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

Publisher's Cataloging-in-Publication Data

Bertsekas, Dimitri P.
Class Notes for a Course on 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-??-?, ISBN-13: 978-1-886529-??-?

CONTENTS

2. Approximation in Value Space - Rollout Algorithms

Preface

These lecture notes were prepared for use in the 2023 ASU research-oriented course on Reinforcement Learning (RL), the fifth offering of this course Their purpose is to give an overview of the RL methodology, particularly as it relates to problems in optimal and suboptimal decision and control, as well as discrete optimization.

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 the subject. This framework centers on approximate forms of Dynamic Programming (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 for solution of 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 lookahead 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 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 can be understood in terms of abstract models of infinite horizon DP and simple geometrical constructions, 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 these notes we will aim to explain (often with visualization) the beneficial effects of on-line decision making on top of off-line training. 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.

We will also aim to show, through the algorithmic ideas of Newton's

method and the unifying principles of abstract DP, that the AlphaZero/TD-Gammon 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. Moreover, we will 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.

We finally note that while we will deemphasize mathematical proofs in these notes, there is considerable related analysis, which supports our conclusions and can be found in the author's recent RL and DP books. These books also contain additional material on off-line training of neural networks, on the use of policy gradient methods for approximation in policy space, and on aggregation.

Sources

While these notes are focused primarily on on-line play, the algorithmic aspects of off-line training are covered at some length (see Chapter 3). They are also discussed in greater detail in 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.

The two books above also include a far more detailed discussion of MPC, adaptive control, and discrete optimization topics, than the present class notes. Moreover, other popular methods, such as temporal difference methods, Q-learning, policy gradient methods for approximation in policy space, and a variety of approximation methodologies, are discussed in the books [1] and [2], but not in these class notes.

The author's two-volume DP book (4th edition)

[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 course:

[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 class notes. 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 class notes 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.

All of the above books are available as ebooks as well as in print form; see the Athena Scientific website. Most of the material in these class notes is essentially replicated and adapted from these books. However, the books themselves collectively provide a presentation that is far more detailed and mathematically rigorous than these notes.

The present class notes can 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 Course Notes - Course Adaptations

An important structural characteristic of these notes is that they are organized in a modular way, with a view towards flexibility, so they can be easily modified to accommodate changes in course content. In particular, the notes are 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, other choices for in-depth coverage may be made, using the same foundational platform. In particular, both more and less mathematically-oriented courses can be built upon the platform of Chapter 1.

Let us also mention that the notes contain 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 notes 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 these notes.

Thanks and Appreciation

The hospitable and stimulating environment at ASU contributed much to shaping my course during the period 2019-2023. For this I am also very thankful to my ASU colleagues and students, including 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 also provided valuable proofreading support.

Dimitri P. Bertsekas
May 2023

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 present class notes. 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 towards 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 notes reflect the author's decision/control and operations research orientation, which has in turn guided the choices of terminology, notation, and mathematical style for these notes. 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 notes, 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 these notes, 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.

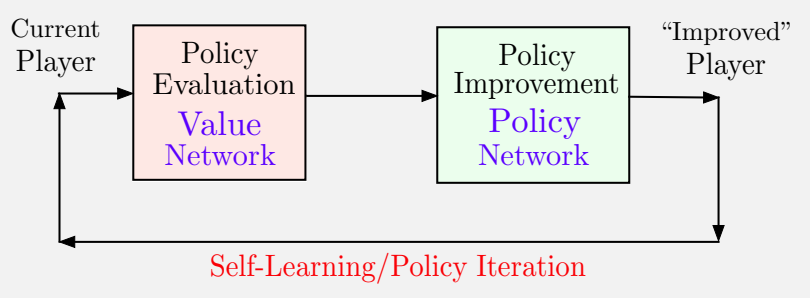


Figure 1.1.1 Illustration of the AlphaZero training algorithm. It generates a sequence of position evaluators and chess players. The position evaluator and the chess player are represented by two neural networks, a value network and a policy network, which accept a chess position and generate a position 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 these notes. 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 thus collected, and are 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 is not guaranteed theoretically to play well against a particular human or computer player.

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.

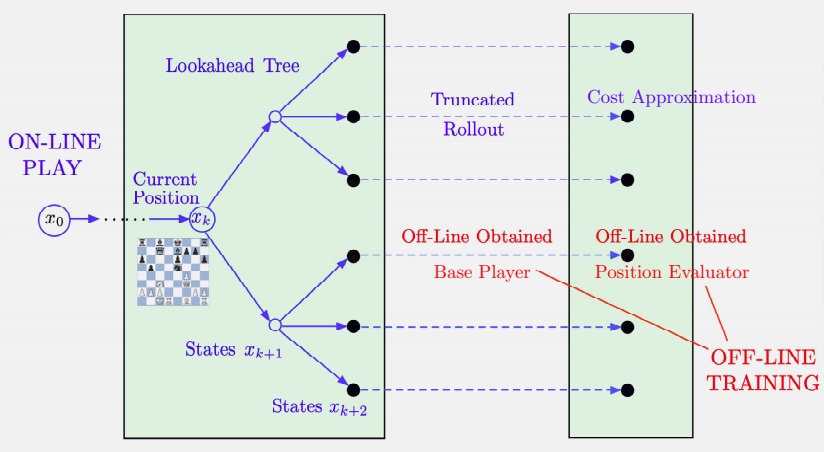


Figure 1.1.2 Illustration of an on-line player such as the one used in AlphaGo, AlphaZero, and Tesauro's backgammon program [TeG96]. At a given position, it generates a lookahead tree of multiple moves up to some depth, then runs the off-line obtained player for some more moves, and evaluates the effect 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 notes progress. However, DP ideas with cost function approximations, similar to the on-line player illustrated in Fig. 1.1.2, will be central for our purposes. Moreover, the algorithmic division between off-line training and on-line policy implementation will be conceptually very important for our purposes.

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.

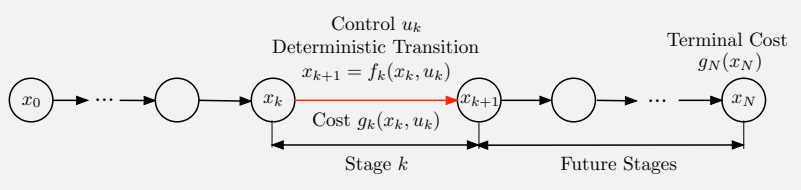


Figure 1.2.1 Illustration of a deterministic N-stage optimal control problem. Starting from state x_k, the next state under control u_k is generated nonrandomly, according to

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

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 these notes.†

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 very briefly here, but a fuller account that is consistent in terminology with the present notes 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 these notes, 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 these notes. 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).

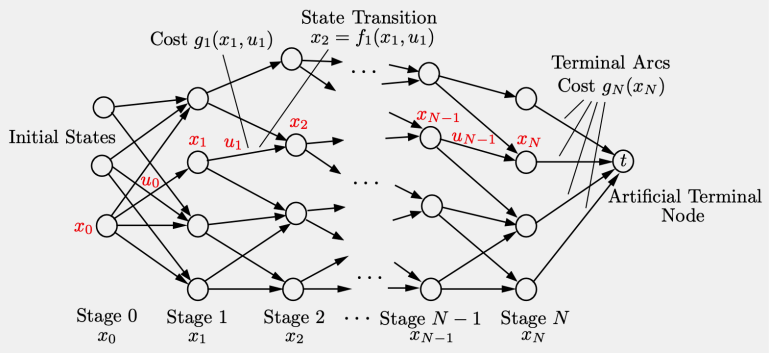


Figure 1.2.2 Transition graph for a deterministic finite-state system. Nodes correspond to states x_k. Arcs correspond to state-control pairs (x_k, u_k). An arc (x_k, u_k) has start and end nodes x_k and $x_{k+1} = f_k(x_k, u_k)$, respectively. We view the cost $g_k(x_k, u_k)$ of the transition as the 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.

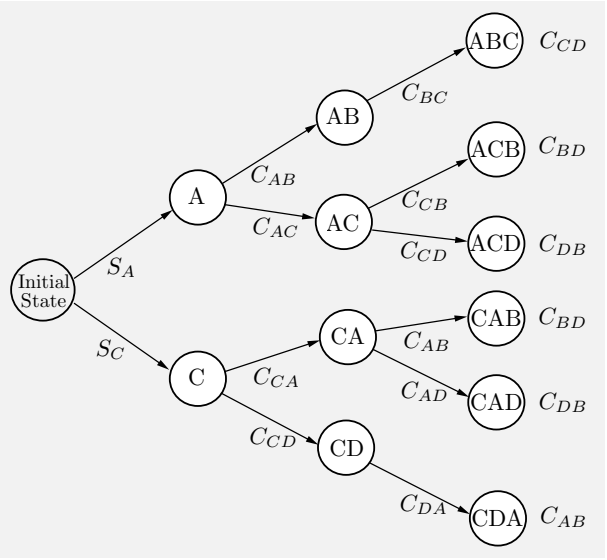


Figure 1.2.3 The transition graph of the deterministic scheduling problem of Example 1.2.1. Each arc of the graph corresponds to a decision leading from some state (the start node of the arc) to some other state (the end node of the arc). The corresponding cost is shown next to the arc. The cost of the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

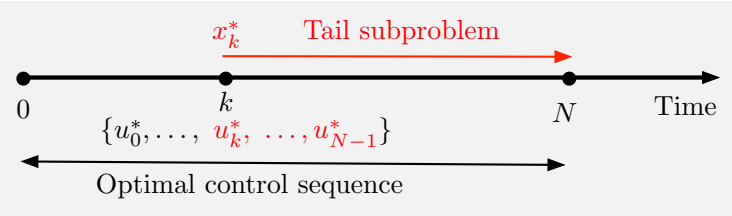


Figure 1.2.4 Schematic illustration of the principle of optimality. The tail $\{u_k^*, \ldots, u_{N-1}^*\}$ of an optimal sequence $\{u_0^*, \ldots, u_{N-1}^*\}$ is optimal for the tail subproblem that starts at the state x_k^* of the optimal state trajectory.

(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."

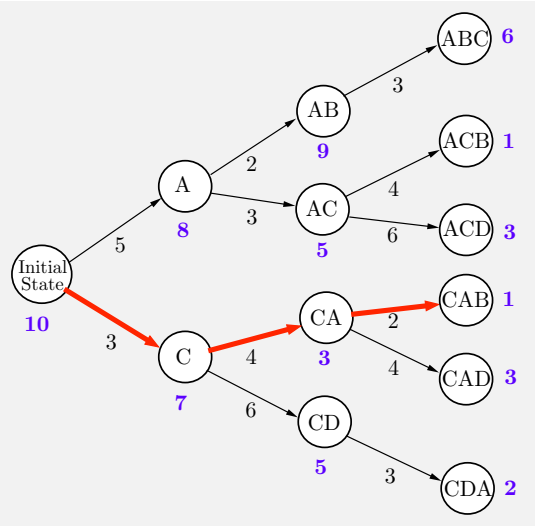


Figure 1.2.5 Transition graph of the deterministic scheduling problem, with the cost of each decision shown next to the corresponding arc. Next to each node/state we show the cost to optimally complete the schedule starting from that state. This is the optimal cost of the corresponding tail subproblem (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.

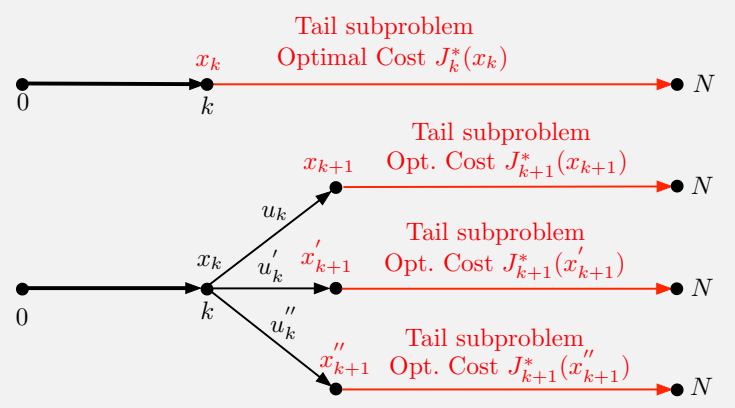


Figure 1.2.6 Illustration of the DP algorithm. The tail subproblem that starts at x_k at time k minimizes over $\{u_k, \ldots, u_{N-1}\}$ the "cost-to-go" from k to N,

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

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

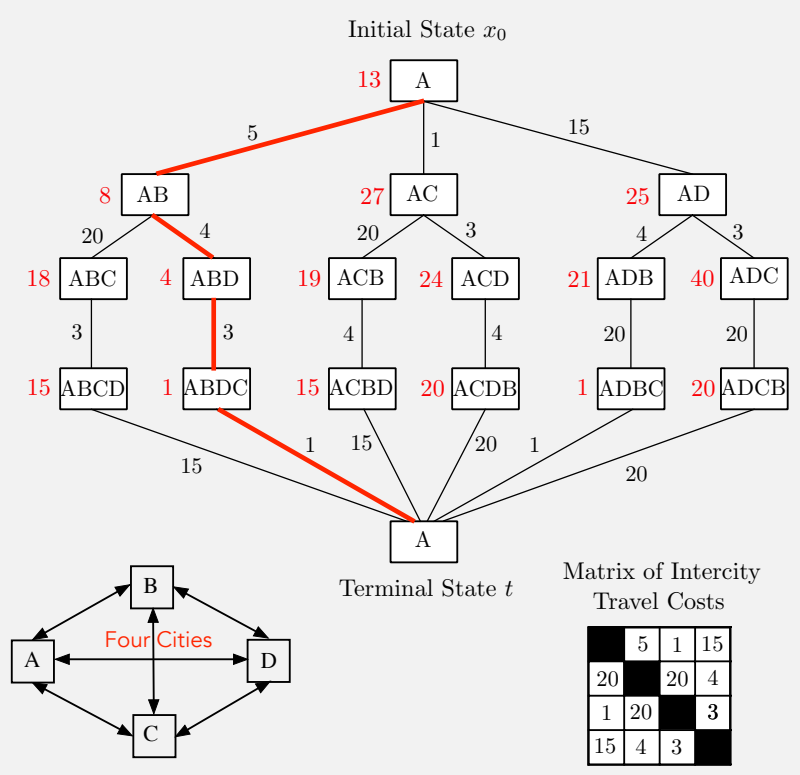


Figure 1.2.7 The DP formulation of the traveling salesman problem of Example 1.2.2. The travel times between the four cities A, B, C, and D are shown in the matrix at the bottom. We form a graph whose nodes are the k-city sequences and correspond to the states of the kth stage, assuming that A is the 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.

traveling salesman and related scheduling problems are typically addressed with approximation methods, some of which are based on DP, and will be discussed in future chapters.

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 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 underlies the widely used A^* method for computing approximate solutions to large scale shortest path problems.

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 these notes. 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.

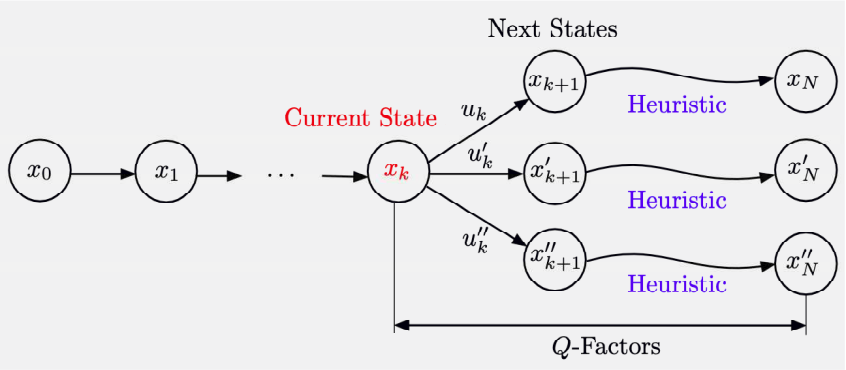


Figure 1.2.9 Schematic illustration of rollout with one-step lookahead for a deterministic problem. At state x_k, for every pair (x_k, u_k), $u_k \in U_k(x_k)$, the base heuristic generates an approximate Q-factor

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

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

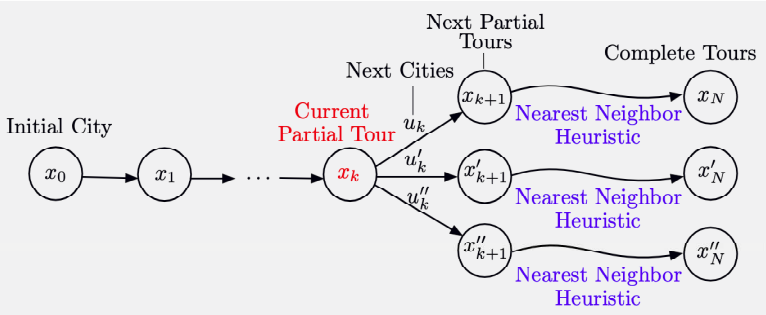


Figure 1.2.10 Rollout with the nearest neighbor heuristic for the traveling salesman problem of Example 1.2.3. The initial state x_0 consists of a single city. The final state x_N is a complete tour of N cities, containing each city exactly once.

