. Section 1.6.5. Of course in the multiagent case, the one-component-at-a-time implementation has an additional favorable property: it can be viewed as rollout (without simplification) for a modified but equivalent DP problem (see Section 1.6.5).

2.3.5 Truncated Rollout with Terminal Cost Approximation

An important variation of rollout algorithms is *truncated rollout* with terminal cost approximation. Here the rollout trajectories are obtained by running the base policy from the leaf nodes of the lookahead tree, but they are truncated after a given number of steps, while a terminal cost approximation is added to the heuristic cost to compensate for the resulting error. This is important for problems with a large number of stages, and it is also

essential for infinite horizon problems where the rollout trajectories have infinite length.

One possibility that works well for many problems is to simply set the terminal cost approximation to zero. Alternatively, the terminal cost function approximation may be obtained by using some sophisticated off-line training process that may involve an approximation architecture such as a neural network, or by using some heuristic calculation based on a simplified version of the problem. This form of truncated rollout may also be viewed as an intermediate approach between standard rollout where there is no truncation (and hence no cost function approximation), and approximation in value space without any rollout.

2.3.6 Rollout with an Expert - Model-Free Rollout

We will now consider a rollout algorithm for discrete deterministic optimization for the case where *we do not know the cost function and the constraints of the problem*. Instead we have access to a base heuristic, and also a human or software "expert" who can rank any two feasible solutions without assigning numerical values to them.

We consider the general discrete optimization problem of selecting a control sequence $u = (u_0, \ldots, u_{N-1})$ to minimize a function $G(u)$. For simplicity we assume that each component u_k is constrained to lie in a given constraint set U_k, but extensions to more general constraint sets are possible. We assume the following:

(a) A base heuristic with the following property is available: Given any $k < N-1$, and a partial solution $(u_0, \ldots, u_k)$, it generates, for every $\tilde{u}_{k+1} \in U_{k+1}$, a complete feasible solution by concatenating the given partial solution $(u_0, \ldots, u_k)$ with a sequence $(\tilde{u}_{k+1}, \ldots, \tilde{u}_{N-1})$. This complete feasible solution is denoted

$$S_k(u_0, \ldots, u_k, \tilde{u}_{k+1}) = (u_0, \ldots, u_k, \tilde{u}_{k+1}, \ldots, \tilde{u}_{N-1}).$$

The base heuristic is also used to start the algorithm from an artificial empty solution, by generating all components $\tilde{u}_0 \in U_0$ and a complete feasible solution $(\tilde{u}_0, \ldots, \tilde{u}_{N-1})$, starting from each $\tilde{u}_0 \in U_0$.

(b) An "expert" is available that can compare any two feasible solutions u and $\overline{u}$, in the sense that he/she can determine whether

$$G(u) > G(\overline{u}), \qquad \text{or} \qquad G(u) \leq G(\overline{u}).$$

It can be seen that deterministic rollout can be applied to this problem, even though the cost function G is unknown. The reason is that the rollout algorithm uses the cost function only as a means of ranking complete solutions in terms of their cost. Hence, if the ranking of any two

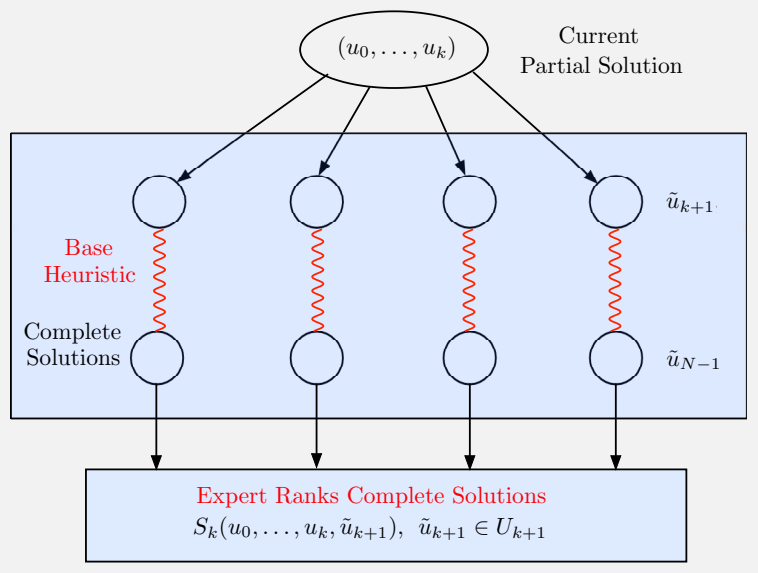

Figure 2.3.7 Schematic illustration of the rollout with an expert for minimizing $G(u)$ subject to

$$u \in U_0 \times \cdots \times U_{N-1}.$$

We assume that we do not know G and/or $U_0, \ldots, U_{N-1}$. Instead we have a base heuristic, which given a partial solution $(u_0, \ldots, u_k)$, outputs all next controls $\tilde{u}_{k+1} \in U_{k+1}$, and generates from each a complete solution

$$S_k(u_0, \ldots, u_k, \tilde{u}_{k+1}) = (u_0, \ldots, u_k, \tilde{u}_{k+1}, \ldots, \tilde{u}_{N-1}).$$

Also, we have a human or software "expert" that can rank any two complete solutions without assigning numerical values to them. The control that is selected from U_{k+1} by the rollout algorithm is the one whose corresponding complete solution is ranked best by the expert.

solutions can be revealed by the expert, this is all that is needed.† In fact, the constraint sets $U_0, \ldots, U_{N-1}$ need not be known either, as long as they can be generated by the base heuristic. Thus, the rollout algorithm can be described as follows (see Fig. 2.3.7):

We start with an artificial empty solution, and at the typical step, given the partial solution $(u_0, \ldots, u_k)$, $k < N-1$, we use the base heuristic

† Note that for this to be true, it is important that the problem is deterministic, and that the expert ranks solutions using some underlying (though unknown) cost function. In particular, the expert's rankings should have a transitivity property: if u is ranked better than u' and u' is ranked better than u'', then u is ranked better than u''.

to generate all possible one-step-extended solutions

$$(u_0, \ldots, u_k, \tilde{u}_{k+1}), \qquad \tilde{u}_{k+1} \in U_{k+1},$$

and the set of complete solutions

$$S_k(u_0, \ldots, u_k, \tilde{u}_{k+1}), \qquad \tilde{u}_{k+1} \in U_{k+1}.$$

We then use the expert to rank this set of complete solutions. Finally, we select the component u_{k+1} that is ranked best by the expert, extend the partial solution $(u_0, \ldots, u_k)$ by adding u_{k+1}, and repeat with the new partial solution $(u_1, \ldots, u_k, u_{k+1})$.

Except for the (mathematically inconsequential) use of an expert rather than a cost function, the preceding rollout algorithm can be viewed as a special case of the one given earlier. As a result several of the rollout variants that we have discussed so far (rollout with multiple heuristics, simplified rollout, and fortified rollout) can also be easily adapted.

Example 2.3.5 (Using a Large Language Model for Rollout)

The problem of minimizing $G(u)$ over a constraint set can be viewed as an N-gram optimization problem, discussed in Example 1.6.2, where the text window consists of a partial solution $(u_0, \ldots, u_k, u_{k+1})$ (preceded by $N-k-2$ "default" words to bring the total to N). We noted in that example that a GPT can be used as a policy that generates next words within the context of N-gram optimization. Thus the problem can be addressed within the model-free rollout framework of this section, whereby a GPT is used as a base heuristic for completion of partial solutions. The main issues with this approach are how to train a GPT for the problem at hand, and also in the absence of an explicit cost function $G(u)$, how to properly design the expert software for comparing complete solutions. Both of these issues are actively researched at present.

Example 2.3.6 (RNA Folding)

In a classical problem from computational biology, we are given a sequence of nucleotides, represented by circles in Fig. 2.3.8, and we want to "fold" the sequence in an "interesting" way (introduce pairings of nucleotides that result in an "interesting" structure). There are some constraints on which pairings are possible, but we will not go into the details of this (some types of constraints may require the use of the constrained rollout framework of Section 2.5). A common constraint is that the pairings should not "cross," i.e., given a pairing (i_1, i_2) there should be no pairing (i_3, i_4) where either $i_3 < i_1$ and $i_1 < i_4 < i_2$, or $i_1 < i_3 < i_2$ and $i_2 < i_4$. This type of problem has a long history of solution by DP, starting with the paper by Zuker and Stiegler [ZuS81]. There are several formulations, where the aim is to optimize some criterion, e.g., the number of pairings, or the "energy" of the folding. However, biologists do not agree on a suitable criterion, and have developed

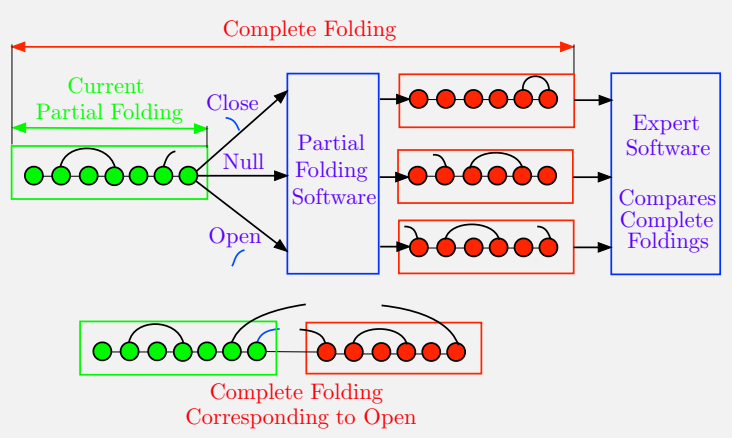

Figure 2.3.8 Schematic illustration of rollout for the RNA folding problem. The current state is the partial folding depicted on the left side. There are at most three choices for control at each state.

software to generate "reasonable" foldings, based on semi-heuristic reasoning. We will develop a rollout approach that makes use of such software without discussing their underlying principles.

We formulate the folding problem as a discrete optimization problem involving a pairing decision at each nucleotide in the sequence with at most three choices (open a pairing, close a pairing, do nothing); see Fig. 2.3.8. To apply rollout, we need a base heuristic, which given a partial folding, generates a complete folding (this is the *partial folding software* shown in Fig. 2.3.8). Two complete foldings can be compared by some other software, called the *expert software*. An interesting aspect of this problem is that there is no explicit cost function here (it is internal to the expert software). Thus by trying different partial folding and expert software, we may obtain multiple solutions, which may be used for further screening and/or experimental evaluation. For a recent implementation and variations, see Liu et al. [LPS21].