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

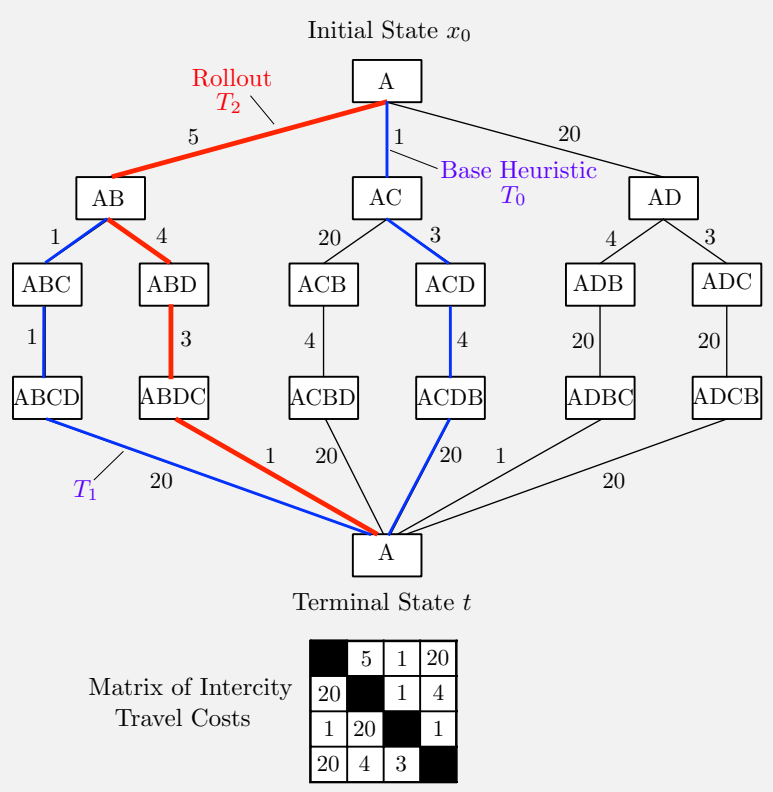


Figure 1.2.11 Rollout with the nearest neighbor base heuristic, applied to a traveling salesman problem. At city A, the nearest neighbor heuristic generates the tour ACDBA (labelled T_0). At city A, the rollout algorithm compares the tours ABCDA, ACDBA, and ADCBA, finds ABCDA (labelled T_1) to have the 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 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

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

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.,

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

† We assume an introductory probability background on the part of the reader. For an account that is consistent with our use of probability in these notes; see the text by Bertsekas and Tsitsiklis [BeT08].

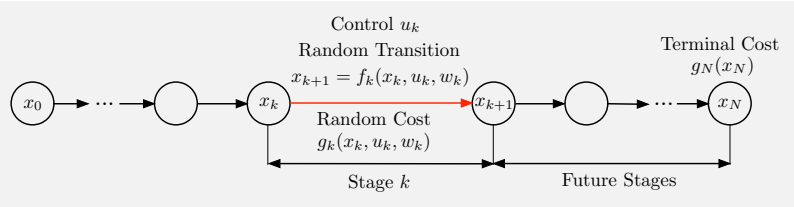


Figure 1.3.1 Illustration of an N-stage stochastic optimal control problem. Starting from state x_k, the next state under control u_k is generated randomly, according to $x_{k+1} = f_k(x_k, u_k, w_k)$, where w_k is the random disturbance, and a random stage cost $g_k(x_k, u_k, w_k)$ is incurred.

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.

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

$$\begin{aligned}
J_{N-1}^*(x_{N-1}) &= \min_{u_{N-1}} E\Big\{qx_{N-1}^2 + ru_{N-1}^2 + J_N^*(ax_{N-1} + bu_{N-1} + w_{N-1})\Big\} \\
&= \min_{u_{N-1}} E\Big\{qx_{N-1}^2 + ru_{N-1}^2 + q(ax_{N-1} + bu_{N-1} + w_{N-1})^2\Big\} \\
&= \min_{u_{N-1}} \Big[qx_{N-1}^2 + ru_{N-1}^2 + q(ax_{N-1} + bu_{N-1})^2 \\
&\qquad + 2qE\{w_{N-1}\}(ax_{N-1} + bu_{N-1}) + qE\{w_{N-1}^2\}\Big],
\end{aligned}$$

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

$$J_{N-1}^*(x_{N-1}) = qx_{N-1}^2 + q\sigma^2 + \min_{u_{N-1}} \Big[ru_{N-1}^2 + q(ax_{N-1} + bu_{N-1})^2\Big]. \tag{1.17}$$

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

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

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

$$\mu_k^*(x_k) = L_k x_k, \qquad k = 0, 1, \ldots, N-1,$$

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

where

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

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

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

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

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

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.

Truncated Rollout

Our discussion of rollout of Section 1.2 also applies to stochastic problems: we select $\tilde{J}_k$ to be the cost function of a suitable base policy (perhaps with some approximation). Note that 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 the cost improvement property, whereby it yields an improved cost relative to its underlying base policy.

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

† The principal role of neural networks within the context of these notes 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. Detailed discussions are found in many sources, including the RL books [Ber19a], [Ber20a], and the neuro-dynamic programming book [BeT96].

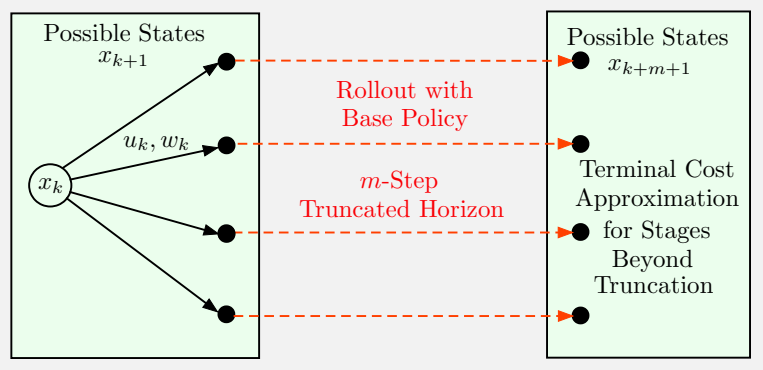


Figure 1.3.2 Schematic illustration of truncated rollout. One-step lookahead is followed by simulation of the base policy for m steps, and an approximate cost $\tilde{J}_{k+m+1}(x_{k+m+1})$ is added to the cost of the simulation, which depends on the state x_{k+m+1} obtained at the end of the rollout. 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 Chapter 2.

value space scheme (1.21); see Fig. 1.3.2. 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.

Further Approximations

When designing approximation in value space schemes, one may consider several interesting simplification ideas, which are aimed at alleviating the computational overhead. One possibility 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.

A different type of simplification relates to approximations in the computation of the expected value in Eq. (1.21) by using limited Monte

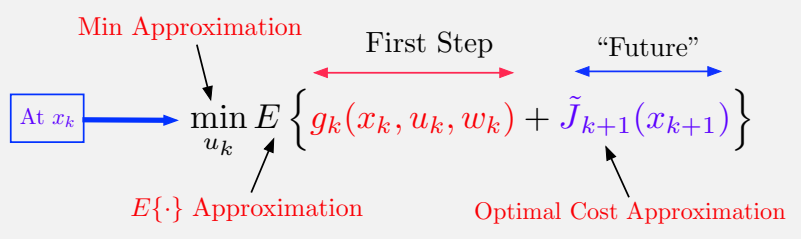


Figure 1.3.3 Schematic illustration of approximation in value space for stochastic problems, and the three approximations involved in its design. Typically the approximations can be designed independently of each other. There are also multistep lookahead versions of approximation in value space, which will be discussed later.

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.

Another 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 random variables $w_{k+1}, \ldots, w_{k+m}$ associated with truncated rollout 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 combination of truncated rollout with a *problem approximation approach*, 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.

Figure 1.3.3 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 with a variety of methods. Much of the discussion in these notes will revolve around different ways to organize these three approximations for both cases of one-step and multistep lookahead.

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 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.

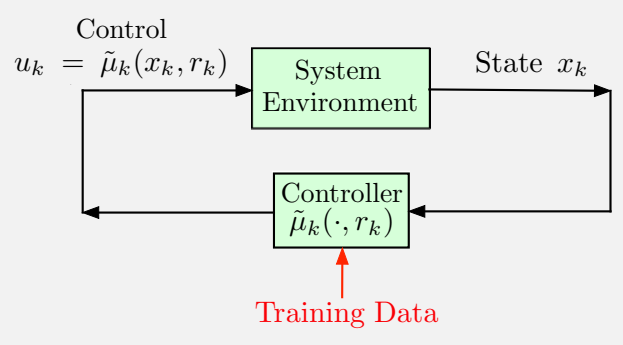


Figure 1.3.4 Schematic illustration of parametric approximation in policy space. A policy

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

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.

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

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

(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"

† 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 $\left(x_k^s, u_k^s\right)$, $s = 1, \ldots, q$, which are viewed as state-category pairs,

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

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

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

---

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 $\hat{r}$.

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 not be discussed in these notes (for details and many references to the literature, see 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).

1.4 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

$$J_\pi(x_0) = \lim_{N\to\infty} \mathop{E}_{\substack{w_k \\ k=0,1,\ldots}} \left\{ \sum_{k=0}^{N-1} \alpha^k g\big(x_k, \mu_k(x_k), w_k\big) \right\}; \tag{1.25}$$

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

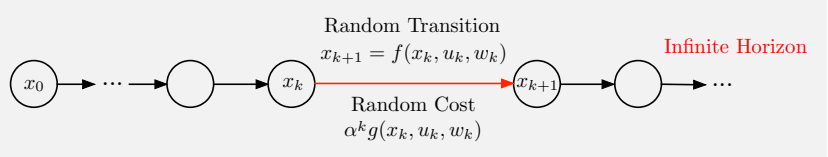


Figure 1.4.1 Illustration of an infinite horizon problem. The system and cost per stage are stationary, except for the use of a discount factor α. If $\alpha = 1$, there is typically a special cost-free termination state that we aim to reach.

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

$$x_{k+1} = f(x_k, u_k), \qquad k = 0, 1, \ldots, \tag{1.26}$$

and

$$J_\pi(x_0) = \lim_{N\to\infty} \sum_{k=0}^{N-1} \alpha^k g\big(x_k, \mu_k(x_k)\big). \tag{1.27}$$

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

$$g(x,u) \geq 0, \qquad \text{for all } x \neq t,\ u \in U(x), \tag{1.28}$$

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 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 these notes 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

$$J_{k+1}(x) = \min_{u \in U(x)} E_w \Big\{ g(x, u, w) + \alpha J_k\big(f(x, u, w)\big) \Big\}, \qquad k = 0, 1, \ldots, \tag{1.29}$$

starting from $J_0(x) \equiv 0$.† The algorithm (1.29) 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.,

$$J^*(x) = \lim_{N \to \infty} J_N(x) \tag{1.30}$$

† 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 (1.29).

for all states x.

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

$$J^*(x) = \min_{u \in U(x)} E_w\Big\{g(x,u,w) + \alpha J^*\big(f(x,u,w)\big)\Big\}. \tag{1.31}$$

This is obtained by taking the limit as $N \to \infty$ in the VI algorithm (1.29) using Eq. (1.30). 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 (1.31) 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

$$J_\mu(x) = E_w\Big\{g\big(x,\mu(x),w\big) + \alpha J_\mu\big(f(x,\mu(x),w)\big)\Big\}, \qquad \text{for all } x. \tag{1.32}$$

We can view this as just the Bellman equation (1.31) 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

$$J_{\mu,k+1}(x) = E_w\Big\{g\big(x,\mu(x),w\big) + \alpha J_{\mu,k}\big(f(x,\mu(x),w)\big)\Big\}, \tag{1.33}$$

starting from $J_{\mu,0}(x) \equiv 0$ or some other initial condition; cf. Eqs. (1.29)-(1.30).

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 these notes, 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 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. (1.32). 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

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

† 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].

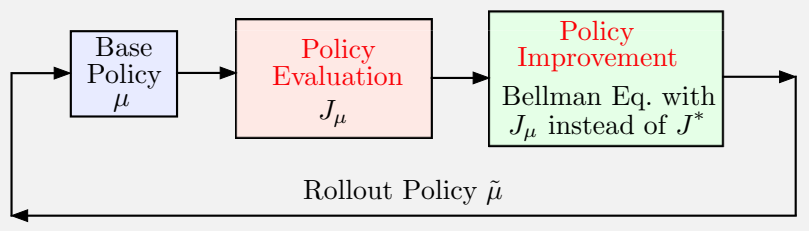


Figure 1.4.2 Schematic illustration of PI as repeated rollout. It generates a sequence of policies, with each policy μ in the sequence being the base policy that generates the next policy $\tilde{\mu}$ in the sequence as the corresponding rollout policy. This rollout policy is used as the base policy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

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*

$$\tilde{\mu}(x) \in \arg\min_{u \in U(x)} E\Big\{g(x,u,w) + \alpha \tilde{J}\big(f(x,u,w)\big)\Big\}. \tag{1.34}$$

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 (1.31) 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

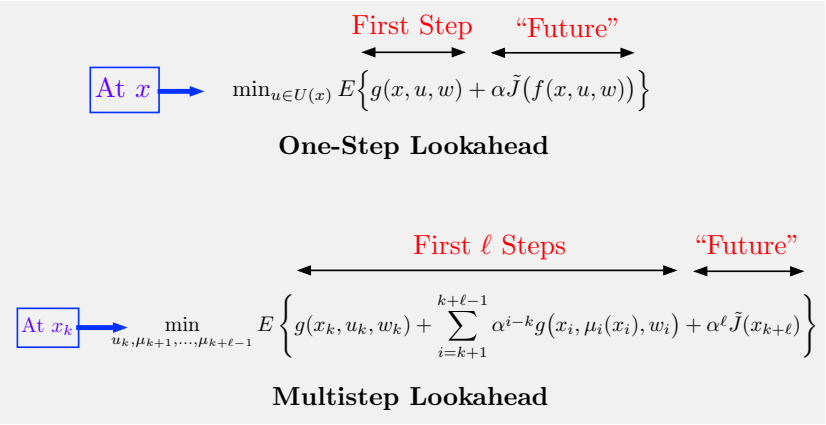


Figure 1.4.3 Schematic illustration of approximation in value space with one-step and ℓ-step lookahead minimization for infinite horizon problems. In the former case, the minimization yields at state x a control $\tilde{u}$, which defines the one-step lookahead policy $\tilde{\mu}$ via

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

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 $\tilde{\mu}$.

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 these notes, 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.,

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

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 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

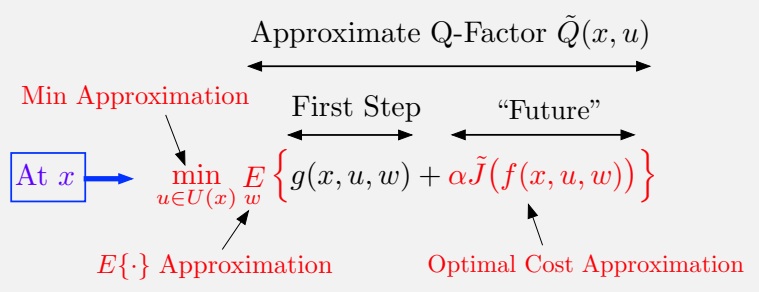


Figure 1.4.4 Approximation in value space with one-step lookahead for infinite horizon problems. There are three potential areas of approximation, which can be considered independently of each other: optimal cost approximation, expected value approximation, and minimization approximation.

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 hand, solving the multistep lookahead minimization problem, instead of the one-step lookahead counterpart of Eq. (1.34), 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 ex-

ample 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*, which is described and analyzed in the books [Ber12], [Ber19a], and the 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 a yield 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, lin-

ear 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 (1.34) 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. (1.34)]. 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

$$\tilde{\mu}(x) \in \arg\min_{u\in U(x)} E\Big\{g(x,u,w)+\alpha \min_{u'\in U(f(x,u,w))} \tilde{Q}\big(f(x,u,w),u'\big)\Big\}; \quad (1.35)$$

cf. Eq. (1.34). 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. (1.35) 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?* Here we are interested in whether $J_{\tilde{\mu}}(x)$ is smaller than $\tilde{J}(x)$ across a range of states x of interest, and by how much. *A fundamental fact 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. (1.31).* This will be explained through intuitive visulaization in the next section for the case of linear quadratic problems.