One more aspect of the problem that is worth noting is that there are at most three choices for control at each state, while the problem is deterministic. As a result, the problem is a good candidate for the use of multistep lookahead. In particular, with ℓ-step lookahead, the number of Q-factors to be computed at each state increases from 3 (or less) to 3^ℓ (or less).

Learning to Imitate the Expert

To implement model-free rollout, we need both a base heuristic and an expert. None of these may be readily available, particularly the expert, which involves a hidden cost function that is implicitly used to rank complete solutions. Within this context, it is worth considering the case where an expert is not available but can be emulated by training with the use of data. In particular, suppose that we are given a set of control sequence

pairs $(u^s, \overline{u}^s)$, $s = 1, \ldots, q$, with

$$G(u^s) > G(\overline{u}^s), \qquad s = 1, \ldots, q, \tag{2.19}$$

which we can use for training. Such a set may be obtained in a variety of ways, including querying the expert. We may then train a parametric approximation architecture such as a neural network to produce a function $\tilde{G}(u, r)$, where r is a parameter vector, and use this function in place of the unknown $G(u)$ to implement the preceding rollout algorithm.

A method, known as *comparison training*, has been suggested for this purpose, and has been used in a variety of game contexts, including backgammon and chess by Tesauro [Tes89b], [Tes01]. Briefly, given the training set of pairs $(u^s, \overline{u}^s)$, $s = 1, \ldots, q$, which satisfy Eq. (2.19), we generate for each $(u^s, \overline{u}^s)$, two solution-cost pairs $(u^s, 1)$, $(\overline{u}^s, -1)$, $s = 1, \ldots, q$. A parametric architecture $\tilde{G}(\cdot, r)$, involving a parameter vector r, such as a neural network, is then trained by some form of regression with these data to produce an approximation $\tilde{G}(\cdot, \bar{r})$ to be used in place of $G(\cdot)$ in a rollout scheme. We refer to Chapter 3 and to the aforementioned papers by Tesauro for implementation details of the regression procedure. See also Section 3.4 on parametric approximation in policy space through the use of classification methods.

Learning the Base Policy's Q-Factors

In another type of imitation approach, we view the base policy decisions as being selected by a process the mechanics of which are not observed except through its generated cost samples at the various stages. In particular, the stage costs starting from any given partial solution $(u_0, \ldots, u_k)$ are added to form samples of the base policy's Q-factors $Q_k(u_0, \ldots, u_k)$. In this way we can obtain Q-factor samples starting from many partial solutions $(u_0, \ldots, u_k)$. Moreover, a single complete solution $(u_0, \ldots, u_{N-1})$ generated by the base policy provides multiple Q-factor samples, one for each of the partial solutions $(u_0, \ldots, u_k)$.

We can then use the sample (partial solution, cost) pairs in conjunction with a training method (see Chapter 3) in order to construct parametric approximations $\tilde{Q}_k(u_0, \ldots, u_k, r_k)$, $k = 1, \ldots, N$, to the true Q-factors $Q_k(u_0, \ldots, u_k)$, where r_k is the parameter vector. Once the training has been completed and the Q-factors $\tilde{Q}_k(u_0, \ldots, u_k, r_k)$ have been obtained for all k, we can construct complete solutions step-by-step, by selecting the next component $\tilde{u}_{k+1}$, given the partial solution $(u_0, \ldots, u_k)$, through the minimization

$$\tilde{u}_{k+1} \in \arg \min_{u_{k+1} \in U_{k+1}} \tilde{Q}_{k+1}(\tilde{u}_0, \ldots, \tilde{u}_k, u_{k+1}, r_{k+1}).$$

Note that even though we are "learning" the base policy, our aim is not to imitate it, but rather to generate a rollout policy. The latter policy

will make better decisions than the base policy, thanks to the cost improvement property of rollout. This points to an important issue of *exploration*: we must ensure that the training set of sample (partial solution, cost) pairs is broadly representative, in the sense that it is not unduly biased towards sample pairs that are generated by the base policy.

2.3.7 Most Likely Sequence Generation for n-Grams, HMMs, and Markov Chains

In this section we consider a type of deterministic sequential decision problem involving n-grams and transformers, which provide next word probabilities that can be used to generate word sequences (cf. Section 1.6, Example 1.6.2). We consider methods for computing N-step word sequences that are highly likely, based on these probabilities.† Computing the optimal (i.e., most likely) word sequence starting with a given initial state is an intractable problem, so we consider rollout algorithms that compute highly likely N-word sequences in time that is a low order polynomial in N and in the vocabulary size of the n-gram.

Our n-gram model generates a sequence $\{x_1, \ldots, x_N\}$ of text strings, starting from some initial string x_0 (here n and N are fixed positive integers). Each string x_k consists of a sequence of n words, chosen from a given list (the *vocabulary* of the n-gram). The kth string x_k is transformed into the next string x_{k+1} by adding a word at the front end of x_k and deleting the word at the back end of x_k; see Example 1.6.2.

Given a text string x_k, the n-gram provides probabilities $p(x_{k+1} \mid x_k)$ for the next text string x_{k+1}. These probabilities also define the probabilities of the possible next words, since x_{k+1} is determined by the next word that is added to the front of x_k. We assume that the probabilities $p(x_{k+1} \mid x_k)$ depend only on x_k. Thus they can be viewed as the transition probabilities of a stationary Markov chain, whose state space is the set of all n-word sequences x_k.‡ Bearing this context in mind, we also refer to x_k as the *state* (of the underlying Markov chain).

The transition probabilities $p(x_{k+1} \mid x_k)$ can provide guidance for generating state sequences with some specific purpose in mind. To this end, a transformer may use a (next word) *selection policy*, i.e., a (possibly time-dependent) function μ_k, which selects the text string that follows x_k as

$$x_{k+1} = \mu_k(x_k).$$

† This section is based on joint work with Yuchao Li; see the paper by Li and Bertsekas [LiB24], which also contains extensive computational experimentation results.

‡ The stationarity assumption simplifies our notation, but is not essential to our methodology, as we will discuss later.

We are generally interested in selection policies that give preference to high-probability future words.

Our methods also apply to the problem of finding the most likely sequence generated by a general finite-state Markov chain. This problem arises in many important contexts. A major example is inference of the sequence of states of a Hidden Markov Model (HMM), given an associated sequence of observed data. This is the problem where Viterbi decoding [Vit67], [For73], and related algorithms are used widely, and it plays an important role in several diverse fields, such as speech recognition [Rab69], [EpM02], computational linguistics and language translation [JuM23], [MaS99], coding and error correction [PrS01], [PrS08], bioinformatics [Edd96], [DEK98], computational finance [MaE07], and others; see the edited volumes by Bouguila, Fan, and Amayri [BFA22], Mamon and Elliott [MaE14], and Westhead and Vijayabaskar [WeV17]. Compared to these fields, the transformer/n-gram context tends to involve Markov chains with an intractably larger state space. A DP-oriented discussion of the Viterbi algorithm and its applications to HMM inference is given in Section 2.2.2 of the textbook [Ber17]. In this section, we will not consider the problem of most likely sequence selection in the context of inference of state sequences in HMMs, but our methods fully apply to that context.

Computing the Most Likely Sequence in a Markov Chain

We will next consider a finite-state stationary Markov chain and various policies for generating highly likely sequences according to the transition probabilities of the Markov chain. We will generally use the symbols x and y for states, and we will denote the chain's transition probabilities by $p(y \mid x)$. We assume that given a state x, the probabilities $p(y \mid x)$ are either known or can be generated on-line by means of software such as a transformer.

We assume stationarity of the Markov chain in part to alleviate an overburdened notation, and also because n-gram and transformer models are typically assumed to be stationary. However, *the rollout methodology and the manner in which we use it do not depend at all on stationarity* of the transition probabilities, or infinite horizon properties of Markov chains, such as ergodic classes, transient states, etc. In fact, they also do not depend on the stationarity of the state space either. Only the Markov property is used in our discussion, i.e., the probability of the next state depends on the immediately preceding state, and not on earlier states.

A *selection policy* π is a sequence of functions $\{\mu_0, \ldots, \mu_{N-1}\}$, which given the current state x_k, determines the next state x_{k+1} as

$$x_{k+1} = \mu_k(x_k).$$

Note that for a given π, the state evolution is *deterministic*; so for a given π and x_0, the generated state sequence $\{x_1, \ldots, x_N\}$ is fully determined.

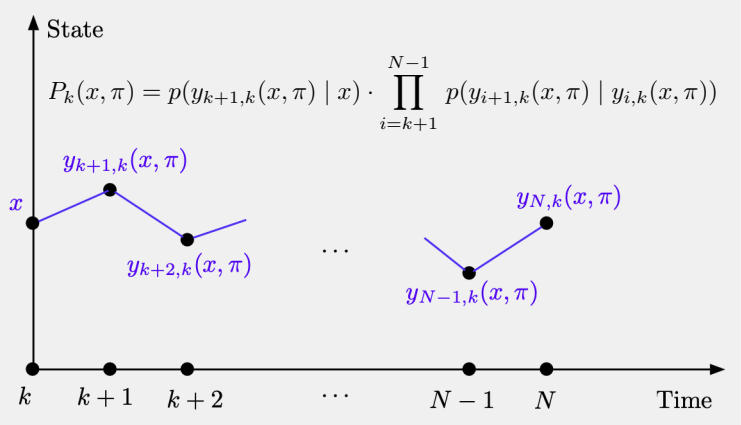

Figure 2.3.9 Illustration of the state trajectory generated by a policy π, starting at state x at time k. The probability of its occurrence, $P_k(x,\pi)$, is the product of the transition probabilities along the $N-k$ steps of the trajectory [cf. Eq. (2.20)].

Moreover the choice of the policy π is arbitrary, although we are primarily interested in π that give preference to high probability next states.

Given a policy $\pi = \{\mu_0, \ldots, \mu_{N-1}\}$ and a starting state x at time k, the state at future times $m > k$ is denoted by $y_{m,k}(x,\pi)$:

$$y_{m,k}(x,\pi) = \text{ state at time } m > k \text{ starting at state } x \text{ and using } \pi.$$

The *state trajectory generated by a policy π, starting at state x at time k*, is the sequence

$$y_{k+1,k}(x,\pi), \ldots, y_{N,k}(x,\pi),$$

(cf. Fig. 2.3.9), and the probability of its occurrence in the given Markov chain is

$$P_k(x,\pi) = p\big(y_{k+1,k}(x,\pi) \mid x\big) \cdot \prod_{i=k+1}^{N-1} p\big(y_{i+1,k}(x,\pi) \mid y_{i,k}(x,\pi)\big), \quad (2.20)$$

according to the multiplication rule for conditional probabilities.

Optimal/Most Likely Selection Policy

The *most likely selection policy*, denoted by $\pi^* = \{\mu_0^*, \ldots, \mu_{N-1}^*\}$, maximizes over all policies π the probabilities $P_k(x,\pi)$ for every initial state x and time k. The corresponding probabilities of π^*, starting at state x at time k, are denoted by $P_k^*(x)$:

$$P_k^*(x) = P_k(x,\pi^*) = \max_{\pi} P_k(x,\pi).$$

This is similar to our finite-horizon DP context, except that we consider a multiplicative reward function, instead of an additive cost function [$P_k^*(x)$ can be viewed as an optimal reward-to-go from state x at time k, in place of the optimal cost-to-to $J_k^*(x)$ that we have considered so far in the additive DP case].†

The policy π^* and its probabilities $P_k^*(x)$ can be generated by the following DP-like algorithm, which operates in two stages:

(a) It first computes the probabilities $P_k^*(x)$ backwards, for all x, according to

$$P_k^*(x) = \max_y p(y \mid x) P_{k+1}^*(y), \qquad k = N-1, \ldots, 0, \tag{2.21}$$

starting with

$$P_N^*(x) \equiv 1.$$

(b) It then generates sequentially the selections $x_1^*, \ldots, x_N^*$ of π^* according to

$$x_{k+1}^* = \mu_k^*(x_k^*) \in \arg\max_y p(y \mid x_k^*) P_{k+1}^*(y), \tag{2.22}$$

starting with $x_0^* = x_0$.

This algorithm is equivalent to the usual DP algorithm for multistage additive costs, after we take logarithms of the multiplicative expressions defining the probabilities $P_k(x, \pi)$.

Greedy Policy

At any given state x_k, the *greedy policy* produces the next state by maximization of the corresponding transition probability over all y:

$$\max_y p(y \mid x_k).$$

We assume that ties in the above maximization are broken according to some prespecified deterministic rule. For example if the states are labeled by distinct integers, one possibility is to specify the greedy selection at x_k as the state y with minimal label, among those that attain the maximum above. Note that the greedy policy is not only deterministic, but it is also stationary (its selections depend only on the current state and not on the time k). We will consequently use the notation $\overline{\pi} = \{\overline{\mu}, \ldots, \overline{\mu}\}$ for the greedy policy, where

$$\overline{\mu}(x_k) \in \arg\max_y p(y \mid x_k), \tag{2.23}$$

† Generally multiplicative reward problems can be converted to additive cost DP problems involving negative logarithms of the multiplicative reward factors (assuming they are positive).

and $\overline{\mu}(x_k)$ is uniquely defined according to our deterministic convention for breaking ties in the maximization above. The corresponding probabilities $P_k(x_k, \overline{\pi})$ are given by the DP-like algorithm

$$P_k(x, \overline{\pi}) = p\big(\overline{\mu}(x) \mid x\big) P_{k+1}\big(\overline{\mu}(x), \overline{\pi}\big), \qquad k = N-1, \ldots, 0, \tag{2.24}$$

starting with

$$P_N(x, \overline{\pi}) \equiv 1.$$

Equivalently, we can compute $P_k(x, \overline{\pi})$ by using forward multiplication of the transition probabilities along the trajectory generated by the greedy policy, starting from x; cf. Eq. (2.20).

The limitation of the greedy policy is that it chooses the locally optimal next state without considering the impact of this choice on future state selections. The rollout approach, to be discussed next, mitigates this limitation with a mechanism for looking into the future, and balancing the desire for a high-probability next transition with the potential undesirability of low-probability future transitions.

Rollout Policy

At any given state x_k, the *rollout policy with one-step lookahead* produces the next state, denoted $\tilde{\mu}_k(x_k)$, by maximizing $p(y \mid x_k)P_{k+1}(y, \overline{\pi})$ over all y:

$$\tilde{\mu}_k(x_k) \in \arg\max_y p(y \mid x_k) P_{k+1}(y, \overline{\pi}). \tag{2.25}$$

Thus *it optimizes the selection of the first state y, assuming that the subsequent states will be chosen using the greedy policy.*

By comparing the maximization (2.25) with the one for the most likely selection policy [cf. Eq. (2.22)], we see that it chooses the next state similarly, except that $P^*_{k+1}(y)$ (which is hard to compute) is replaced by the (much more easily computable) probability $P_{k+1}(y, \overline{\pi})$. In particular, the latter probability is computed for every y by running the greedy policy forward starting from y and multiplying the corresponding transition probabilities along the generated state trajectory. This is a polynomial computation, which is roughly larger by a factor N over the greedy selection method. However, there are ways to reduce this computation, including the use of parallel processing and other possibilities, which we will discuss later.

The expression $p(y \mid x_k)P_{k+1}(y, \overline{\pi})$ that is maximized over y in Eq. (2.25) can be viewed as the *Q-factor of the pair (x_k, y) corresponding to the base policy $\overline{\pi}$*, and is denoted by $Q_{\overline{\pi},k}(x_k, y)$:

$$Q_{\overline{\pi},k}(x_k, y) = p(y \mid x_k) P_{k+1}(y, \overline{\pi}). \tag{2.26}$$

This is similar to the approximation in value space context, except that we consider a multiplicative reward function, whereby at state x_k we choose the action y that yields the maximal Q-factor.

Rollout Policy with ℓ-Step Lookahead

Another rollout possibility includes *rollout with ℓ-step lookahead* ($\ell > 1$), whereby given x_k we maximize over all sequences $\{y_1, y_2, \ldots, y_\ell\}$ up to ℓ steps ahead, the *ℓ-step Q-factor*

$$Q_{\overline{\pi},k,\ell}(x_k, y_1, \ldots, y_\ell) = p(y_1 \mid x_k)p(y_2 \mid y_1) \cdots p(y_\ell \mid y_{\ell-1})P_{k+\ell}(y_\ell, \overline{\pi}), \tag{2.27}$$

and if $\{\tilde{y}_1, \tilde{y}_2, \ldots, \tilde{y}_\ell\}$ is the maximizing sequence, we select $\tilde{y}_1$ at x_k, and discard the remaining states $\tilde{y}_2, \ldots, \tilde{y}_\ell$.† In practice the performance of ℓ-step lookahead rollout policies almost always improves with increasing ℓ. However, artificial examples have been constructed where this is not so; see the book [Ber19a], Section 2.1.1. Moreover, the computational overhead of ℓ-step lookahead increases with ℓ.

Simplified and Truncated Rollout

As we have already noted, one of the difficulties that arises in the application of rollout is the potentially very large number of the Q-factors that need to be calculated at each time step at the current state x [it is equal to the number of states y for which $p(y \mid x) > 0$]. In practice the computation of Q-factors can be restricted to a subset of most probable next states, as per the transition probabilities $p(y \mid x)$ (this is similar to simplified rollout that we discussed in Section 2.3.4). For example, often many of the transition probabilities $p(y \mid x)$ are very close to 0, and can be safely ignored.

Another possibility to reduce computation is to truncate the trajectories generated from the next states y by the greedy policy, up to m steps (assuming that $k + m < N$, i.e., if we are more than m steps away from the end of the horizon). This is essentially the truncated rollout algorithm, discussed in Section 2.3.5, whereby we maximize over y the m-step Q-factor of the greedy policy $\overline{\pi}$:

$$p(y \mid x_k)P_{k+1,m}(y, \overline{\pi}), \tag{2.28}$$

where

$$P_{k+1,m}(y, \overline{\pi}) = p\big(y_{k+2,k+1}(y, \overline{\pi}) \mid y\big) \cdot \prod_{i=k+2}^{k+m} p\big(y_{i+1,k+1}(y, \overline{\pi}) \mid y_{i,k+1}(y, \overline{\pi})\big)$$

is the m-step product of probabilities along the path generated by the greedy policy $\overline{\pi}$ starting from y at time $k+1$ [cf. Eq. (2.20)]. By contrast, in rollout without truncation, we maximize over y

$$p(y \mid x_k)P_{k+1}(y, \overline{\pi}),$$

† If $\ell > N - k$, then ℓ must be reduced to $N - k$, to take into account end-of-horizon effects.

where

$$P_{k+1}(y,\overline{\pi}) = p\big(y_{k+2,k+1}(y,\overline{\pi}) \mid y\big) \cdot \prod_{i=k+2}^{N-1} p\big(y_{i+1,k+1}(y,\overline{\pi}) \mid y_{i,k+1}(y,\overline{\pi})\big)$$

is the $(N-k-1)$-step product of probabilities along the path generated by the greedy policy starting from y at time $k+1$; cf. Eqs. (2.20) and (2.25).

Multiple On-line Policy Iterations - Double Rollout

Still another possibility is to apply the rollout approach successively, in *multiple policy iterations*, by using the rollout policy obtained at each iteration as base policy for the next iteration. This corresponds to the fundamental DP algorithm of *policy iteration*.

Performing on-line just two policy iterations amounts to using the rollout algorithm as a base policy for another rollout algorithm. This has been called *double rollout*, and it has been discussed in Section 2.3.5 of the book [Ber20] and Section 6.5 of the book [Ber22]. Generally, one-step lookahead rollout requires $O(q \cdot N)$ applications of the base policy where q is the number of Q-factors calculated at each time step.† Thus with each new policy iteration, there is an amplification factor $O(q \cdot N)$ of the computational requirements. Still, however, the multiple iteration approach may be viable, even on-line, when combined with some of the other time-saving computational devices described above (e.g., truncation and simplification to reduce q), in view of the relative simplicity of the calculations involved and their suitability for parallel computation. This is particularly so for double rollout. Policy iteration/double rollout is discussed by Yan et al. [YDR04] in the context of the game of solitaire, and by Silver and Barreto [SiB22] in the context of a broader class of search methods.