A related insight is that in approximation in value space with multistep lookahead schemes, *$J_{\tilde{\mu}}$ 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 sensitive is the quality of the lookahead policy $\tilde{\mu}$ to the quality of the off-line training?* Here we are interested to understand how much $J_{\tilde{\mu}}$ changes when $\tilde{J}$ changes across a range of interest. For example, how much should we care about improving $\tilde{J}$ through a longer and more sophisticated training process?

(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. As noted earlier, we will use an optimization-based definition of stability, calling the lookahead policy $\tilde{\mu}$ stable if $J_{\tilde{\mu}}(x) < \infty$, for all x. An example is an SSP problem with positive cost per stage, where some policies may not guarantee that the termination state will be reached; these policies are viewed as unstable. An example of a context where there are no stability concerns is discounted problems with bounded cost per stage; here all policies are stable according to our definition. While there are several alternative definitions of stability, which may be better-matched to specific contexts, our definition of stability is suitable for the very broad class of problems that we are dealing with.

(d) *What is the region of stability?* Here we are interested to characterize the set of terminal cost approximations $\tilde{J}$ that lead to a stable lookahead policy $\tilde{\mu}$.

(e) *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 INFINITE HORIZON 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 nonnegative cost problem. This is the classical continuous-spaces problem where the system is linear, with no control constraints, and the cost function is quadratic. While this problem 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 case where the system is

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

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], Chapter 3, and [Ber12], Chapter 4.

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

$$x_{k+1} = ax_k + bu_k; \tag{1.36}$$

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

$$\sum_{k=0}^{\infty} (qx_k^2 + ru_k^2), \tag{1.37}$$

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

$$J^*(x) = K^*x^2, \tag{1.38}$$

where the scalar K^* solves the equation

$$K = F(K), \tag{1.39}$$

with F defined by

$$F(K) = \frac{a^2rK}{r + b^2K} + q. \tag{1.40}$$

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

Moreover, the optimal policy is linear of the form

$$\mu^*(x) = L^*x, \tag{1.41}$$

where L^* is the scalar given by

$$L^* = -\frac{abK^*}{r + b^2K^*}. \tag{1.42}$$

For justification of Eqs. (1.39)-(1.42), we show that J^* as given by Eq. (1.38), satisfies the Bellman equation

$$J(x) = \min_{u\in\Re}\big\{qx^2 + ru^2 + J(ax + bu)\big\}, \tag{1.43}$$

and that $\mu^*(x)$, as given by Eqs. (1.41)-(1.42), 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 (1.43) is written as

$$\min_{u\in\Re}\big\{qx^2 + ru^2 + K(ax + bu)^2\big\}. \tag{1.44}$$

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. (1.44), we obtain

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

so the minimizing control and corresponding policy are given by

$$\mu_K(x) = L_Kx, \tag{1.45}$$

where

$$L_K = -\frac{abK}{r + b^2K}. \tag{1.46}$$

By substituting this control, the minimized expression (1.44) takes the form

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

After straightforward algebra, using Eq. (1.46) for L_K, it can be verified that this expression is written as $F(K)x^2$, with F given by Eq. (1.40). Thus when $J(x) = Kx^2$, the Bellman equation (1.43) takes the form

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

or equivalently $K = F(K)$ [cf. Eq. (1.39)].

In conclusion, when restricted to quadratic functions $J(x) = Kx^2$ with $K \geq 0$, the Bellman equation (1.43) is equivalent to the equation

$$K = F(K) = \frac{a^2rK}{r + b^2K} + q. \tag{1.47}$$

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. (1.45)-(1.46), attains the minimum in Bellman's equation, and is given by Eqs. (1.41)-(1.42).

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. (1.38)] is the unique solution of the Riccati equation $K = F(K)$ within the nonnegative real line.

---

† 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.

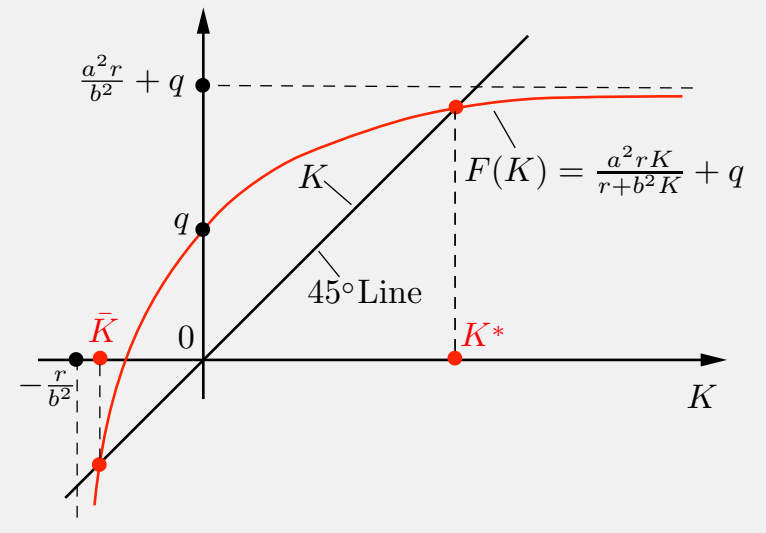


Figure 1.5.1 Graphical construction of the solutions of the Riccati equation (1.39)-(1.40) for the linear quadratic problem. The optimal cost function is $J^*(x) = K^*x^2$, where the scalar K^* solves the fixed point equation $K = F(K)$, with F being the function given by

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

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.

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

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

where K_L solves the equation

$$K = F_L(K), \tag{1.48}$$

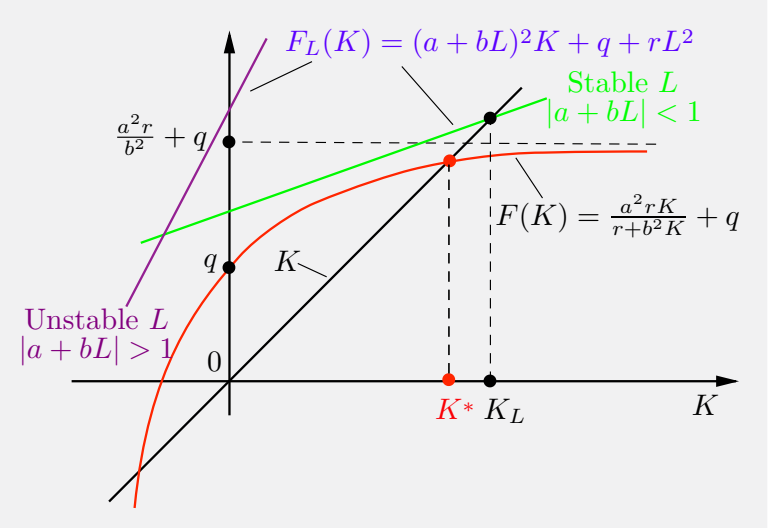


Figure 1.5.2 Illustration of the construction of the cost function of a linear policy $\mu(x) = Lx$, which is stable, i.e., $|a + bL| < 1$. The cost function $J_\mu(x)$ has the form

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

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.

with F_L defined by

$$F_L(K) = (a + bL)^2 K + q + rL^2. \tag{1.49}$$

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 (1.48)-(1.49) for $\mu(x) = Lx$ may be justified by verifying that it is in fact the Bellman equation for μ,

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

[cf. Eq. (1.32)], restricted to quadratic functions of the form $J(x) = Kx^2$.

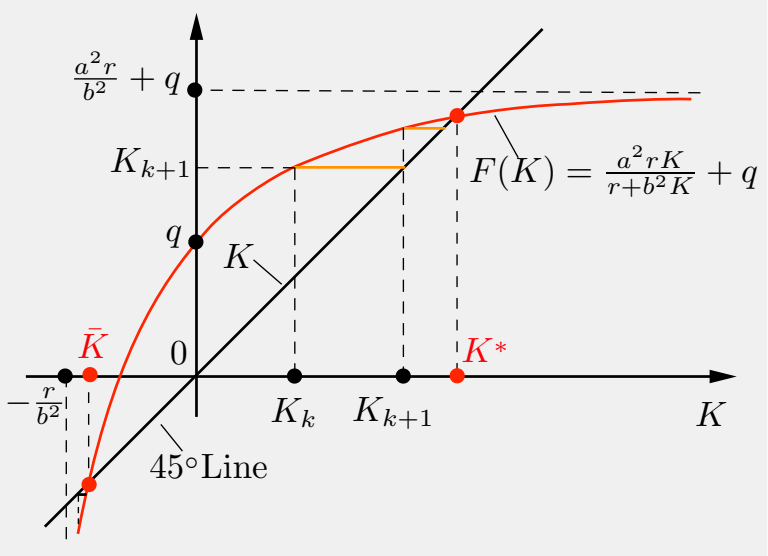


Figure 1.5.3 Graphical illustration of value iteration for the linear quadratic problem. It has the form $K_{k+1} = F(K_k)$, where F is the Riccati operator,

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

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

We note, however, that $J_\mu(x) = K_L x^2$ is the solution of the Riccati equation (1.48)-(1.49) 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 (1.48)-(1.49) 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

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

with F being the Riccati operator of Eq. (1.47). 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^*.

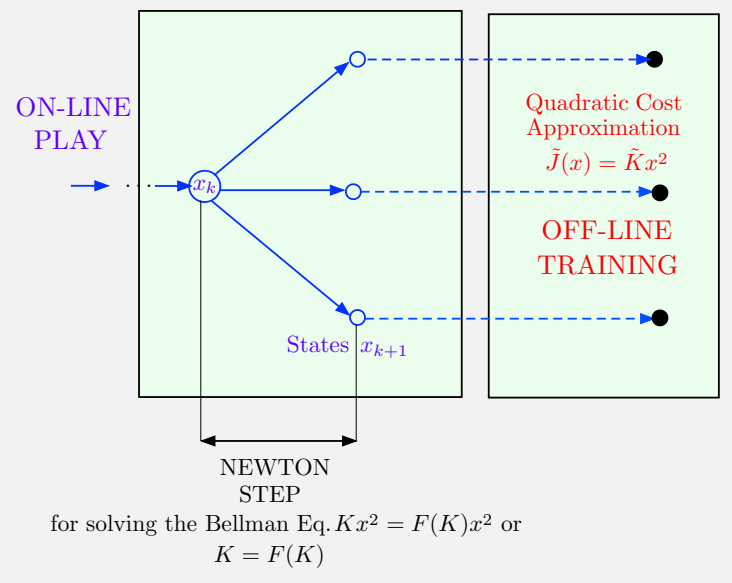


Figure 1.5.4 Illustration of the interpretation of approximation in value space with one-step lookahead as a Newton step.

1.5.1 Visualizing Approximation in Value Space - Newton's Method

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. 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.

(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

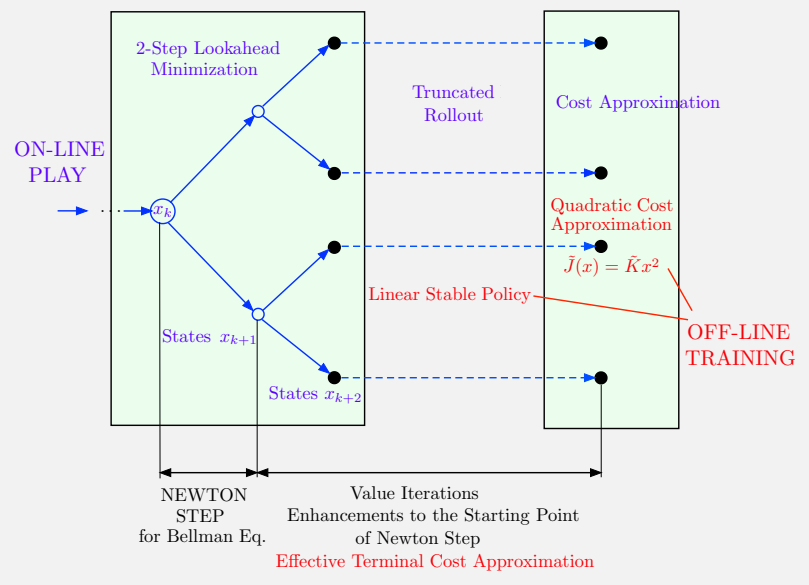


Figure 1.5.5 Illustration of the interpretation of approximation in value space with multistep lookahead and truncated rollout as a Newton step.

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 with Newton's method was first developed in great 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. (1.45)-(1.46), by minimizing the right side of Bellman's equation when $J(x) = Kx^2$:

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

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

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

From this equation, it follows that

$$F(K) = \min_{L \in \Re} F_L(K), \tag{1.50}$$

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

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

Figure 1.5.6 illustrates the relation (1.50)-(1.51), 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.

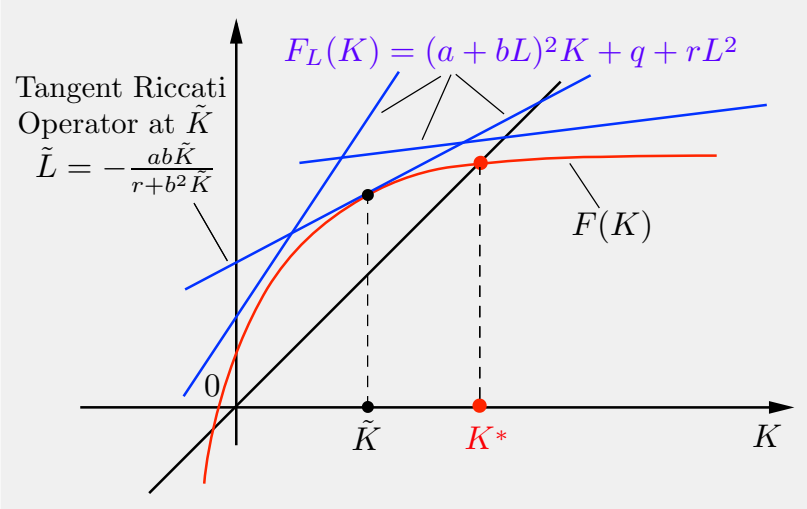


Figure 1.5.6 Illustration of how the graph of the Riccati operator F can be obtained as the lower envelope of the linear operators

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

as L ranges over the real numbers. We have

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

cf. Eq. (1.50). 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. (1.46)], 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.

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. (1.51)].

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

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

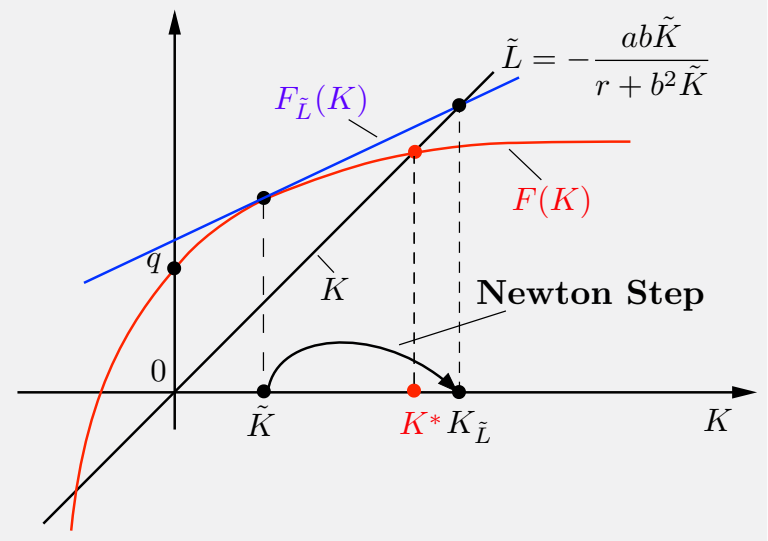


Figure 1.5.7 Illustration of approximation in value space with one-step lookahead for the linear quadratic problem. Given a terminal cost approximation $\tilde{J} = \tilde{K}x^2$, we compute the corresponding linear policy $\tilde{\mu}(x) = \tilde{L}x$, where

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

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

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 1.6.

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 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

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

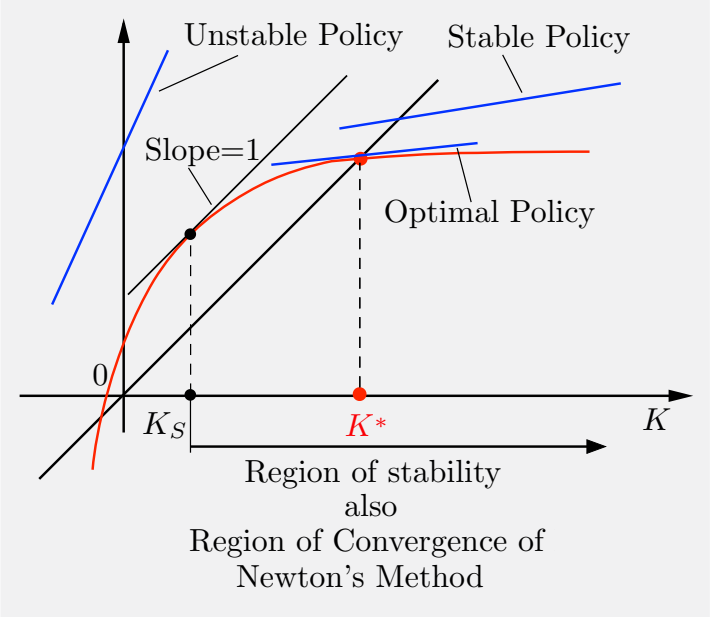


Figure 1.5.8 Illustration the region of stability, i.e., the set of $K \geq 0$ such that the one-step lookahead policy μ_K is stable. This is also the set of initial conditions for which Newton's method converges to K^* asymptotically.

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.

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 $x \neq 0$.

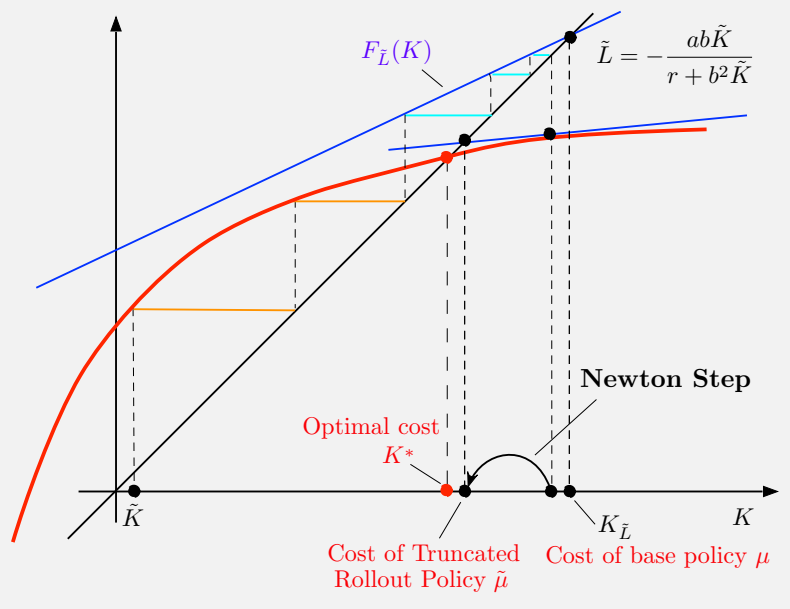


Figure 1.5.9 Illustration of approximation in value space with two-step lookahead for the linear quadratic problem. Starting with a terminal cost approximation $\tilde{J} = \tilde{K}x^2$, we obtain K_1 using a single value iteration. We then compute the corresponding linear policy $\tilde{\mu}(x) = \tilde{L}x$,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

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 $\ell = 2$.

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. (1.46). 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.

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*.

It is interesting to note that as the length of lookahead increases, 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.

We summarize the Riccati equation formulas and the relation between linear policies of the form $\mu(x) = Lx$ and their quadratic cost functions in the following table.

Riccati Equation Formulas for One-Dimensional Problems

Riccati equation for minimization [cf. Eqs. (1.39) and (1.40)]

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

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. (1.46)]

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

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.

1.5.2 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. In this section, we focus on a one-step lookahead scheme that produces a policy $\tilde{\mu}$ where $\tilde{J}$ is the terminal quadratic cost function approximation. We will try to estimate the difference $J_{\tilde{\mu}} - J^*$, where $J_{\tilde{\mu}}$ is the cost function of $\tilde{\mu}$ and J^* is the optimal cost function, assuming that $\tilde{J}$ lies within the region of stability, so that $\tilde{\mu}$ is well-defined as a stable policy and $J_{\tilde{\mu}}$ is finite-valued. The analysis easily extends to ℓ-step lookahead by viewing it as one-step lookahead with a terminal cost function $T^{\ell-1}\tilde{J}$, i.e., $\tilde{J}$ transformed by $\ell - 1$ value iterations.

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

$$\|J_{\tilde{\mu}} - J^*\| \leq \frac{2\alpha}{1-\alpha}\|\tilde{J} - J^*\|; \tag{1.52}$$

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. This bound also applies more generally, to the case 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, however, this error bound is very conservative, and does not reflect practical reality. The reason is that this is a *global* error bound, i.e., it holds 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 step that starts at $\tilde{J}$. In other words, for $\tilde{J}$ relatively close to J^*, we have the *local* estimate

$$\|J_{\tilde{\mu}} - J^*\| = o\big(\|\tilde{J} - J^*\|\big). \tag{1.53}$$

In practical terms, *there is often a huge difference, both quantitative and qualitative, between the error bounds (1.52) and (1.53).*

For an illustration, 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

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

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

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

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 (1.52) can be rederived for the case of quadratic functions and can be rewritten in terms of quadratic cost coefficients as

$$K_{\tilde{\mu}} - K^* \leq \frac{2\alpha}{1-\alpha}|\tilde{K} - K^*|, \tag{1.54}$$

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. 1.5.10. It can be seen that $(K_{\tilde{\mu}} - K^*)$ exhibits the qualitative behavior of Newton's method, which is very different than the bound (1.54). An interesting fact is that the bound (1.54) 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.

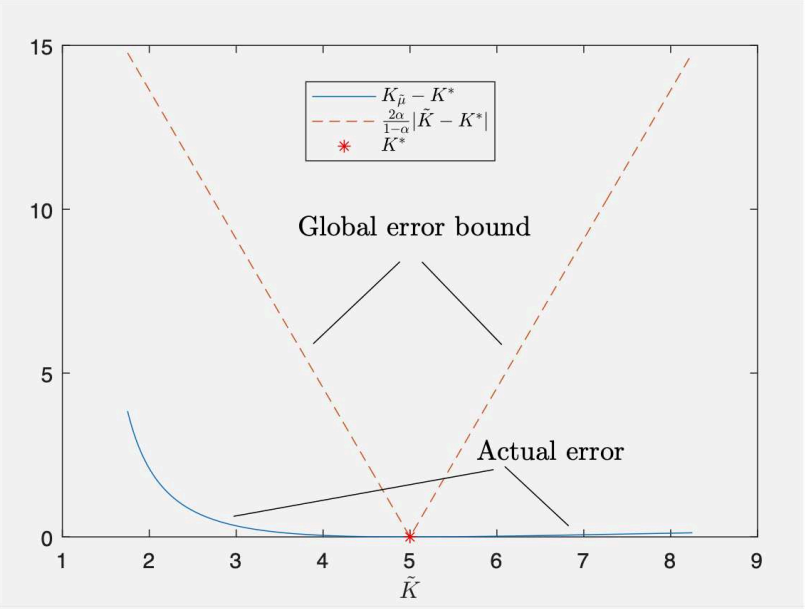


Figure 1.5.10 Illustration of the global error bound (1.54) for the one-step lookahead error $K_{\tilde{\mu}} - K^*$ as a function of $\tilde{K}$, compared with the true error obtained by one step of Newton's method starting from $\tilde{K}$.

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 (1.54).

1.5.3 Rollout and Policy Iteration

The rollout algorithm starts from some linear stable base policy μ, and obtains the rollout policy $\tilde{\mu}$ using a policy improvement operation, which by definition, yields the one-step lookahead policy that corresponds to terminal cost approximation J_μ. Figure 1.5.11 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 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

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

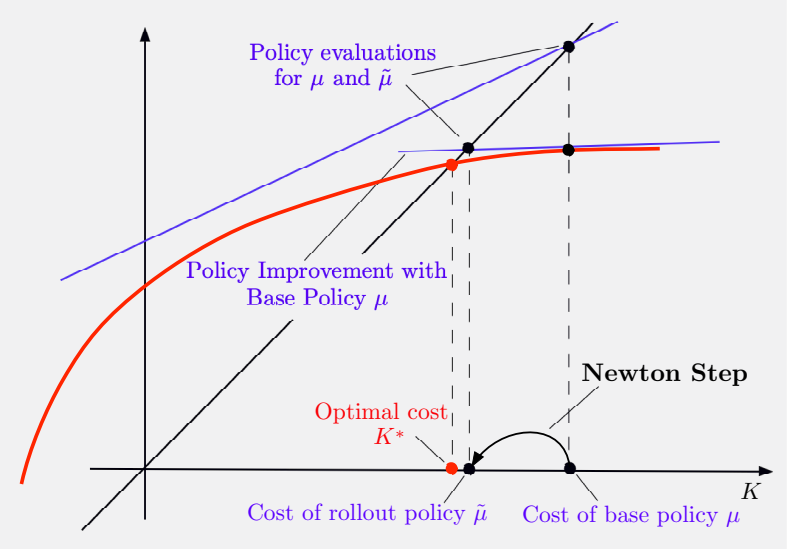


Figure 1.5.11 Illustration of the rollout algorithm for the linear quadratic problem. Starting from a linear stable base policy μ, it generates a stable rollout policy $\tilde{\mu}$. The quadratic cost coefficient of $\tilde{\mu}$ is obtained from the quadratic cost coefficient of μ with a Newton step for solving the Riccati equation.

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

$$J_{\mu^0}(x) = K_0 x^2, \tag{1.55}$$

where

$$K_0 = \frac{q + rL_0^2}{1 - (a + bL_0)^2}. \tag{1.56}$$

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

$$L_1 = -\frac{abK_0}{r + b^2 K_0}; \tag{1.57}$$

cf. Eqs. (1.45)-(1.46).

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

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

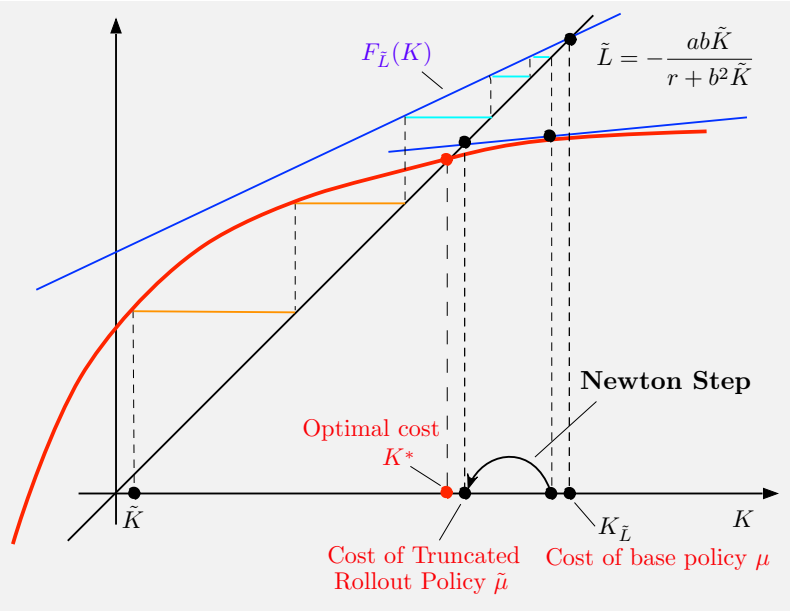


Figure 1.5.12 Illustration of truncated rollout with a stable base policy $\mu(x) = Lx$ and terminal cost approximation $\tilde{K}$ for the linear quadratic problem. In this figure, we use one-step lookahead minimization and the number of rollout steps is $m = 4$.

with K_k given by

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

[cf. Eqs. (1.56)-(1.57)].

The corresponding cost function sequence has the form

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

cf. Eq. (1.55). 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].

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. 1.5.12. 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:

(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. 1.5.12. They are also consistent with the results of computational experimentation. We refer to the monograph [Ber22a] for further discussion.

Newton Step Interpretation of Approximation in Value Space in General

The interpretation of approximation in value space as a Newton step, and related notions of stability and error bounds that we have discussed in this section admit a broad generalization to the infinite horizon problems that we consider in these notes 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, and the region of stability; see the book [Ber22a] for an extensive discussion, and Section 1.6.7 for an example in the context of MPC. Naturally, the graphical interpretations and visualizations are limited to one dimension. However, the visualizations provide insight and motivate conjectures and mathematical analysis, some of which is given in the book [Ber20a].

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 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 these notes, 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 these notes. 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 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 approximations and 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 problems*. However, there are enough methods to try on a given problem with a reasonable chance of success in the end. For this reason, it is important to develop insight into the inner workings of each type of method, as a means of selecting the proper type of methods to try on a given problem.†

A related consideration is the context within which a particular 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? Moreover, are the problem data fixed or may they change over time as the system is being controlled? Generally, RL methods that require extensive tuning of parameters, including ones that involve approximation in policy space and the use of neural

† 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, delineate the sound from the flawed ideas.

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 environement.

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 is that even if the termination state t is reached at a time $k < N$, we can

† 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.

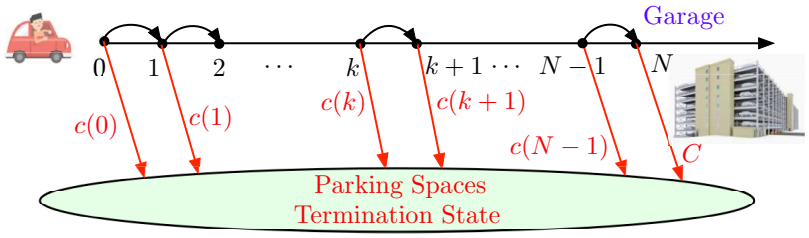


Figure 1.6.1 Cost structure of the parking problem. The driver may park at space $k = 0, 1, \ldots, N-1$ at cost $c(k)$, if the space is free, or continue to the next space $k+1$ at no cost. At space N (the garage) the driver must park at cost C.

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.

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. The reformulation 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

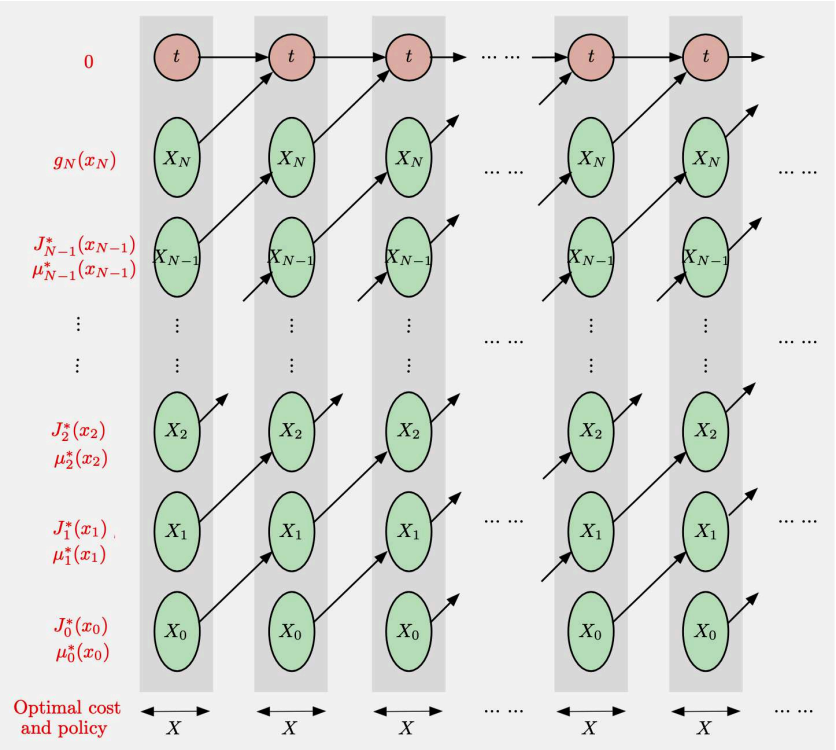


Figure 1.6.2 Illustration of the infinite horizon equivalent of a finite horizon problem. The state space is $X = \left(\cup_{k=0}^{N} X_k\right) \cup \{t\}$, and the control space is $U = \cup_{k=0}^{N-1} U_k$. Transitions from states $x_k \in X_k$ lead to states in $x_{k+1} \in X_{k+1}$ according to 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.

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

$$x_{k+1} = f_k(x_k, u_k, w_k), \qquad k = 0, \ldots, N-1, \tag{1.58}$$

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

$$X = \left(\cup_{k=0}^{N} X_k\right) \cup \{t\}, \qquad U = \cup_{k=0}^{N-1} U_k; \tag{1.59}$$

see Fig. 1.6.2.

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. (1.58), 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. 1.6.2. 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. (1.59), as illustrated in Fig. 1.6.2. 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.

1.6.3 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

$$x_{k+1} = f_k(x_k, x_{k-1}, u_k, u_{k-1}, w_k), \qquad k = 1, \ldots, N-1, \tag{1.60}$$

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

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 (1.60) yields

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \\ z_{k+1} \end{pmatrix} = \begin{pmatrix} f_k(x_k, y_k, u_k, z_k, w_k) \\ x_k \\ u_k \end{pmatrix}. \tag{1.61}$$

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. (1.61).