We next show that the rollout selection policy with one-step lookahead has a *performance improvement property*: it generates more likely state sequences than the greedy policy, starting from any state, and the improvement is often very substantial.

† For a more accurate estimate of the complexity of the greedy, rollout, and double rollout algorithms, note that the basic operation of the greedy operation is the maximization over the q numbers $p(y \mid x_k)$. Thus m steps of the greedy algorithm, as in an m-step Q-factor calculation, costs $q \cdot m$ comparisons. In m-step truncated rollout, we compare q greedy Q-factors so the number of comparisons per rollout time step is $q^2m + q$. Over N time steps the total is $(q^2m + q) \cdot N$ comparisons, while for the greedy algorithm starting from the initial state x_0, the corresponding number is $q \cdot N$. Thus there is an amplification factor of $qm+1$ for the computation of simplified m-step truncated rollout over the greedy policy. Similarly it can be estimated that there is an amplification factor of no more than $qm + 1$ for using double rollout with (single) rollout as a base policy.

Performance Improvement Properties of Rollout Policies

We will show by induction a performance improvement property of the rollout algorithm with one-step lookahead, namely that for all states $x \in X$ and k, we have

$$P_k(x, \overline{\pi}) \leq P_k(x, \tilde{\pi}), \tag{2.29}$$

i.e., the probability of the sequence generated by the rollout policy is greater or equal to the probability of the sequence generated by the greedy policy; this is true for any starting state x at any time k. This is similar to our earlier rollout cost improvement results in Section 2.3.1.

Indeed, for $k = N$ this relation holds, since we have

$$P_N(x, \overline{\pi}) = P_N(x, \tilde{\pi}) \equiv 1.$$

Assuming that

$$P_{k+1}(x, \overline{\pi}) \leq P_{k+1}(x, \tilde{\pi}), \qquad \text{for all } x,$$

we will show that

$$P_k(x, \overline{\pi}) \leq P_k(x, \tilde{\pi}), \qquad \text{for all } x.$$

Indeed, we use the preceding relations to write

$$\begin{aligned} P_k(x, \tilde{\pi}) &= p\big(\tilde{\mu}_k(x) \mid x\big) P_{k+1}\big(\tilde{\mu}_k(x), \tilde{\pi}\big) \\ &\geq p\big(\tilde{\mu}_k(x) \mid x\big) P_{k+1}\big(\tilde{\mu}_k(x), \overline{\pi}\big) \\ &\geq p\big(\tilde{\mu}_k(x) \mid x\big) P_{k+1}\big(\overline{\mu}_k(x), \overline{\pi}\big) \\ &= P_k(x, \overline{\pi}), \end{aligned}$$

where

- The first equality holds from the definition of the probabilities corresponding to the rollout policy $\tilde{\pi}$.
- The first inequality holds from the definition by the induction hypothesis.
- The second inequality holds from the fact that the rollout choice $\tilde{\mu}_k(x)$ maximizes the Q-factor $p\big(\tilde{y} \mid x\big) P_{k+1}\big(y, \overline{\pi}\big)$ over y.
- The second equality holds from the definition of the probabilities corresponding to the greedy policy $\overline{\pi}$.

Thus the induction proof of the improvement property (2.29) is complete.

Clearly, the performance improvement property continues to hold in successive multiple iterations of the rollout policy, and in fact it can be shown that after a sufficiently large number of iterations it yields the most

likely selection policy. This is a consequence of classical policy iteration convergence results, which establish the finite convergence of the policy iteration algorithm for finite-state Markovian decision problems, see e.g., [Ber12], [Ber17a], [Ber19a].

Performance improvement can also be established for the ℓ-step lookahead version of the rollout policy, using an induction proof that is similar to the one given above for the one-step lookahead case. Moreover, the books [Ber20a], [Ber22a] describe conditions under which simplified rollout maintains the performance improvement property. However, it is not necessarily true that the performance of the ℓ-step lookahead rollout policy improves as ℓ increases; see an example in the book [Ber19a], Section 2.1.1. Similarly, it is not necessarily true that the m-step truncated rollout policy performs better than the greedy policy.† On the other hand, known performance deterioration examples of this type are artificial and are apparently rare in practice.

Computational Comparison of Greedy, Optimal, and Rollout Policies

We will now present some illustrative computational comparisons, using Markov chains that are small enough for the optimal/most likely selection policy to be computed exactly via the DP-like algorithm (2.21)-(2.22). Thus, the performance differences between the rollout, greedy, and optimal policies can be accurately assessed. The experiments are presented in more detail in the paper [LiB24], which also contains qualitatively similar computational results with much larger Markov chains involving n-grams, and a trained transformer neural network. We used Markov chains where there is a fixed number q of distinct states y such that $p(y \mid x) > 0$, with q being the same for all states x. These states were selected according to a uniform distribution, whereby all y with $p(y \mid x) > 0$ are equally likely. The probabilities $p(y \,|\, x)$ were also generated according to a uniform distribution.

Let us denote the state space by X and the number of states by $|X|$. We refer to the ratio $q/|X|$ (in percent) as the *branching factor* of the chain. We represent the probability of an entire sequence generated by a policy as the average of its constituent transition probabilities (i.e., a geometric mean over N as will be described below). In particular, given a sample set C of Markov chains, we compute the optimal occurrence probability of generated sequences, averaged over all chains, states, and transitions, and

† It performs better than an m-step version of the greedy policy, which generates a sequence of $m + 1$ states, starting from the current state and using the greedy policy.

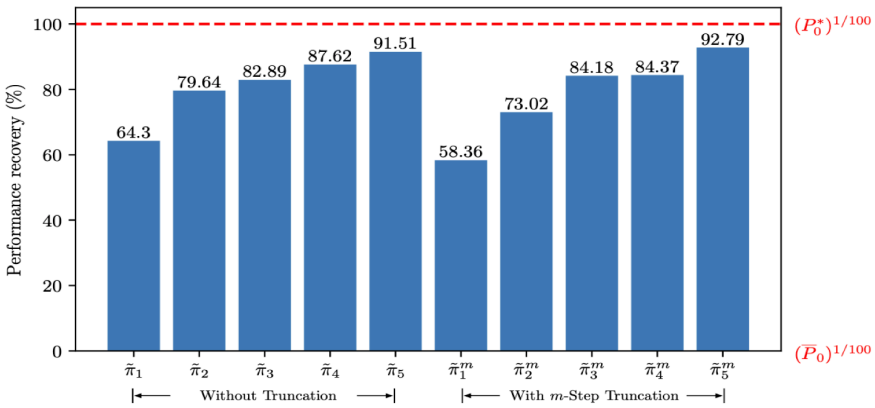

Figure 2.3.10 Percentage recovery of the optimality loss of the greedy policy through the use of rollout and its variants, applied to sequence selection problems with $N = 100$ for 50 randomly generated Markov chains with 100 states and 5% branching factor. Here $\tilde{\pi}_\ell$ is rollout with ℓ-step lookahead, and $\tilde{\pi}_\ell^m$ represents m-step truncated rollout for $m = 10$ with ℓ-step lookahead. It can be seen that on average, rollout and its variants provide a substantial improvement over the greedy policy, the improvement increases with the size of the lookahead, and truncated rollout methods perform comparable to their exact counterparts.

denoted by $(P_0^*)^{1/N}$, according to the average geometric mean formula

$$(P_0^*)^{1/N} = \frac{\sum_{c\in C}\sum_{x\in X}\left(P_{0,c}^*(x)\right)^{1/N}}{|C|\cdot|X|},$$

where $P_{0,c}^*(x)$ is the optimal occurrence probability with $x_0 = x$ and Markov chain c in the sample set. Similarly, we compute the occurrence probabilities of sequences generated by the greedy policy averaged over all chains, states, and transitions, and denoted by $(\overline{P}_0)^{1/N}$, according to

$$(\overline{P}_0)^{1/N} = \frac{\sum_{c\in C}\sum_{x\in X}\left(P_{0,c}(x,\overline{\pi})\right)^{1/N}}{|C|\cdot|X|}, \tag{2.30}$$

where $P_{0,c}(x,\overline{\pi})$ is the transition probability of the sequence generated by the greedy policy with $x_0 = x$ and Markov chain indexed by c. For the rollout algorithm (or variants thereof), we compute its averaged occurrence probability $(\tilde{P}_0)^{1/N}$ similar to Eq. (2.30) with $\tilde{\pi}$ in place of $\overline{\pi}$. Then the performance of the rollout algorithm can be measured by its *percentage recovery* of the optimality loss of the greedy policy, given by

$$\frac{(\tilde{P}_0)^{1/N} - (\overline{P}_0)^{1/N}}{(P_0^*)^{1/N} - (\overline{P}_0)^{1/N}} \times 100 \quad (\%). \tag{2.31}$$

This performance measure describes accurately how the rollout performance compares with the greedy policy and how close it comes to optimality.

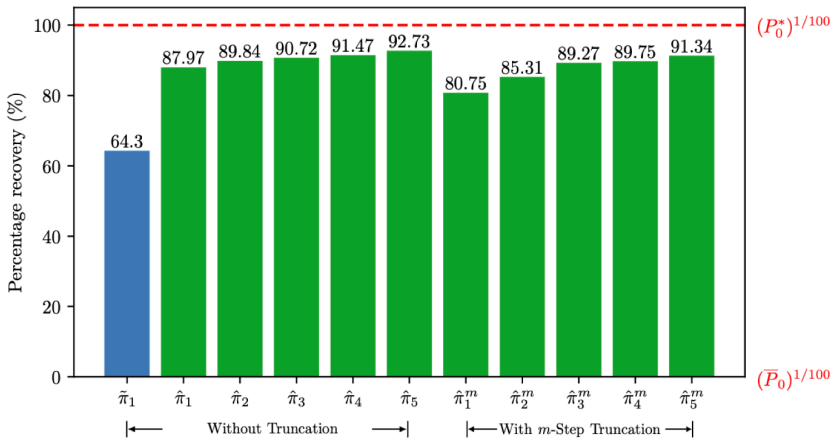

Figure 2.3.11 Percentage recovery of the optimality loss of the greedy policy through the use of double rollout and its variants, applied to sequence selection problems with $N = 100$ for 50 randomly generated Markov chains with 100 states and 5% branching factor. Here $\tilde{\pi}_\ell$ is rollout with ℓ-step lookahead, and $\tilde{\pi}_\ell^m$ represents m-step truncated rollout for $m = 10$ with ℓ-step lookahead.

In the experiments presented here we have used small Markov chains with $|X| = 100$ states, branching factor $q = 5\%$, and sequence length $N = 100$. In summary, the percentage recovery has ranged roughly from 60% to 90% for one-step to five-step lookahead, untruncated and truncated rollout with $m = 10$ steps up to truncation; see Fig. 2.3.10. The performance improves as the length of the lookahead increases, but seems remarkably unaffected by the 90% truncation of the rollout horizon (the relative insensitivity of the performance of truncated rollout to the number of rollout steps m has been observed in other application contexts as well). The figure has been generated with a sample of 50 different Markov chains with $|X| = 100$ states, branching factor equal to 5%, and sequence length $N = 100$. The figure shows the results obtained by rollout with one-step and multi-step lookahead (ranging from 2 to 5 steps), and their m-step truncated counterparts with $m = 10$. Their percentage recovery, evaluated according to Eq. (2.31), is given in Fig. 2.3.10, where $\tilde{\pi}_\ell$ denotes rollout with ℓ-step lookahead, and $\tilde{\pi}_\ell^m$ denotes m-step truncated rollout with ℓ-step lookahead.

Figure 2.3.11 provides corresponding results using double rollout, which show a significant improvement over the case of single rollout. Moreover, truncating the rollout horizon by 90% has remarkably small effect on the percentage recovery, similar to the case of a single rollout.

The preceding results are consistent with those of other computational studies using rollout and its variants. The sequences produced by rollout

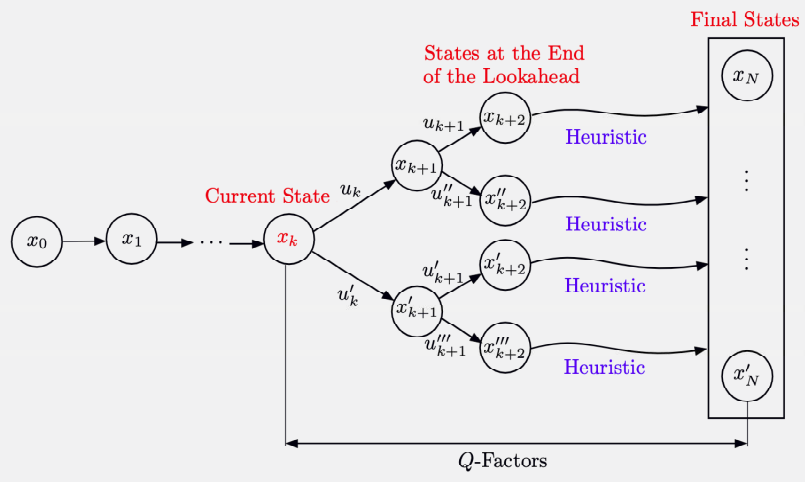

Figure 2.4.1 Illustration of multistep rollout with $\ell = 2$ for deterministic problems. We run the base heuristic from each leaf $x_{k+\ell}$ at the end of the lookahead graph. We then construct an optimal solution for the lookahead minimization problem, where the heuristic cost is used as terminal cost approximation. We thus obtain an optimal ℓ-step control sequence through the lookahead graph, use the first control in the sequence as the rollout control, discard the remaining controls, move to the next state, and repeat. Note that the multistep lookahead minimization may involve approximations aimed at simplifying the associated computations.

improve substantially over those generated by the base policy. Moreover, there is typically a relatively small degradation of performance when applying the truncated rollout compared with untruncated rollout. This is significant as truncated rollout greatly reduces the computation if m is substantially smaller than N, and also makes possible the use of double rollout.

2.4 ROLLOUT AND APPROXIMATION IN VALUE SPACE WITH MULTISTEP LOOKAHEAD

We will now consider approximation in value space with multistep lookahead minimization, possibly also involving some form of rollout. Figure 2.4.1 describes the case of pure (nontruncated) form of rollout with two-step lookahead for deterministic problems. In particular, suppose that after k steps we have reached state x_k. We then consider the set of all possible 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

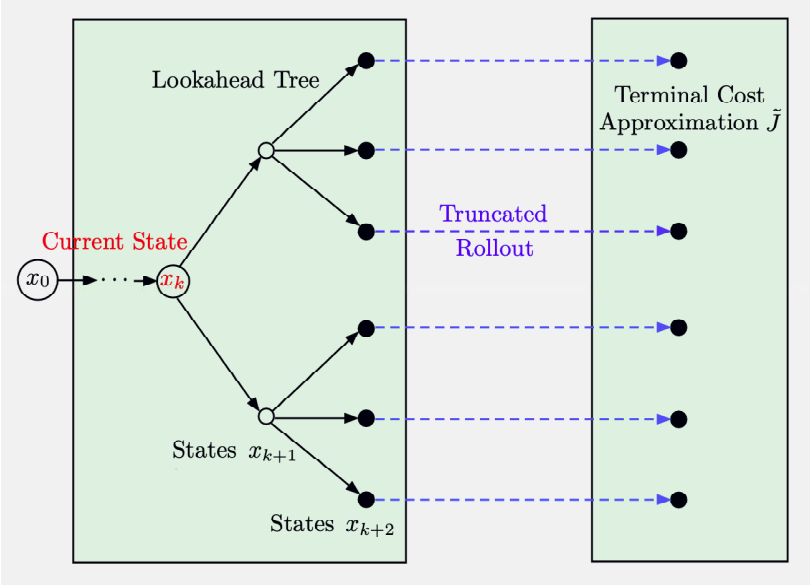

Figure 2.4.2 Illustration of truncated rollout with two-step lookahead and a terminal cost approximation $\tilde{J}$. The base heuristic is used for a limited number of steps and the terminal cost is added to compensate for the remaining steps.

lead from x_k to $\tilde{x}_{k+2}$, choose $\tilde{u}_k$ as the next control and $x_{k+1} = f_k(x_k, \tilde{u}_k)$ as the next state, and discard $\tilde{u}_{k+1}$.

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 cases where the ℓ-step lookahead minimization is very time consuming, we may consider variants involving approximations aimed at simplifying the associated computations.

An important variation is *truncated rollout with terminal cost approximation*. Here the rollout trajectories are obtained by running the base heuristic from the leaf nodes of the lookahead graph, and they are truncated after a given number of steps, while a terminal cost approximation is added to the heuristic cost to compensate for the resulting error; see Fig. 2.4.2. One possibility that works well for many problems, particularly when the combined lookahead for minimization and base heuristic simulation is long, is to simply set the terminal cost approximation to zero. Alternatively, the terminal cost function approximation can be obtained by problem approximation or by using some sophisticated off-line training process that may involve an approximation architecture such as a neural network. Generally, the terminal cost approximation is especially important if a large portion of the total cost is incurred upon termination (this is true for example in games).

Note that the preceding algorithmic scheme can be viewed as multistep approximation in value space, and *it can be interpreted as a Newton step*, with suitable starting point that is determined by the truncated rollout with the base heuristic, and the terminal cost approximation. This interpretation is possible once the discrete optimal control problem is reformulated to an equivalent infinite horizon SSP problem; cf. the discussion of Sections 1.6.2 and 2.1. Thus the algorithm inherits the fast convergence property of the Newton step, which we have discussed in the context of infinite horizon problems in Section 1.5; see also the book [Ber22a].

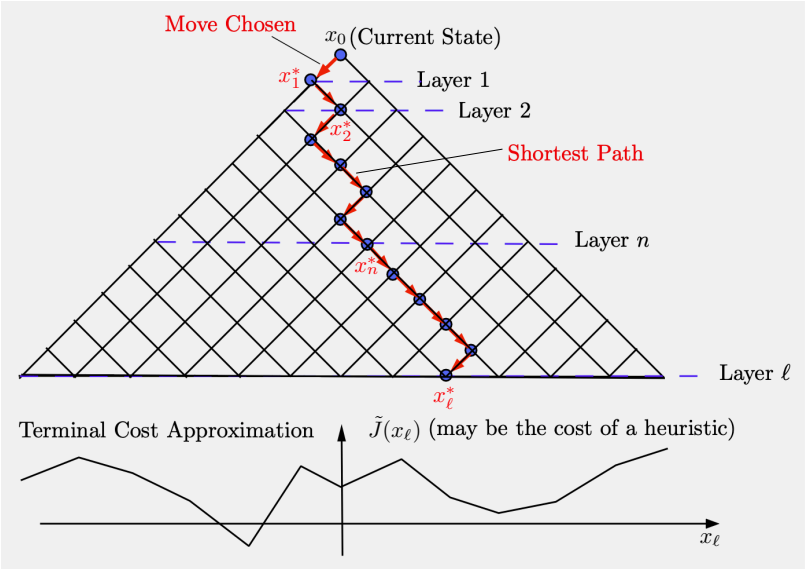

Figure 2.4.3 Illustration of the general ℓ-step approximation in value space scheme with a terminal cost approximation $\tilde{J}$ where x_0 denotes the current state. It involves an acyclic graph of ℓ layers, with layer n, $n = 1, \ldots, \ell$, consisting of all the states x_n that can be reached from x_0 with a sequence of n feasible controls. In ℓ-step approximation in value space, we obtain a trajectory

$$\{x_0, x_1^*, \ldots, x_\ell^*\}$$

that minimizes the shortest distance from x_0 to x_ℓ plus $\tilde{J}(x_\ell)$. We then use the control that corresponds to the first move $x_0 \to x_1^*$.