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

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

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

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

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

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

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

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

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

$$J_k^*(x_k, y_k) = \min_{u_k \in U_k(x_k)} E_{w_k} \Bigg\{ g_k(x_k, u_k, w_k) + \sum_{i=1}^{m} p_i J_{k+1}^*\big(f_k(x_k, u_k, w_k), i\big) \;\Big|\; y_k \Bigg\}, \tag{1.62}$$

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:

$$\hat{J}_k(x_k) = E_{y_k}\Bigg\{\min_{u_k \in U_k(x_k, y_k)} E_{w_k}\Big\{g_k(x_k, y_k, u_k, w_k) + \hat{J}_{k+1}\big(f_k(x_k, y_k, u_k, w_k)\big) \mid x_k, y_k, u_k\Big\}\Big| \, x_k\Bigg\}. \tag{1.63}$$

This is a consequence of the calculation

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

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 (1.63) 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. (1.63). 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. (1.62),

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

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. (1.62).

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

Example 1.6.2 (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. 1.6.3). 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

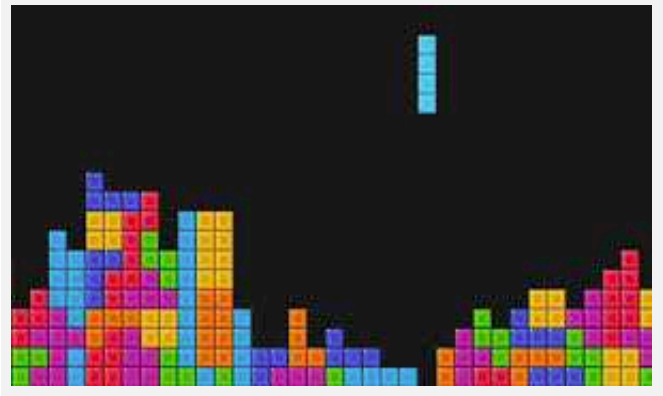


Figure 1.6.3 Illustration of a tetris board.

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.

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

$$\hat{J}_k(x) = \sum_y p(y) \max_u \Big[g(x,y,u) + \hat{J}_{k+1}\big(f(x,y,u)\big) \Big], \qquad \text{for all } x, \qquad (1.64)$$

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 (1.64) 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 (1.64), 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.†

1.6.4 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

† Tetris is generally considered as 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.

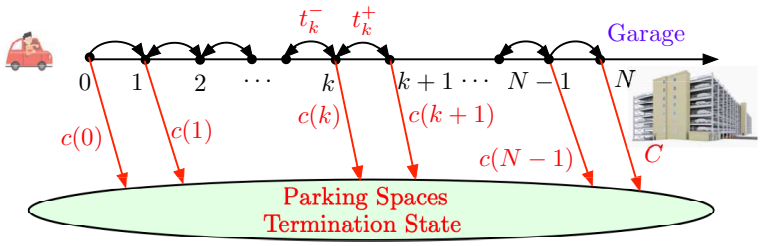


Figure 1.6.4 Cost structure and transitions of the bidirectional parking problem. The driver may park at space $k = 0, 1, \ldots, N-1$ at cost $c(k)$, if the space is free, can move to $k-1$ at cost t_k^- or can move to $k+1$ at cost t_k^+. At space N (the garage) the driver must park at cost C.

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 these notes; see e.g., the DP textbook [Ber17a].

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 1.6.3 (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. 1.6.4.

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

$$J_k^*(b_k) = \min_{u_k \in U_k} \left[\hat{g}_k(b_k, u_k) + E_{z_{k+1}} \Big\{ J_{k+1}^*\big(F_k(b_k, u_k, z_{k+1})\big) \;\Big|\; b_k, u_k \Big\} \right], \tag{1.65}$$

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:

$$b_{k+1} = F_k(b_k, u_k, z_{k+1}). \tag{1.66}$$

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

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

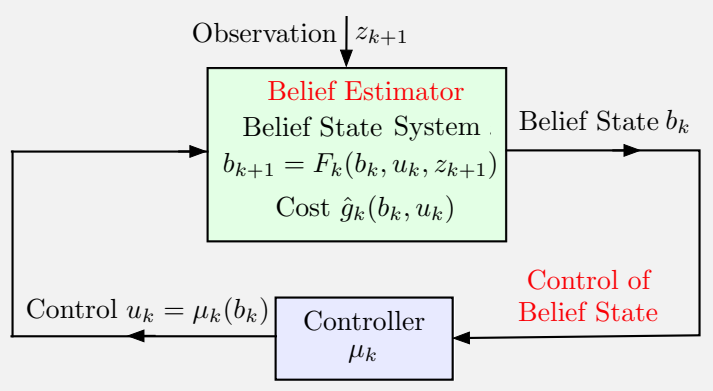


Figure 1.6.5 Schematic illustration of the view of an imperfect state information problem as one of perfect state information, whose state is the belief state b_k, i.e., the conditional probability distribution of x_k given all the observations up to time k. The observation z_{k+1} plays the role of the stochastic disturbance. The function F_k is a sequential estimator that updates the current belief state b_k.

in Eq. (1.65) is a function of b_k and u_k.

The algorithm (1.65) is just the ordinary DP algorithm for the perfect state information problem shown in Fig. 1.6.5. It involves the system/belief estimator (1.66) 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 (1.65) will typically require an approximate implementation, using approximation in value space methods.

We refer to the textbook [Ber17a], Chapter 4, for a detailed derivation of the DP algorithm (1.65), 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 (1.65) 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

$$J_k^*(I_k) = \min_{u_k \in U_k(x_k)} E_{w_k, z_{k+1}} \Big\{ g_k(x_k, u_k, w_k) + J_{k+1}^*\big(I_k, z_{k+1}, u_k\big) \mid I_k, u_k \Big\}, \tag{1.67}$$

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.

1.6.5 Multiagent Problems and Multiagent Rollout

In these notes, 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

† 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.

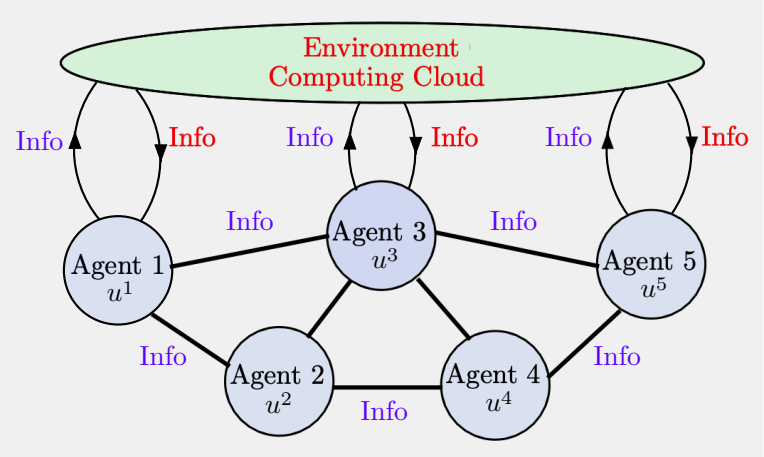


Figure 1.6.6 Schematic illustration of a multiagent problem. There are multiple "agents," and each agent $\ell = 1, \ldots, m$ controls its own decision variable u^ℓ. At each stage, agents exchange new information and also exchange information with the "environment," and then select their decision variables for the stage.

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

$$U(x) = U^1(x) \times \cdots \times U^m(x), \tag{1.68}$$

where the sets $U^\ell(x)$ are given. This structure arises in applications involving distributed decision making by multiple agents; see Fig. 1.6.6.

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

$$\tilde{u} \in \arg\min_{u \in U(x)} E_w\Big\{g(x,u,w) + \alpha J_\mu\big(f(x,u,w)\big)\Big\}, \tag{1.69}$$

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 (1.68)]. 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 (1.69) 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

$$\tilde{\mu}^1(x) \in \arg\min_{u^1 \in U^1(x)} E_w\Big\{g\big(x, u^1, \mu^2(x), \ldots, \mu^m(x), w\big) + \alpha J_\mu\big(f(x, u^1, \mu^2(x), \ldots, \mu^m(x), w)\big)\Big\},$$

$$\tilde{\mu}^2(x) \in \arg\min_{u^2 \in U^2(x)} E_w\Big\{g\big(x, \tilde{\mu}^1(x), u^2, \mu^3(x) \ldots, \mu^m(x), w\big) + \alpha J_\mu\big(f(x, \tilde{\mu}^1(x), u^2, \mu^3(x), \ldots, \mu^m(x), w)\big)\Big\},$$

$$\cdots \qquad \cdots \qquad \cdots \qquad \cdots$$

$$\tilde{\mu}^m(x) \in \arg\min_{u^m \in U^m(x)} E_w\Big\{g\big(x, \tilde{\mu}^1(x), \tilde{\mu}^2(x), \ldots, \tilde{\mu}^{m-1}(x), u^m, w\big) + \alpha J_\mu\big(f(x, \tilde{\mu}^1(x), \tilde{\mu}^2(x), \ldots, \tilde{\mu}^{m-1}(x), u^m, w)\big)\Big\}.$$

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 (1.69) 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"

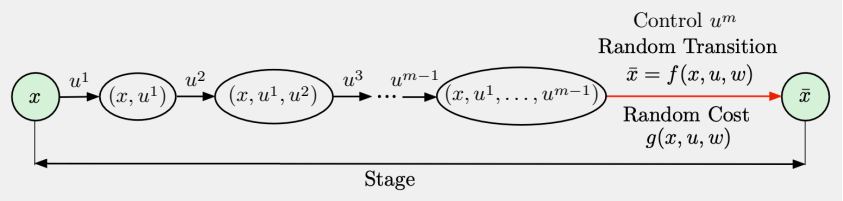


Figure 1.6.7 Equivalent formulation of the stochastic optimal control problem for the case where the control u consists of m components $u^1, u^2, \ldots, u^m$:

$$u = (u^1, \ldots, u^m) \in U(x) = U^1(x) \times \cdots \times U^m(x).$$

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.

$(x, u^1, \ldots, u^{m-1})$ 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. 1.6.7.

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.

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 (1.69) is performed for just one state at each stage.

The major fact that can be proved about multiagent rollout (see 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

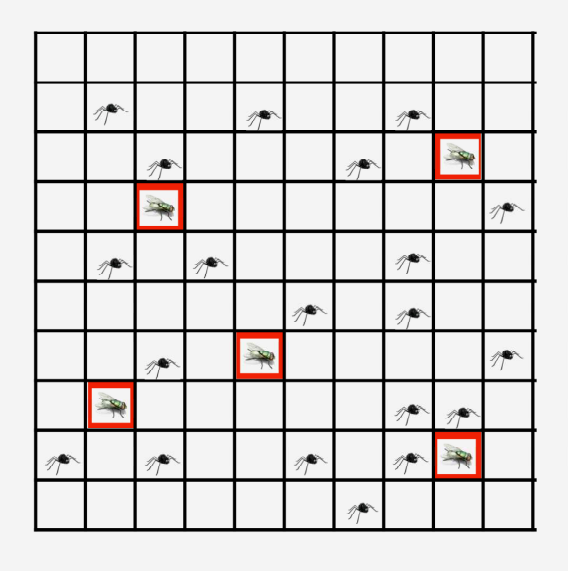


Figure 1.6.8 Illustration of a 2-dimensional spiders-and-fly problem with 20 spiders and 5 flies (cf. Example 1.6.4). The flies moves randomly, regardless of the position of the spiders. During a stage, each spider moves to a neighboring location or stays where it is, so there are 5 moves per spider (except for spiders at the edges of the grid). The total number of possible joint spiders moves is a little less than 5^{20}.

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 various related PI schemes that are well suited for distributed operation in important practical contexts involving multiple autonomous decision makers; see the book [Ber20a], the papers [Ber20b], and the tutorial survey [Ber21a].

Example 1.6.4 (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 and rescue, firefighting, etc. Here there are m spiders and several flies moving on a 2-dimensional grid; cf. Fig. 1.6.8. 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.

1.6.6 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 oversim-

† The difficulties of decision and control within a changing environment are often underestimated. Among others, they complicate the balance between off-

plified cruise control system that we noted in Example 1.3.1 or its infinite horizon version. The state evolves according to

$$x_{k+1} = x_k + bu_k + w_k, \tag{1.70}$$

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 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

---

line 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!

† 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.

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. (1.70) 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 control in these notes 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. 1.6.9.

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

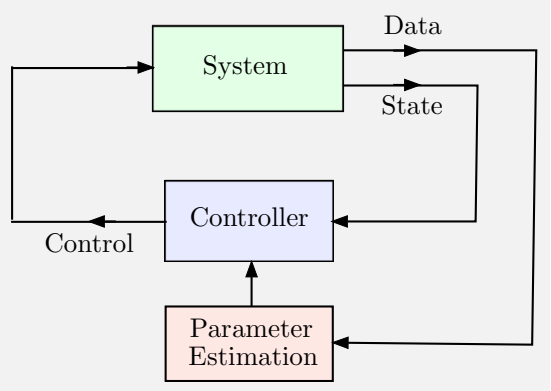


Figure 1.6.9 Schematic illustration of concurrent parameter estimation and system control. The system parameters are estimated on-line and the estimates are periodically passed on to the controller.

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 1.6.5 (Parameter Identifiability Under Closed-Loop Control)

For a simple example, consider the scalar system

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

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

$$\tilde{\mu}_k(x_k) = \gamma x_k \tag{1.71}$$

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 (1.71), 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 (1.71) 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 these notes. 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

† 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.

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 1.6.4).

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 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. 1.6.10).

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.

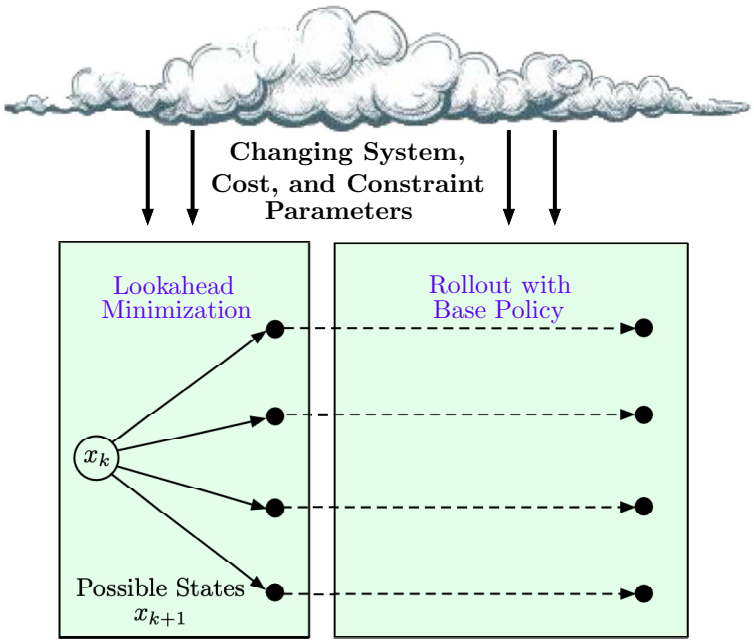


Figure 1.6.10 Schematic illustration of adaptive control by rollout. One-step lookahead is followed by simulation with the base policy, which stays fixed. The system, cost, and constraint parameters are changing over time, and the most recent values are incorporated into the lookahead minimization and 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.

Example 1.6.6 (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

$$K = \frac{rK}{r + b^2 K} + 1. \tag{1.72}$$

The optimal policy has the form

$$\mu^*(x) = L^* x, \tag{1.73}$$

where

$$L^* = -\frac{bK^*}{r + b^2 K^*}. \tag{1.74}$$

As an example, consider the optimal policy that corresponds to the nominal problem parameters $b = 2$ and $r = 0.5$: this is the policy (1.73)-(1.74), with K obtained as the positive solution of the quadratic Riccati Eq. (1.72) for $b = 2$ and $r = 0.5$. In particular, we can verify that

$$K = \frac{2 + \sqrt{6}}{4}.$$

From Eq. (1.74) we then obtain

$$L = -\frac{2 + \sqrt{6}}{5 + 2\sqrt{6}}. \tag{1.75}$$

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. (1.72).

(b) The cost function $K_L x^2$ that corresponds to the base policy

$$\mu_L(x) = Lx,$$

where L is given by Eq. (1.75). From our earlier discussion, we have

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

(c) The cost function $\tilde{K}_L x^2$ 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

$$\tilde{L} = -\frac{bK_L}{r + b^2 K_L},$$

and

$$\tilde{K}_L = \frac{1 + r\tilde{L}^2}{1 - (1 + b\tilde{L})^2}.$$

Figure 1.6.11 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 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. 1.6.11 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.

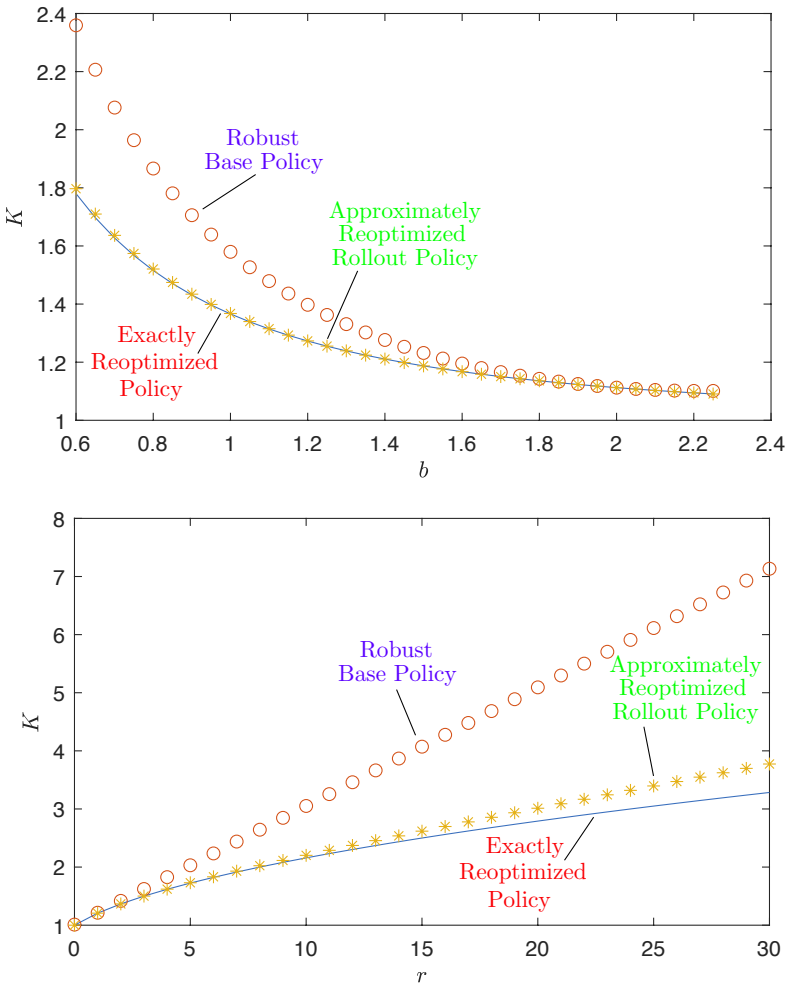


Figure 1.6.11 Illustration of adaptive control by rollout under changing problem parameters. The quadratic cost coefficients K^* (optimal, denoted by solid line), K_L (base policy, denoted by circles), and $\tilde{K}_L$ (rollout policy, denoted by asterisks) for the two cases where $r = 0.5$ and b varies, 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. (1.75)].

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.

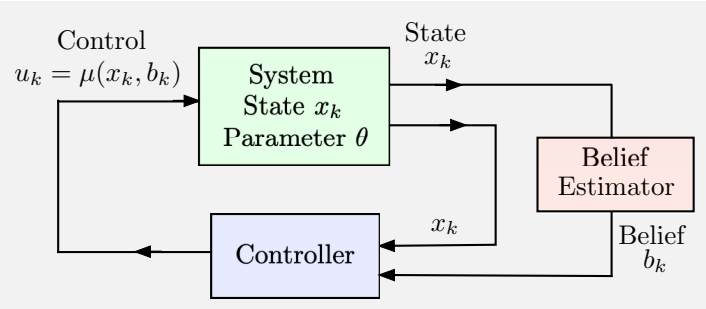


Figure 1.6.12 Schematic illustration of simultaneous control and belief estimation for the unknown system parameter θ. The control applied is a function of the current belief state (x_k, b_k), where b_k is the conditional probability distribution of θ given the observations accumulated up to time k (the current and past states $x_k, \ldots, x_0$, and the past controls $u_{k-1}, \ldots, u_0$).

We view θ as a parameter in the system equation that governs the evolution of x_k. Thus we have

$$x_{k+1} = f_k(x_k, \theta, u_k, w_k), \tag{1.76}$$

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 $\theta^1, \ldots, \theta^m$.

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. 1.6.12.

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.

1.6.7 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. 1.6.13). Another type of classical control problem is to keep the system close to a given trajectory (see Fig. 1.6.14) can also be treated by forms of MPC, but will not be discussed in these notes.

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 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. 1.6.14). 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

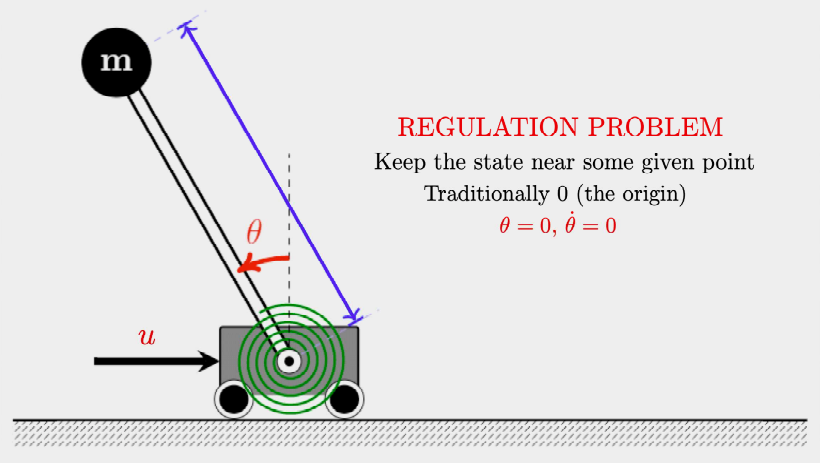


Figure 1.6.13 Illustration of a classical regulation problem, known as the "cart-pole problem" or "inverse pendulum problem." The state is the two-dimensional vector of angular position and angular velocity. We aim to keep the pole at the upright position (state equal to 0) by exerting horizontal force u on the cart.

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

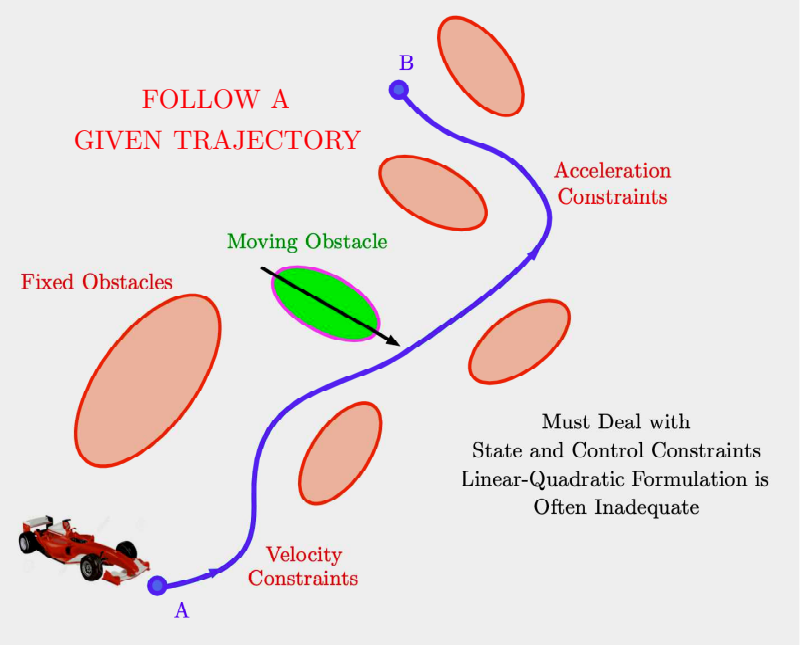


Figure 1.6.14 Illustration of constrained motion of a car from point A to point B. There are state (position/velocity) constraints, and control (acceleration) constraints. When there are mobile obstacles, the state constraints may change unpredictably, necessitating on-line replanning.

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,

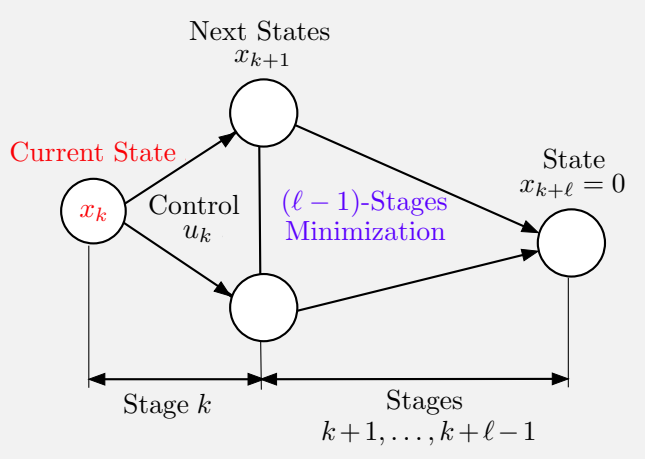


Figure 1.6.15 Illustration of the problem solved by a classical form of MPC at state x_k. We minimize the cost function over the next ℓ stages while imposing the requirement that $x_{k+\ell}=0$. We then apply the first control of the optimizing sequence. In the context of rollout, the minimization over 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.

at each encountered state x_k, we apply a control $\tilde{u}_k$ that is computed as follows; see Fig. 1.6.15:

(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

$$\min_{u_t,\, t=k,\ldots,k+\ell-1} \sum_{t=k}^{k+\ell-1} g(x_t,u_t), \tag{1.77}$$

subject to the system equation constraints

$$x_{t+1}=f(x_t,u_t), \qquad t=k,\ldots,k+\ell-1, \tag{1.78}$$

the control constraints

$$u_t\in U(x_t), \qquad t=k,\ldots,k+\ell-1, \tag{1.79}$$

and the terminal state constraint

$$x_{k+\ell}=0. \tag{1.80}$$

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. (1.77)] 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. 1.5.11.

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 (1.77), 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

$$\min_{u_t,\, t=k,\ldots,k+\ell-1} \left[G(x_{k+\ell}) + \sum_{t=k}^{k+\ell-1} g(x_t, u_t) \right], \tag{1.81}$$

instead of problem (1.77) 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.

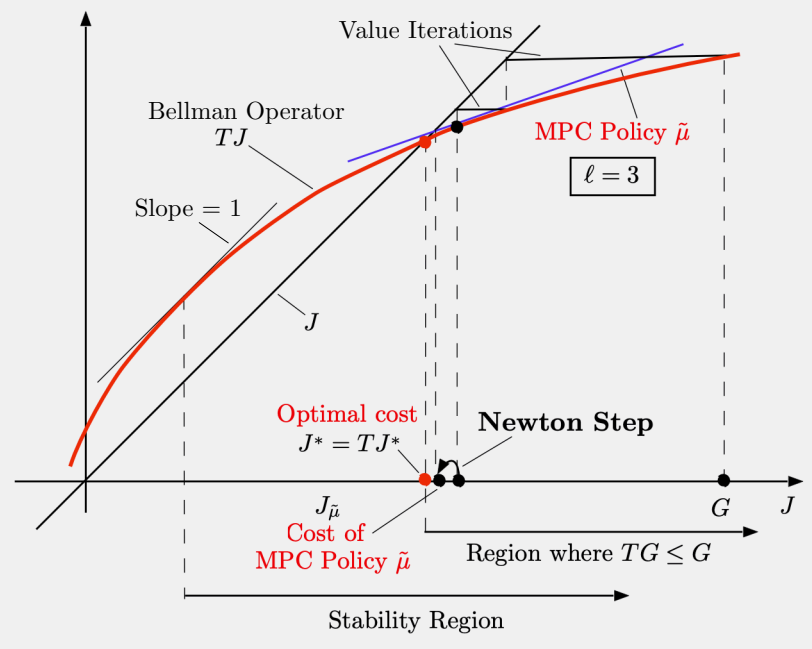


Figure 1.6.16 Illustration of the Bellman operator, defined by

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

The condition in (1.82) 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, $\ell = 3$.

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:

$$\min_{u \in U(x)} \Big\{ g(x,u) + G\big(f(x,u)\big) \Big\} \leq G(x), \qquad \text{for all } x. \tag{1.82}$$

Figure 1.6.16 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

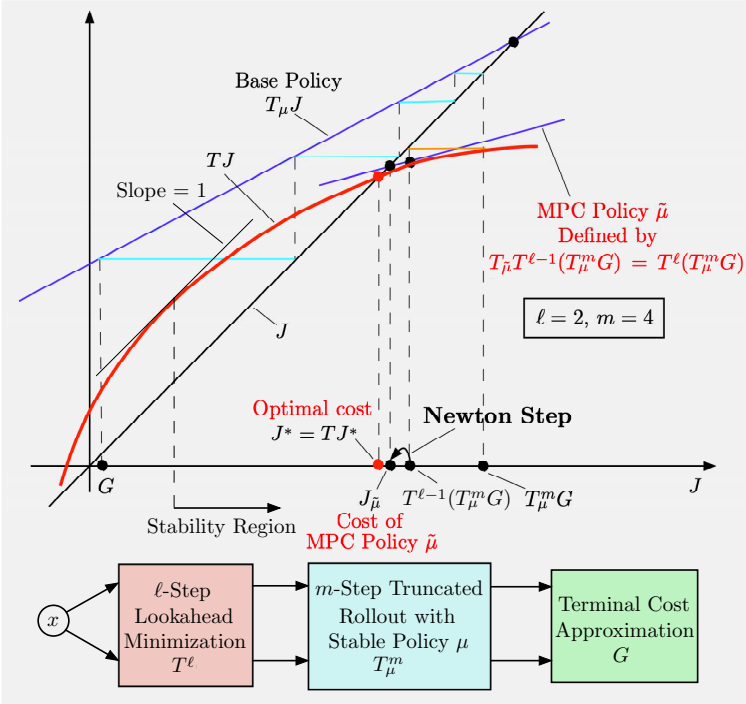


Figure 1.6.17 An MPC scheme with ℓ-step lookahead minimization, m-step truncated rollout with a stable base policy μ, and a terminal cost function approximation G, together with its interpretation as a Newton step. In this figure, $\ell = 2$ and $m = 4$. The truncated rollout with base policy μ consists 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.

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 stable base policy μ for a number of steps m; see Fig. 1.6.17. This is quite similar to standard truncated rollout, except that the computational solution of the lookahead minimization problem (1.81) 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. (1.81)] must be modified. Assuming that the current state x_k belongs to the constraint set X, the MPC problem should take the form

$$\min_{u_t,\, t=k,\ldots,k+\ell-1} \left[G(x_{k+\ell}) + \sum_{t=k}^{k+\ell-1} g(x_t, u_t) \right], \tag{1.83}$$

subject to the control constraints

$$u_t \in U(x_t), \qquad t = k, \ldots, k+\ell-1, \tag{1.84}$$

and the state constraints

$$x_t \in X, \qquad t = k+1, \ldots, k+\ell. \tag{1.85}$$

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 (1.83)-(1.85) has a feasible solution for all initial states* $x_k \in X$. Here is a simple example.

Example 1.6.7 (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

$$X = \big\{x_k \mid |x_k| \leq \beta\big\}. \tag{1.86}$$

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

$$\hat{X} = \big\{x_k \mid |x_k| \leq 1\big\};$$

see Fig. 1.6.18, which also illustrates the trajectories generated by the MPC scheme of Eq. (1.81), 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. (1.86) is invariant if and only if $\beta \leq 1$.

Given an MPC calculation of the form (1.83)-(1.85), 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 (1.83)-(1.85) will be feasible provided the initial state x_0 belongs to $\hat{X}$.

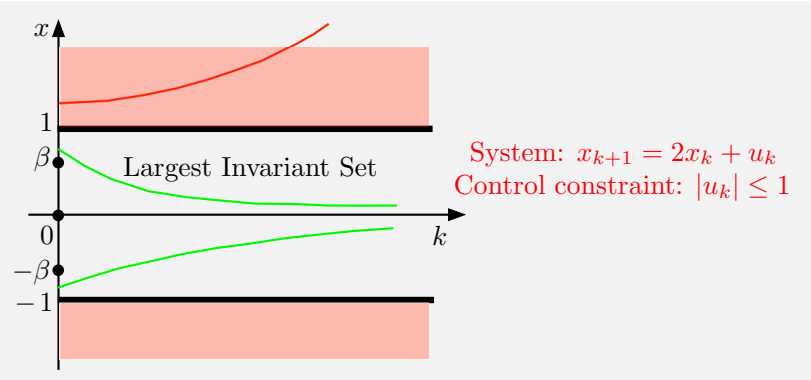


Figure 1.6.18 An illustration of invariance of a state constraint set X. Here the sets of the form $X = \{x_k \mid |x_k| \leq \beta\}$ are invariant for $\beta \leq 1$. For $\beta = 1$, we obtain the largest invariant set (the one that contains all other invariant sets). The figure shows some state trajectories 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.