The architecture of Fig. 2.4.2 contains as a special case the general multistep approximation in value space scheme, where there is no rollout at all; i.e., the leaves of the multistep lookahead tree are evaluated with the function $\tilde{J}$. Figure 2.4.3 illustrates this special case, where for notational simplicity we have denoted the current state by x_0. The illustration involves an acyclic graph with a single root (the current state) and ℓ layers, with

the nth layer consisting of the states x_n that are reachable from x_0 with a feasible sequence of n controls. In particular, there is an arc for every state x_1 of the 1st layer that can be reached from x_0 with a feasible control, and similarly an arc for every pair of states (x_n, x_{n+1}), of layers n and $n+1$, respectively, for which x_{n+1} can be reached from x_n with a feasible control. The cost of each of these arcs is the stage cost of the corresponding state-control pair, minimized over all possible controls that correspond to the same pair (x_n, x_{n+1}). Mathematically, the cost of the arc (x_n, x_{n+1}) is

$$\hat{g}_n(x_n, x_{n+1}) = \min_{\{u_n \in U_n(x_n) \mid x_{n+1} = f_n(x_n, u_n)\}} g_n(x_n, u_n). \tag{2.32}$$

For the states x_ℓ of the last layer there is also the terminal cost approximation $\tilde{J}(x_\ell)$, which may be obtained through off-line training or some other means. It can be thought of as the cost of an artificial arc connecting x_ℓ to an artificial termination state.

Once we have computed all the shortest distances $D(x_\ell)$ from x_0 to all states x_ℓ of the last layer ℓ, we obtain the ℓ-step lookahead control to be applied at the current state x_0, by minimizing over x_ℓ the sum

$$D(x_\ell) + \tilde{J}(x_\ell).$$

If x_ℓ^* is the state that attains the minimum, we generate the corresponding trajectory $(x_0, x_1^*, \ldots, x_\ell^*)$, and then use the control that corresponds to the first move $x_0 \to x_1^*$; see Fig. 2.4.3. Note that the shortest path problems from x_0 to all states x_n of all the layers $n = 1, \ldots, \ell$ can be solved simultaneously by backward DP (start from layer ℓ and go back towards x_0).

Long Lookahead for Deterministic Problems

The architecture of Figs. 2.4.2 and 2.4.3 is similar to the one we discussed in Section 1.1 for AlphaZero and related programs. However, because it is adapted to deterministic problems, it is much simpler to implement and to use. In particular, the truncated rollout portion does not involve expensive Monte Carlo simulation, while the multistep lookahead minimization portion involves a deterministic shortest path problem, which is much easier to solve than its stochastic counterpart. These favorable characteristics can be exploited to facilitate implementations that involve very long lookahead.

Generally speaking, *longer lookahead is desirable because it typically results in improved performance*. We will adopt this as a working hypothesis. It is typically true in practice, although it cannot be established analytically in the absence of additional assumptions.† On the other hand,

† Indeed, there are examples where as the size ℓ of the lookahead becomes longer, the performance of the multistep lookahead policy deteriorates (see [Ber17a], Section 6.1.2, or [Ber19a], Section 2.2.1). However, these examples are isolated and artificial. They are not representative of practical experience.

the on-line computational cost of multistep lookahead increases, often exponentially, with the length of lookahead. We conclude that *we should aim to use a lookahead that is as long as is allowed by the on-line computational budget* (the amount of time that is available for calculating a control to apply at the current state).

Long Lookahead by Rollout is Far More Economical than Long Lookahead Minimization

Our preceding discussion leads to the question of how to economize in computation in order to effectively increase the length of the multistep lookahead within a given on-line computational budget. One way to do this, which we have already discussed, is the use of truncated rollout that explores forward through a deterministic base policy at far less computational cost than lookahead minimization of equal length. As an example, let us consider the possibility of starting with a terminal cost function $\tilde{J}$, possibly generated by off-line training, and use as base policy for rollout the one-step lookahead policy $\tilde{\mu}$, defined by $\tilde{J}$ using the equation†

$$\tilde{\mu}(x) \in \arg\min_{u \in U(x)} \Big[g(x,u) + \tilde{J}\big(f(x,u)\big) \Big]. \tag{2.33}$$

Let us assume that the principal computation in the minimization of Eq. (2.33) is the calculation of $\tilde{J}\big(f(x,u)\big)$, and compare two possibilities:

(a) Using *ℓ-step lookahead minimization with $\tilde{J}$ as the terminal cost approximation without any rollout*; cf. Fig. 2.4.3.

(b) Using *one-step lookahead minimization, with $(\ell-1)$-step truncated rollout and $\tilde{J}$ as the terminal cost approximation.*

Note that scheme (b) is the one used by the TD-Gammon program of Tesauro and Galperin [TeG96], out of necessity: multistep lookahead minimization is very expensive in backgammon, due to the rapid growth of the lookahead graph as ℓ increases (cf. the discussion of Section 1.1).

Suppose that the control set $U(x)$ has m elements for every x. Then the ℓ-step lookahead minimization *scheme (a) requires the calculation of as many as m^ℓ values of $\tilde{J}$*, because the number of leaves of the m-step lookahead graph are as many as m^ℓ.

The corresponding number of calculations of the value of $\tilde{J}$ for scheme (b) is clearly much smaller. To verify this, let us calculate this number as a function of m and ℓ. In particular, the first lookahead stage starting from the current state x_k requires m calculations corresponding to the m controls in $U(x_k)$, and yields corresponding states x_{k+1}, which are as many as m. For each of these states x_{k+1}, we must calculate a sequence of $\ell-1$

† For simplicity, we use stationary system notation, omitting the time subscripts of U, g, and f.

controls using the base policy (2.33) for stages $(k+1)$ to $(k+\ell)$. Each of these $\ell-1$ controls requires m calculations of the value of $\tilde{J}$. Thus, for the $\ell-1$ stages of truncated rollout, there are $m\cdot(\ell-1)$ calculations of the value of $\tilde{J}$ per state x_{k+1}, for a total of as many as $m^2\cdot(\ell-1)$ calculations. Adding the m calculations at state x_k, we conclude that *scheme (b) requires a total of as many as $m^2\cdot\ell$ calculations of the value of $\tilde{J}$.*

In conclusion, *both schemes (a) and (b) above look forward for ℓ stages, but their associated total computation grows exponentially and linearly with ℓ, respectively.* Thus, for a given computational budget, short lookahead minimization with long truncated rollout, can increase the total amount of lookahead and improve the performance of approximation in value space schemes. This is particularly so since based on the Newton step interpretations of approximation in value space of Section 1.5, truncated rollout with a reasonably good (e.g., stable) base policy often works about as well as long lookahead minimization. Extensive computational practice, starting with the rollout/TD-Gammon scheme of [TeG96], is consistent with this assessment.

In the following two sections, we will explore two alternative ways to speed up the lookahead minimization calculation, thereby allowing a larger number ℓ of computational stages for a given on-line computational budget. These are based on *iterative deepening* of the shortest path computation, and *pruning* of the lookahead minimization graph.

2.4.1 Iterative Deepening Using Forward Dynamic Programming

As noted earlier, the shortest path problems from x_0 to x_ℓ in Fig. 2.4.3 can be solved simultaneously by the familiar backward DP that starts from layer ℓ and goes towards x_0. An important alternative for solving these problems is the *forward DP* algorithm. This is the same as the backwards DP algorithm with the *direction of the arcs reversed* (start from x_0 and go towards layer ℓ). In particular, the shortest distances $D_{n+1}(x_{n+1})$ to layer $n+1$ states are obtained from the shortest distances $D_n(x_n)$ to layer n states through the equation

$$D_{n+1}(x_{n+1})=\min_{x_n}\Big[\hat{g}_n(x_n,x_{n+1})+D_n(x_n)\Big],$$

which is also illustrated in Fig. 2.4.4. Here $\hat{g}_n(x_n,x_{n+1})$ is the cost (or length) of the arc (x_n,x_{n+1}); cf. Eq. (2.32).

In particular, the solution of the ℓ-step lookahead problem is obtained from the shortest path to the state x_ℓ^* of layer ℓ that minimizes $D_\ell(x_\ell)+\tilde{J}(x_\ell)$. The idea of iterative deepening is to *progressively solve the n-step lookahead problem first for $n=1$, then for $n=2$, and so on, until our on-line computational budget is exhausted.* In addition to fitting perfectly the mechanism of the forward DP algorithm, this scheme has the character of an *"anytime" algorithm*; i.e., it returns the shortest distances to some

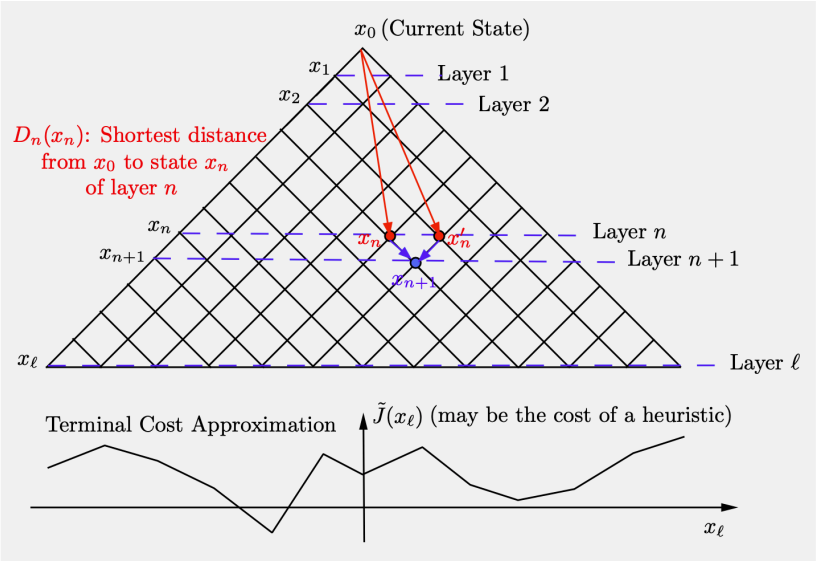

Figure 2.4.4 Illustration of the forward DP algorithm for computing the shortest distances from the current state x_0 to all the states x_n of the layers $n = 1, \ldots, \ell$. The shortest distance $D_{n+1}(x_{n+1})$ to a state x_{n+1} of layer $n+1$ is obtained by minimizing over all predecessor states x_n the sum

$$\hat{g}_n(x_n, x_{n+1}) + D_n(x_n).$$

depth $n \leq \ell$, which can in turn yield a solution of an n-step lookahead minimization after adding a suitable terminal cost function. In practice, this is an important advantage, well known from chess programming, which allows us to keep on aiming for longer lookahead minimization, within the limit imposed by our computational budget constraint.