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

$$X_{k+1} = \{x \in X_k \mid f(x,u) \text{ belongs to } X_k \text{ for some } u \in U(x)\}, \quad k \geq 0. \tag{1.87}$$

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 (1.87) starting with a set $X_0 = \{x \mid |x| \leq \beta\}$ with $\beta > 1$ is given by

$$X_k = \{x \mid |x| \leq \beta_k\}, \quad \text{with } \beta_0 = \beta \text{ and } \beta_{k+1} = \tfrac{\beta_k + 1}{2} \text{ for all } k \geq 0.$$

† The term used in [Ber71] and [Ber72a] is *reachability of a target tube* $\{X, X, \ldots\}$, which is synonymous to invariance of X.

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 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].

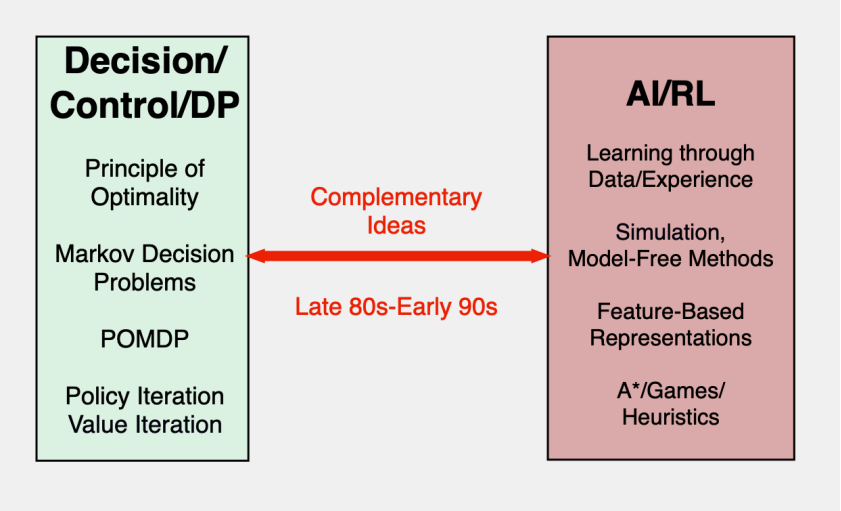


Figure 1.7.1 A schematic illustration of the synergy of ideas between artificial intelligence on one hand, and decision and control on the other.

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 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 Terminology

The terminology used in these notes 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 Notation

Unfortunately, the confusion arising from different terminology has been exacerbated by the use of different notations. The present notes roughly follow 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 these notes. 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.† However, 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 these notes.

(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

† 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 these class notes.

‡ A variant of RL called *offline RL* or *batch RL*, starts from a historical dataset, and does not explore the environment to colect new data.

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 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 these class notes 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.

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.

Notice a qualitative difference between optimization and machine learning: *the former is mostly organized around mathematical structures and the analysis 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 a fundamental distinction, which affects profoundly the perspectives of researchers in the two fields.

Relations Between RL and DP Methodologies and Applications

In comparing the RL and DP methodologies, we should note that they are fundamentally connected through their corresponding problem formulations: 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,

† 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."

large language models, and automated planning, where they can have a significant practical impact.

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 broadly 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

† 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.

considered, there may be differences in the emphasis and priority placed on mathematical analysis and generality versus practical effectiveness and efficiency. This is particularly true in certain specialized contexts, and can lead to some tension, as each field may not fully appreciate the other's perspective.

1.8 NOTES, SOURCES, AND EXERCISES

We will now summarize this first chapter and describe how it can be used as a foundational platform for a few different courses. We will also provide a selective overview of the DP and RL literature, and also provide 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 fundamental 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 hinted at several points some of the prin-

cipal schemes for approximation, e.g., neural networks and feature-based architectures.

One of the principal aims of this first 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 are focusing selectively on the methodologies (1)-(4). In a different course, other choices from the above list may be made, by builiding on the content of the current chapter.

Notes and Sources for Individual Sections

In the literature survey 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 notes (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 these notes 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 by the author [Ber12] provide detailed 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 the present notes), and this is fine because they do not play

† The rigorous mathematical theory of stochastic optimal control, including the development of an appropriate measure-theoretic framework, dates to the 60s and 70s. 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 the fact 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.

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 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 these notes in the context of linear quadratic problems.

Approximation in value space, rollout, and policy iteration are the principal subjects of these notes.† 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. Rollout is reliable, easy to implement, and can be used in conjunction with on-line replanning.

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 P 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,

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 author's monograph [Ber22a] describes research that connects policy iteration with Newton's method, together with convergence analysis of variants of Newton's method applied to the solution of nondifferentiable fixed point problems.

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.

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

using so-called "truncated rollouts." In this technique trials are not played out 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.

Multiagent problem research 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]. 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 notes.

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].

A computational study where several of the multiagent algorithmic ideas were tested and validated is the paper by Bhattacharya et al. [BKB20]. This paper considers a large-scale multi-robot routing and repair problem, involving partial state information, and explores some of the attendant implementation issues, including autonomous multiagent rollout, through the use of policy neural networks and other precomputed signaling policies.

The subject of adaptive control has a long history and its literature is very extensive; see the books by Aström and Wittenmark [AsW94], Aström and Hagglund [AsH95], [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 (also known as the "dual control problem"), 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 research on problems involving unknown models and using data for model identification prior to or 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 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]. The connections between MPC and rollout were discussed in the author's survey [Ber05a]. 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].

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 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 these notes, 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 also of major interest in these notes.

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 notes, 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 notes. 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 notes are 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 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 covers substantially policy gradient methods for approximation in policy space, which we will not address here. 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 notes 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 several 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 these notes: 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 focuses on continuous spaces optimal control), and the author's [Ber05a] (which focuses on rollout algorithms and model predictive control), [Ber11a] (which focuses on approximate policy iteration), [Ber18a] (which focuses on aggregation methods), and [Ber20b] (which focuses on multiagent problems).

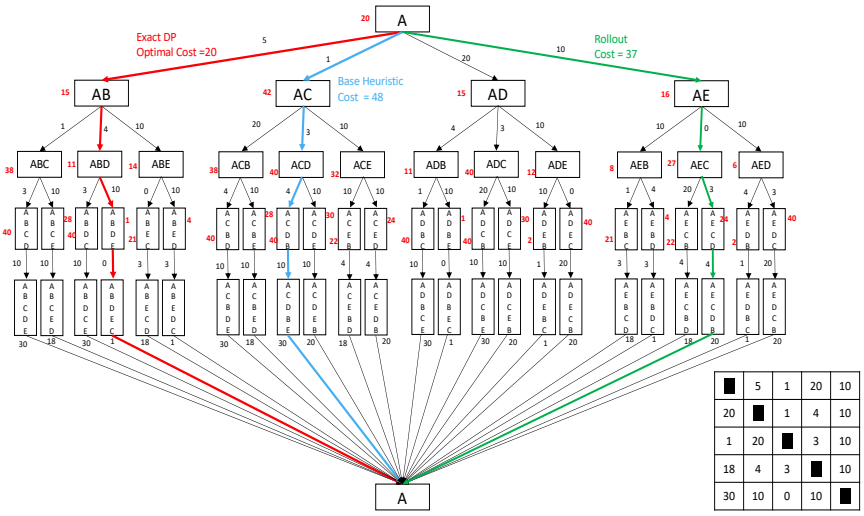


Figure 1.8.1 Solution of parts (a), (b), and (c) of Exercise 1.1. A 5-city traveling salesman problem illustration of rollout with the nearest neighbor base heuristic.

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 - A Stochastic Investment Problem)

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 the following problem.

An investor wants 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:

$$J_N^*(x_N) = x_N, \tag{1.88}$$

and for $k = 0, \ldots, N-1$, if $x_k = 0$, then

$$J_k^*(0) = p^+ J_{k+1}^*(1) + (1 - p^+) J_{k+1}^*(0), \tag{1.89}$$

if $x_k = \bar{x}$, then

$$J_k^*(\bar{x}) = \bar{x}, \tag{1.90}$$

(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

$$J_k^*(x_k) = \max\left\{x_k,\, p^+ J_{k+1}^*(x_k+1) + (1-p^+-p^-) J_{k+1}^*(x_k) + p^- J_{k+1}^*(x_k-1)\right\}. \tag{1.91}$$

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.

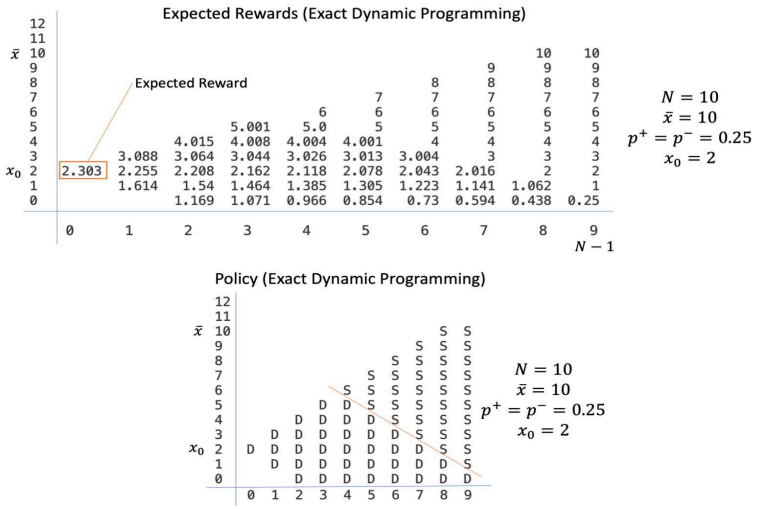


Figure 1.8.2 Table of values of optimal reward-to-go, obtained by exact DP, and corresponding optimal policy [cf. the algorithm (1.88)-(1.91). Only the states x_k that are reachable from x_0 at time k are considered (this is the state space for time k).

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

$$N = 10, \quad x_0 = 20, \quad \bar{x} = 40, \quad p^+ = p^- = 0.25.$$

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.

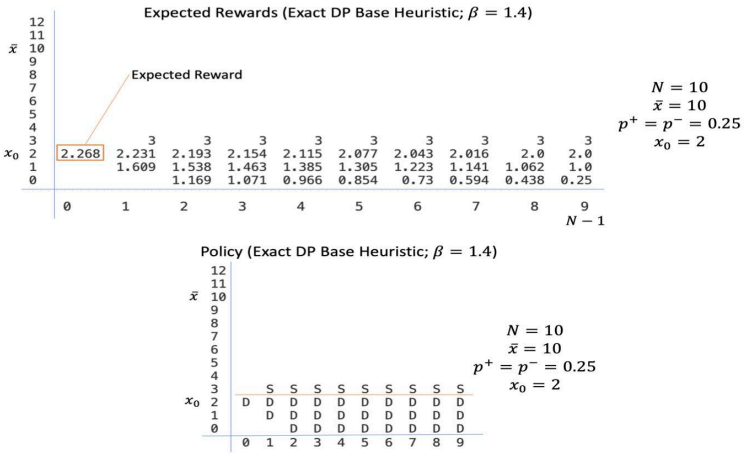


Figure 1.8.3 Table of rewards-to-go for the base policy with $\beta = 1.4$, starting from x_0 [cf. the algorithm (1.92)-(1.95) for $k = 0$].

(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 these notes.) The algorithm is given by

$$J_N^{x_k}(x_N) = x_N, \tag{1.92}$$

and for $n = k, \ldots, N-1$, if $0 < x_n < \beta x_k$, then

$$J_n^{x_k}(x_n) = p^+ J_{n+1}^{x_k}(x_n+1) + (1-p^+-p^-) J_{n+1}^{x_k}(x_n) + p^- J_{n+1}^{x_k}(x_n-1), \tag{1.93}$$

if $x_n = 0$, then

$$J_n^{x_k}(0) = p^+ J_{n+1}^{x_k}(1) + (1-p^+) J_{n+1}^{x_k}(0), \tag{1.94}$$

and if $x_n \geq \beta x_k$, then

$$J_n^{x_k}(x_n) = x_n. \tag{1.95}$$

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.

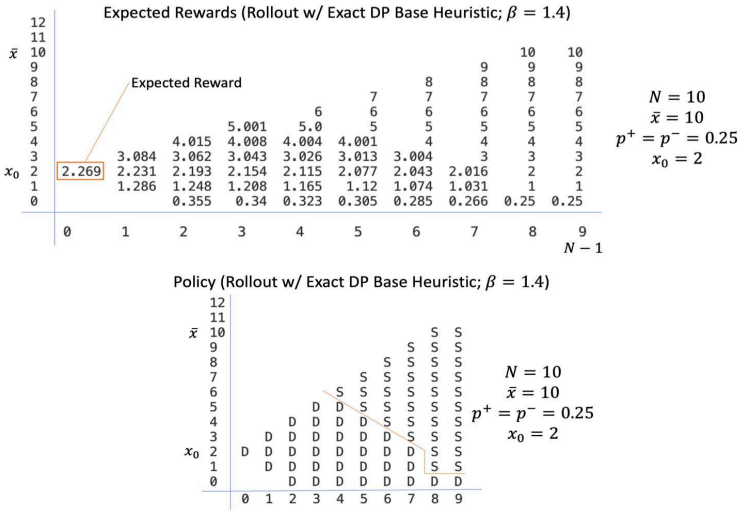


Figure 1.8.4 Table of values of reward-to-go and decisions applied by the rollout policy that corresponds to the base heuristic with $\beta = 1.4$.

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.

(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 $2\ell + 1$.

1.3 (Computational Exercise - Spiders and Flies)

Consider the spiders and flies problem of Example 1.6.4 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. 1.6.8. 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 - 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.

1.5 (Computational Exercise - Parking Problem)

In reference to Example 1.6.3, 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.

1.6 (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

$$K_{k+1} = K_k - \left(\frac{\partial H(K_k)}{\partial K}\right)^{-1} H(K_k), \tag{1.96}$$

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. (1.40), which we write in the form

$$H(K) = 0,$$

where

$$H(K) = K - \frac{a^2 rK}{r + b^2 K} - q. \tag{1.97}$$

(a) Show that the operation that generates K_L starting from K is a Newton iteration of the form (1.96). In other words, show that for all K that lead to a stable one-step lookahead policy, we have

$$K_L = K - \left(\frac{\partial H(K)}{\partial K}\right)^{-1} H(K), \tag{1.98}$$

where we denote by

$$K_L = \frac{q + rL^2}{1 - (a + bL)^2} \tag{1.99}$$

the quadratic cost coefficient of the one-step lookahead linear policy $\mu(x) = Lx$ corresponding to the cost function approximation $J(x) = Kx^2$:

$$L = -\frac{abK}{r + b^2 K}. \tag{1.100}$$

Proof: Our approach for showing the Newton step formula (1.98) 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. (1.100) that K can be expressed in terms of L as

$$K = -\frac{rL}{b(a+bL)}. \tag{1.101}$$

Furthermore, by using Eqs. (1.100) and (1.101), $H(K)$ as given in Eq. (1.97) can be expressed in terms of L as follows:

$$H(K) = -\frac{rL}{b(a+bL)} + \frac{arL}{b} - q. \tag{1.102}$$

Moreover, by differentiating the function H of Eq. (1.97), we obtain after a straightforward calculation

$$\frac{\partial H(K)}{\partial K} = 1 - \frac{a^2r^2}{(r+b^2K)^2} = 1 - (a+bL)^2, \tag{1.103}$$

where the second equation follows from Eq. (1.100). Having expressed all the terms in the Newton step formula (1.98) in terms of L through Eqs. (1.99), (1.101), (1.102), and (1.103), 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

$$|K_{\tilde{L}} - K^*| \le c\left|F^{\ell-1}(\tilde{K}) - K^*\right|^2,$$

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.

1.7 (Post-Decision States)

The purpose of this exercise is to demonstrate a type of DP simplification that arises often (see [Ber12], Section 6.1.5 for further discussion). Consider the finite horizon stochastic DP problem and assume that the system equation has a special structure whereby from state x_k after applying u_k we move to an intermediate "post-decision state"

$$y_k = p_k(x_k, u_k)$$

at cost $g_k(x_k, u_k)$. Then from y_k we move at no cost to the new state x_{k+1} according to

$$x_{k+1} = h_k(y_k, w_k) = h_k\big(p_k(x_k, u_k), w_k\big), \tag{1.104}$$

where the distribution of the disturbance w_k depends only on y_k, and not on prior disturbances, states, and controls. Denote by $J_k(x_k)$ the optimal cost-to-go starting at time k from state x_k, and by $V_k(y_k)$ the optimal cost-to-go starting at time k from post-decision state y_k.