Iterative Deepening Combined with Pruning

A principal difficulty in approximation in value space with ℓ-step lookahead stems from the rapid expansion of the lookahead graph as ℓ increases. One way to mitigate this difficulty is to "prune" the lookahead minimization graph, i.e., to delete some of its arcs in order to expedite the shortest path computations from the current state to the states of subsequent layers; see Fig. 2.4.5. One possibility is to combine pruning with iterative deepening by eliminating from the computation states $\hat{x}_n$ of layer n such that the n-step lookahead cost $D_n(\hat{x}_n) + \tilde{J}(\hat{x}_n)$ is "far from the minimum" over x_n. This in turn prunes automatically some of the states of the next layer $n+1$. The rationale is that such states are "unlikely" to be part of the shortest path that we aim to compute. Note that this type of pruning is progressive, i.e., we prune states in layer n before pruning states in layer $n+1$.

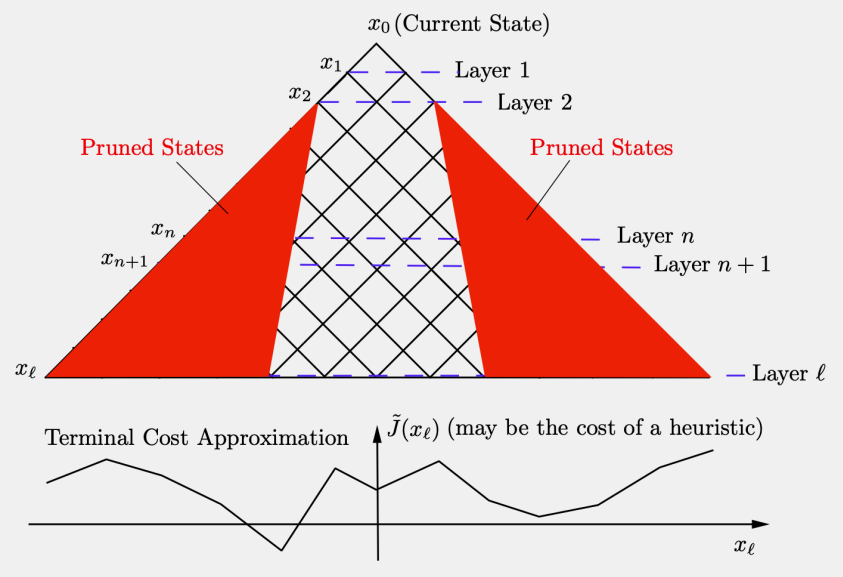

Figure 2.4.5 Illustration of iterative deepening with pruning within the context of forward DP.

2.4.2 Incremental Multistep Rollout

We will now consider a more flexible form of the rollout scheme, which we call *incremental multistep rollout* (IMR). It applies a base heuristic and a forward DP computation to a sequence of subgraphs of a multistep lookahead graph, with the size of the subgraphs expanding iteratively. In particular, in incremental rollout a connected subgraph of multiple paths is iteratively extended starting from the current state going towards the end of the lookahead horizon, instead of extending a single path as in rollout. This is similar to what is done in Monte Carlo Tree Search (MCTS, to be discussed later), which is also designed to solve approximately general multistep lookahead minimization problems (including stochastic ones), and involves iterative expansion of an acyclic lookahead graph to new nodes, as well as backtracking to previously encountered nodes. However, incremental rollout seems to be more appropriate than MCTS for deterministic problems, where there are no random variables in the problem's model and therefore Monte Carlo simulation does not make sense.

The IMR algorithm starts with and maintains a connected acyclic subgraph S of the given multistep lookahead graph G, which contains x_0. At each iteration it expands S by selecting a leaf node of S and by adding its neighbor nodes to S (if not already in S); see Fig. 2.4.6. The leaf node,

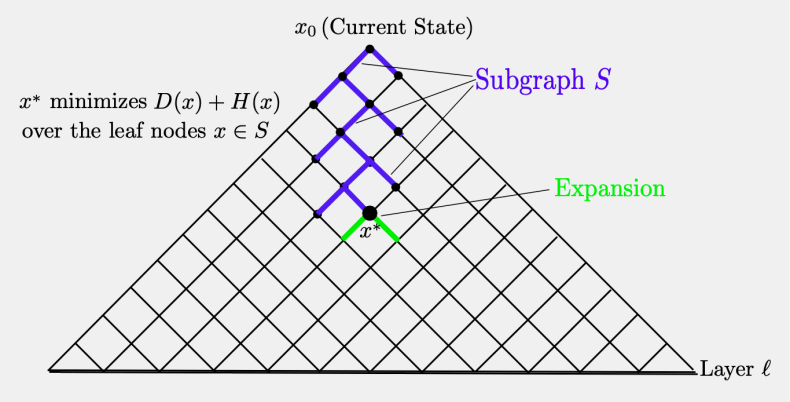

Figure 2.4.6 Illustration of the ℓ-step lookahead minimization problem and its suboptimal solution with the IMR algorithm. The algorithm maintains a connected acyclic subgraph S as shown. At each iteration it expands S by selecting a leaf node of S and by adding its neighbor nodes to S (if not already in S). The leaf node, denoted x^*, is the one that minimizes over all leaf nodes x of S the sum of the shortest distance $D(x)$ from x_0 to x and a "heuristic cost" $H(x)$.

denoted x^*, is the one that minimizes (over all leaf nodes x of S) the sum

$$D(x) + H(x),$$

where

$D(x)$ is the shortest distance from x_0 to the leaf node x using only arcs that belong to S. This can be computed by forward DP. [A noteworthy possibility is to replace $D(x)$ with a conveniently computed approximation, which may be problem-specific. We will discuss such schemes later in this section.]

$H(x)$ is a "heuristic cost" corresponding to x. This is defined as the sum of three terms:

(a) The cost of the base heuristic starting from node x and ending at one of the states x_ℓ in the last layer ℓ.

(b) The terminal cost approximation $\tilde{J}(x_\ell)$, where x_ℓ is the state obtained via the base heuristic as in (a) above.

(c) An additional penalty $P(x)$ that depends on the layer to which x belongs. As an example, we will assume here that

$$P(x) = \delta \cdot (\text{the layer index of } x),$$

where δ is a positive parameter. Thus $P(x)$ adds a cost of δ for each extra arc to reach x from x_0, and penalizes nodes x that

lie in more distant layers from the root x_0. It encourages the algorithm to "backtrack" and select nodes x^* that lie in layers closer to x_0. (Other ways to define P may also be considered.)

The algorithm starts with a connected acyclic subgraph S of the given multistep lookahead graph G, which contains x_0, and with a path P that starts at x_0, goes through S and ends at one of the terminal nodes of S (one possibility is take S and P equal to just the root node x_0). It terminates when the end node of P belongs to layer ℓ (or earlier if a computational budget constraint is reached).

The role of the parameter δ is noteworthy and affects significantly the nature of the algorithm. When $\delta = 0$, the initial graph S consists of the single state x_0, and the base heuristic is sequentially improving, it can be seen that IMR performs exactly like the rollout algorithm for solving the ℓ-step lookahead minimization problem. On the other hand when δ is large enough, the algorithm operates like the forward DP algorithm. The reason is that a very large value of δ forces the algorithm to expand all nodes of a given layer before proceeding to the next layer.

Generally, as δ increases, the algorithm tends to backtrack more often, and to generate more paths through the graph, thereby visiting more nodes and increasing the number of applications of the base heuristic. Thus δ may be viewed as an *exploration parameter*; when δ is large the algorithm tends to explore more paths thereby improving the quality of the multistep lookahead minimization, at the expense of greater computational effort. In the absence of additional problem-specific information, favorable values of δ should be obtained through experimentation. One may also consider alternative and more adaptive schemes; for example with a δ that depends on x_0, and is adjusted in the course of the computation. Finally, we note that if the base heuristic used in the calculation of $H(x)$ is sequentially consistent, a local optimality property of the incremental rollout trajectory can be shown (cf. Section 2.3.1).

Approximations in the Incremental Multistep Rollout Scheme

Let us now consider variants of the IMR scheme, which are aimed towards expediting its calculations, possibly at the expense of some degradation in its performance guarantees. To this end, it is useful to view the algorithm as maintaining a list L of the current leaf nodes of S. At each step a node is removed from L, and its neighbor nodes are added to S and to L, if not already there. This process continues until a path through S that starts at the root node x_0 and ends at some node of the last layer ℓ is constructed. In particular, each step of the IMR algorithm consists of:

(a) The selection of a node x^* of L for expansion.

(b) The addition to S and to L of each neighbor node x of the expanded node x^*, if x does not already belong to S and/or L, respectively.

The values $D(x)$ and $H(x)$ are also computed at the time when x enters the list L.

(c) The removal of x^* from L.

Since evidently a node can enter the list L at most once, the algorithm will terminate with a path that starts at x_0 and ends at some node of layer ℓ. Moreover, this will happen regardless of which leaf node of the current subgraph S is chosen at each step for expansion.

It follows from the preceding argument that selecting the leaf node x^* that minimizes $D(x)+H(x)$ at each step is not essential to the algorithm's termination. This allows some flexibility in designing variations of the IMR algorithm, which aim at expediting the computation. In particular, we may consider selecting a leaf node x that has a "low" value of $D(x)+H(x)$ instead of selecting the one that has minimum value, while maintaining an appropriate cost improvement property. Two possibilities of this type are as follows:

(1) Replace $D(x)$, the shortest distance from x_0 to x through the subgraph S, with an upper bound $\overline{D}(x)$, which is the shortest distance from x_0 to x through the subgraph S *at the time that x becomes a leaf node of S and enters the list L* of leaf nodes. Note that as the set S grows, $D(x)$ may become smaller than $\overline{D}(x)$, as more nodes are added to S and additional paths from x_0 to x through S are created. The important point here is that in general $\overline{D}(x)$ is in many cases likely to be either equal or not too different than $D(x)$. Moreover, $\overline{D}(x)$ is computed only once, at the time when x becomes a leaf node of S, thus saving in computational overhead.

(2) Organize L as a priority queue and instead of selecting a node x^* that minimizes $D(x)+H(x)$ or $\overline{D}(x)+H(x)$ over all nodes x of L, simply let x^* be the top node of the queue. This will save the overhead for minimizing over the nodes of L, which can be very large in number, depending on the problem at hand. We note here that it is possible to reduce this overhead by organizing L with a heap or bucket data structure, whereby the minimizing node is efficiently selected; such schemes are well known from implementations of label setting shortest path computations (i.e., Dijkstra's algorithm, see e.g., [Ber98], Ch. 2). However, simpler schemes to organize L are possible, which place nodes with small value of $D(x)+H(x)$ near the top of the queue. Such schemes have been proposed by the author for label correcting methods for shortest paths, in the context of approximations to Dijkstra's algorithm. We refer to the SLF (Small Label First), LLL (Last Label Last), and threshold shortest path algorithms, which are described in the network optimization book [Ber98] (Chapter 2) and the references given in that book. In practice, these algorithms work very well, and typically better than the heap or bucket schemes.

An important point is that when δ is small or is 0, the preceding variants of the IMR algorithm can be easily modified to coincide with the rollout algorithm, and inherit the corresponding cost improvement property. Moreover the cost improvement property can be restored by using the fortified rollout ideas of Section 2.3. In addition, the flexibility afforded by the modifications (1) and (2) above allow variations of the IMR scheme that are tailored to the problem at hand.

2.5 CONSTRAINED FORMS OF ROLLOUT ALGORITHMS

In this section we will discuss constrained deterministic DP problems, including challenging combinatorial optimization and integer programming problems. We introduce a rollout algorithm, which relies on a base heuristic and applies to problems with general trajectory constraints. Under suitable assumptions, we will show that if the base heuristic produces a feasible solution, the rollout algorithm has a cost improvement property: it produces a feasible solution, whose cost is no worse than the base heuristic's cost.

Before going into formal descriptions of the constrained DP problem formulation and the corresponding algorithms, it is worth to revisit the broad outline of the rollout algorithm for deterministic DP:

(a) It constructs a sequence $\{T_0, T_1, \ldots, T_N\}$ of complete system trajectories with monotonically nonincreasing cost (assuming a sequential improvement condition).

(b) The initial trajectory T_0 is the one generated by the base heuristic starting from x_0, and the final trajectory T_N is the one generated by the rollout algorithm.

(c) For each k, the trajectories $T_k, T_{k+1}, \ldots, T_N$ share the same initial portion $(x_0, \tilde{u}_0, \ldots, \tilde{u}_{k-1}, \tilde{x}_k)$.

(d) For each k, the base heuristic is used to generate a number of candidate trajectories, all of which share the initial portion with T_k, up to state $\tilde{x}_k$. These candidate trajectories correspond to the controls $u_k \in U_k(x_k)$. (In the case of fortified rollout, these trajectories include the current "tentative best" trajectory.)

(e) For each k, the next trajectory T_{k+1} is the candidate trajectory that is best in terms of total cost.

In our constrained DP formulation, to be described shortly, we introduce a trajectory constraint $T \in C$, where C is some subset of admissible trajectories. A consequence of this is that some of the candidate trajectories in (d) above, may be infeasible. Our modification to deal with this situation is simple: *we discard all the candidate trajectories that violate the constraint, and we choose T_{k+1} to be the best of the remaining candidate trajectories, the ones that are feasible.*

Of course, for this modification to be viable, we have to guarantee that at least one of the candidate trajectories will satisfy the constraint for every k. For this we will rely on a sequential improvement condition that we will introduce shortly. For the case where this condition does not hold, we will introduce a fortified version of the algorithm, which requires only that the base heuristic generates a feasible trajectory T_0 starting from the initial condition x_0. Thus *to apply reliably the constrained rollout algorithm, we only need to know a single feasible solution*, i.e., a trajectory T_0 that starts at x_0 and satisfies the constraint $T_0 \in C$.

Constrained Problem Formulation

We assume that the state x_k takes values in some (possibly infinite) set and the control u_k takes values in some finite set. The finiteness of the control space is only needed for implementation purposes of the rollout algorithms to be described shortly. The algorithm can be defined without the finiteness condition, and makes sense, provided the implementation issues associated with infinite control spaces can be dealt with. A sequence of the form

$$T = (x_0, u_0, x_1, u_1, \ldots, u_{N-1}, x_N), \tag{2.34}$$

where

$$x_{k+1} = f_k(x_k, u_k), \qquad k = 0, 1, \ldots, N-1, \tag{2.35}$$

is referred to as a *complete trajectory*. Our problem is stated succinctly as

$$\min_{T \in C} G(T), \tag{2.36}$$

where G is some cost function and C is the constraint set.

Note that G need not have the additive form

$$G(T) = g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k), \tag{2.37}$$

which we have assumed so far. Thus, except for the finiteness of the control space, which is needed for implementation of rollout, this is a very general optimization problem. In fact, later we will simplify the problem further by eliminating the state transition structure of Eq. (2.35).†

† Actually, similar to our discussion on model-free rollout in Section 2.3.6, it is not essential that we know the explicit form of the cost function G and the constraint set C. For our constrained rollout algorithms, it is sufficient to have access to a human or software expert that can determine whether a given trajectory T is feasible, i.e., satisfies the constraint $T \in C$, and also to be able to compare any two feasible trajectories T_1 and T_2, based on some internal process that is unknown to us, without assigning numerical values to them.

Trajectory constraints can arise in a number of ways. A relatively simple example is the standard problem formulation for deterministic DP: an additive cost of the form (2.37), where the controls satisfy the time-uncoupled constraints $u_k \in U_k(x_k)$ [so here C is the set of trajectories that are generated by the system equation with controls satisfying $u_k \in U_k(x_k)$]. In a more complicated constrained DP problem, there may be constraints that couple the controls of different stages such as

$$g_N^m(x_N) + \sum_{k=0}^{N-1} g_k^m(x_k, u_k) \leq b^m, \qquad m = 1, \ldots, M, \tag{2.38}$$

where g_k^m and b^m are given functions and scalars, respectively. Examples of this type include *multiobjective* or *Pareto* optimization problems, where there are multiple cost functions of interest, and all but one of the cost functions are treated through constraints (see e.g., [Ber17a], Ch. 2). Examples where difficult trajectory constraints arise also include situations where the control contains some discrete components, which once chosen must remain fixed for multiple time periods.

Here is a discrete optimization example involving the traveling salesman problem. A related classical example is the knapsack problem, described in most books on discrete optimization algorithms.

Example 2.5.1 (A Constrained Form of the Traveling Salesman Problem)

Let us consider a constrained version of the traveling salesman problem of Example 1.2.2. We want to find a minimum travel cost tour that additionally satisfies a safety constraint that the "safety cost" of the tour should be less than a certain threshold; see Fig. 2.5.1. This constraint need not have the additive structure of Eq. (2.38). We are simply given a safety cost for each tour (see the table at the bottom right), which is calculated in a way that is of no further concern to us. In this example, for a tour to be admissible, its safety cost must be less than or equal to 10. Note that the (unconstrained) minimum cost tour, ABDCA, does not satisfy the safety constraint.

Using a Base Heuristic for Constrained Rollout

We will now describe formally the constrained rollout algorithm. We assume the availability of a base heuristic, which for any given partial trajectory

$$y_k = (x_0, u_0, x_1, \ldots, u_{k-1}, x_k),$$

can produce a (complementary) partial trajectory

$$R(y_k) = (x_k, u_k, x_{k+1}, u_{k+1}, \ldots, u_{N-1}, x_N),$$

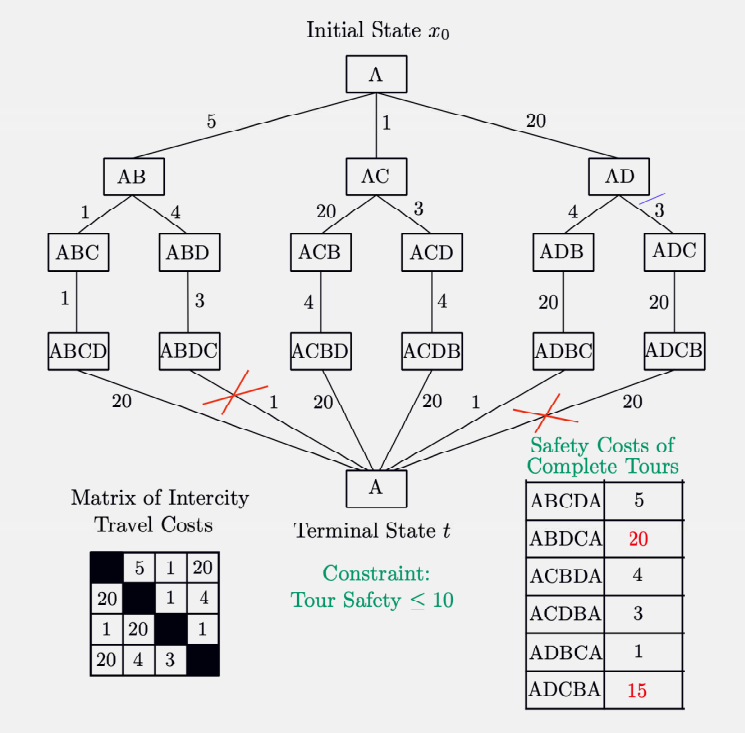

Figure 2.5.1 An example of a constrained traveling salesman problem; cf. Example 2.5.1. We want to find a minimum cost tour that has safety cost less or equal to 10. The safety costs of the six possible tours are given in the table on the right. The (unconstrained) minimum cost tour, ABDCA, does not satisfy the safety constraint. The optimal constrained tour is ABCDA.

that starts at x_k and satisfies the system equation

$$x_{t+1} = f_t(x_t, u_t), \qquad t = k, \ldots, N-1.$$

Thus, given y_k and any control u_k, we can use the base heuristic to obtain a complete trajectory as follows:

(a) Generate the next state $x_{k+1} = f_k(x_k, u_k)$.

(b) Extend y_k to obtain the partial trajectory

$$y_{k+1} = \big(y_k, u_k, f_k(x_k, u_k)\big).$$

(c) Run the base heuristic from y_{k+1} to obtain the partial trajectory $R(y_{k+1})$.

(d) Join the two partial trajectories y_{k+1} and $R(y_{k+1})$ to obtain the complete trajectory $\big(y_k, u_k, R(y_{k+1})\big)$, which is denoted by $T_k(y_k, u_k)$:

$$T_k(y_k, u_k) = \big(y_k, u_k, R(y_{k+1})\big). \tag{2.39}$$