(a) Use Eq. (1.104) to verify that a DP algorithm that generates only J_k is given by

$$J_k(x_k) = \min_{u_k \in U_k(x_k)} \Big[g(x_k, u_k) + E_{w_k}\Big\{ J_{k+1}\big(h_k(p_k(x_k, u_k), w_k)\big)\Big\}\Big].$$

(b) Show that a DP algorithm that generates both J_k and V_k is given by

$$J_k(x_k) = \min_{u_k \in U_k(x_k)} \Big[g(x_k, u_k) + V_k\big(p_k(x_k, u_k)\big)\Big],$$

$$V_k(y_k) = E_{w_k}\Big\{ J_{k+1}\big(h_k(y_k, w_k), w_k\big)\Big\}.$$

(c) Show that a DP algorithm that generates only V_k for all k is given by

$$V_k(y_k) = E_{w_k}\Big\{ \min_{u_{k+1} \in U_{k+1}(h_k(y_k, w_k))} \big[g_{k+1}(h_k(y_k, w_k), u_{k+1}) + V_{k+1}(p_{k+1}(h_k(y_k, w_k), u_{k+1}))\big]\Big\}.$$

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 approximating 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 Section 1.6.2. 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.

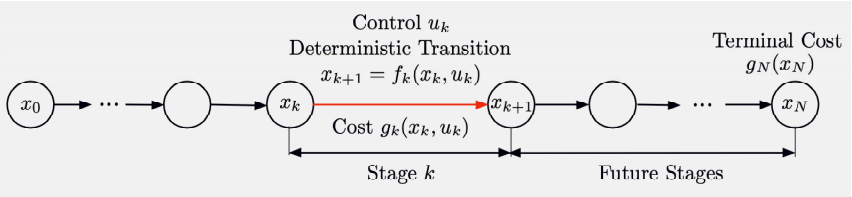


Figure 2.0.1 Illustration of a deterministic N-stage optimal control problem. Starting from state x_k, the next state under control u_k is generated nonrandomly, according to

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

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

2.1 DETERMINISTIC DISCRETE SPACES 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 2.0.1. 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.

The Exact DP Algorithm

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

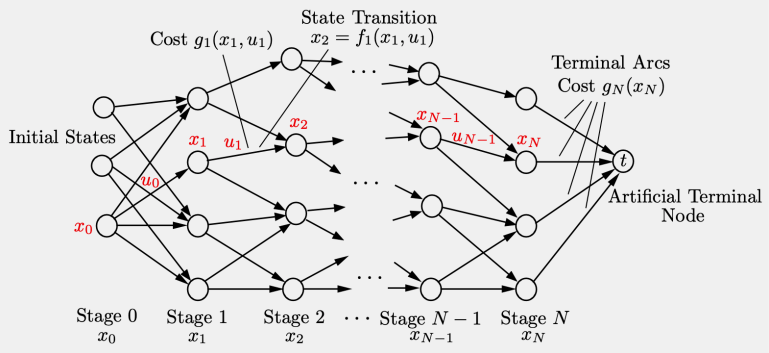


Figure 2.1.1 Illustration of a deterministic finite-state DP problem. Nodes correspond to states x_k. Arcs correspond to state-control pairs (x_k, u_k). An arc (x_k, u_k) has start and end nodes x_k and $x_{k+1} = f_k(x_k, u_k)$, respectively. The cost $g_k(x_k, u_k)$ of the transition is the length 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. 2.1.1. 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-

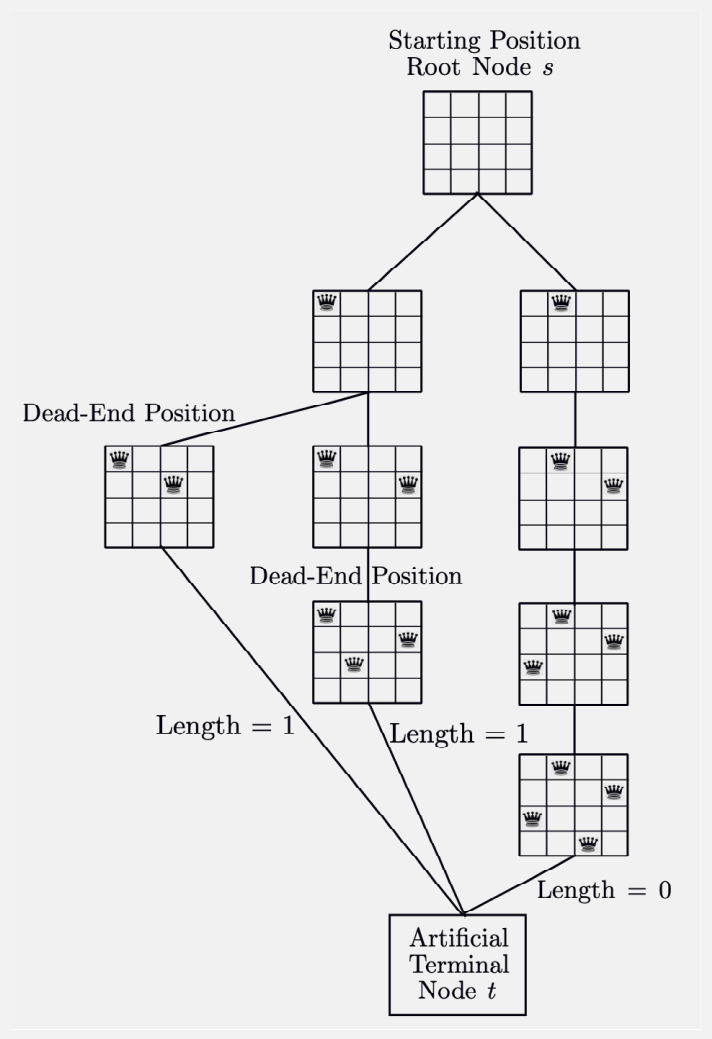


Figure 2.1.2 A finite horizon deterministic DP formulation of the four queens problem. Symmetric positions resulting from placing a queen in one of the rightmost squares in the top row have been ignored. Squares containing a queen have been darkened. All arcs have length zero except for those connecting 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. 2.1.2) 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. 2.1.1 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.

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.

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

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

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. 2.1.3. 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

$$U_0 = \big\{\tilde{u}_0 \mid \text{there exists a solution of the form } (\tilde{u}_0, \tilde{u}_1, \ldots, \tilde{u}_{N-1}) \in U\big\}. \tag{2.7}$$

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

$$\begin{aligned} U_k(u_0, \ldots, u_{k-1}) = \big\{u_k \mid \text{for some } \overline{u}_{k+1}, \ldots, \overline{u}_{N-1} \text{ we have} \\ (u_0, \ldots, u_{k-1}, u_k, \overline{u}_{k+1}, \ldots, \overline{u}_{N-1}) \in U\big\}. \end{aligned} \tag{2.8}$$

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. The last stage corresponds to the N-solutions $u = (u_0, \ldots, u_{N-1})$, and the terminal cost is $G(u)$; see Fig. 2.1.3. All other transitions in this DP problem formulation have cost 0.

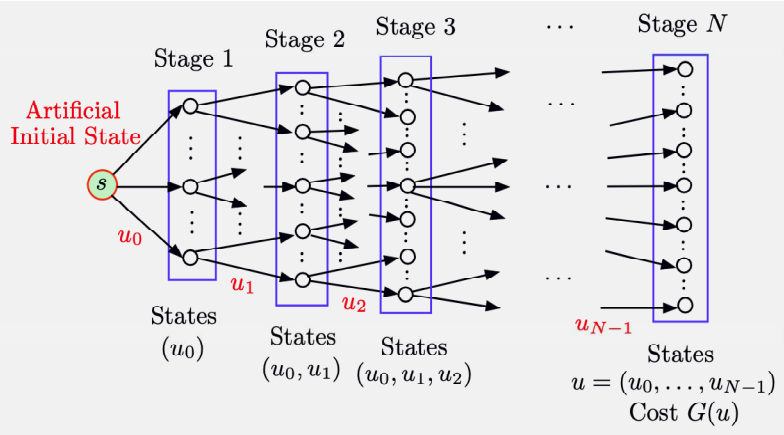


Figure 2.1.3 Formulation of a discrete optimization problem as a DP problem with N stages. There is a cost $G(u)$ only at the terminal stage on the arc connecting an N-solution $u = (u_0, \ldots, u_{N-1})$ upon reaching the terminal state. Note that there is only one incoming arc at each node.

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

$$u_k^* \in \arg \min_{u_k \in U_k(u_0^*, \ldots, u_{k-1}^*)} J_{k+1}^*(u_0^*, \ldots, u_{k-1}^*, u_k), \quad k = 0, \ldots, N-1, \quad (2.9)$$

where U_0 is given by Eq. (2.7), and U_k is given by Eq. (2.8): first compute u_0^*, then u_1^*, and so on up to u_{N-1}^*; cf. Eq. (2.6).

Of course here the number of states typically grows exponentially with N, but we can use the DP minimization (2.9) 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. (2.9). 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 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.

Constraint Programming

An interesting special case of the general discrete optimization problem $\min_{u \in U} G(u)$ is the *feasibility problem*, whereby $G(u) \equiv 0$, so the problem reduces to finding a value of u that satisfies the constraint $u \in U$. Typically, in this case the constraint set U has some structure, such as being the intersection of a finite number of constraint sets $U_1, \ldots, U_m$,

$$U = \cap_{i=1}^m U_i,$$

where each set U_i couples some of the variables $u_0, \ldots, u_{N-1}$. This type of feasibility problem is also known as a *constraint programming problem*. The four queens problem (Example 2.1.1) provides an illustration.

Constraint programming problems can of course be formulated as DP problems using our earlier formulation (cf. Fig. 2.1.3). They 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, the problem of finding a feasible solution of the system of constraints

$$h_k(u_k, u_{k+1}) \leq 0, \qquad k = 0, \ldots, N-1,$$

$$u_k \in U_k, \qquad k = 0, \ldots, N-1,$$

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.

2.2 APPROXIMATION IN VALUE SPACE

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

$$\tilde{u}_k \in \arg\min_{u_k \in U_k(\tilde{x}_k)} \Big[g_k(\tilde{x}_k, u_k) + \tilde{J}_{k+1}\big(f_k(\tilde{x}_k, u_k)\big) \Big], \tag{2.10}$$

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. (2.10)] is known as the (approximate) *Q-factor of* (x_k, u_k). Note that the computation of the suboptimal control (2.10) 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 (2.10) 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).

Multistep Lookahead

The approximation in value space algorithm (2.10) 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 optimization 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.

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. (2.10) 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-to-go $\tilde{J}_{k+1}(x_{k+1})$ in the corresponding approximation in value space scheme (2.10).

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

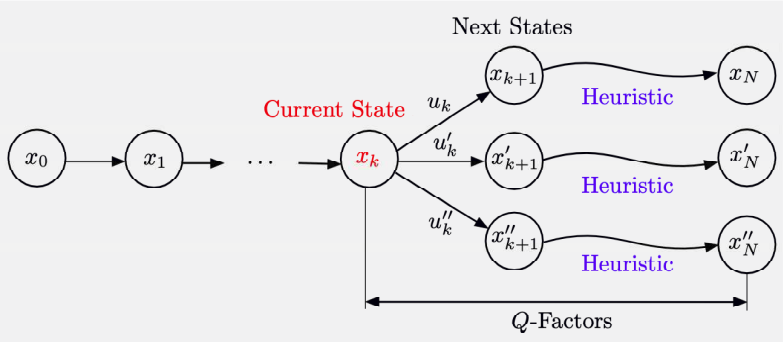


Figure 2.3.1 Schematic illustration of rollout with one-step lookahead for a deterministic problem. At state x_k, for every pair (x_k, u_k), $u_k \in U_k(x_k)$, the base heuristic generates a Q-factor

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

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

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.

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

$$\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k), \tag{2.11}$$

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

$$\tilde{Q}_k(x_k, u_k) = g_k(x_k, u_k) + H_{k+1}\big(f_k(x_k, u_k)\big); \tag{2.12}$$

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 (2.11).

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. (2.10). 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

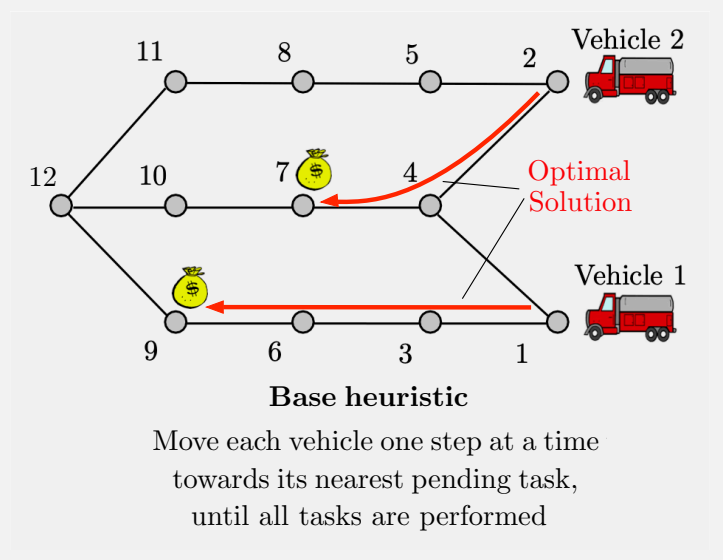


Figure 2.3.2 An instance of the vehicle routing problem of Example 2.3.1. The two vehicles aim to collectively perform the two tasks, at nodes 7 and 9, as fast as possible, by each moving to a neighboring node at each step. The optimal routes are shown.

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 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

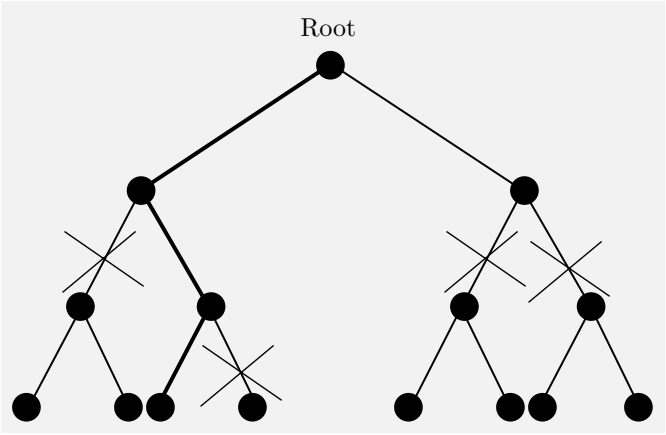


Figure 2.3.3 Binary tree for the breakthrough problem. Each arc is either free or is blocked (crossed out in the figure). The problem is to find a path from the root to one of the leaves, which is free (such as the one shown with thick lines).

per Eq. (2.10). 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), (1,7), (1,2), (1,1), perform a heuristic calculation from each of them, 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 to be discussed next.

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*

† 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.

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 $n = k+1, \ldots, N-1$.

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

$$J_{k,\tilde{\pi}}(x_k) \leq H_k(x_k), \qquad \text{for all } x_k \text{ and } k, \tag{2.13}$$

where $H_k(x_k)$ denotes the cost of the base heuristic starting from x_k.

Proof: We prove this inequality by induction. Clearly it holds for $k = N$, since

$$J_{N,\tilde{\pi}} = H_N = g_N.$$

Assume that it holds for index $k+1$. For any state x_k, let $\overline{u}_k$ be the control applied by the base heuristic at x_k. Then we have

$$\begin{aligned} J_{k,\tilde{\pi}}(x_k) &= g_k\big(x_k, \tilde{\mu}_k(x_k)\big) + J_{k+1,\tilde{\pi}}\Big(f_k\big(x_k, \tilde{\mu}_k(x_k)\big)\Big) \\ &\leq g_k\big(x_k, \tilde{\mu}_k(x_k)\big) + H_{k+1}\Big(f_k\big(x_k, \tilde{\mu}_k(x_k)\big)\Big) \\ &= \min_{u_k \in U_k(x_k)} \Big[g_k(x_k, u_k) + H_{k+1}\big(f_k(x_k, u_k)\big)\Big] \\ &\leq g_k\big(x_k, \overline{u}_k\big) + H_{k+1}\big(f_k(x_k, \overline{u}_k)\big) \\ &= H_k(x_k), \end{aligned} \tag{2.14}$$

where:

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

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

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

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

This completes the proof of the cost improvement property (2.13). **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. (2.12)], 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

$$\min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k) \leq H_k(x_k). \tag{2.15}$$

In words, the sequential improvement property (2.15) 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. (2.15). 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. (2.14), by replacing the last two steps (which rely on sequential consistency) with Eq. (2.15). **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

$$\text{Cost of } T_0 \geq \cdots \geq \text{Cost of } T_k \geq \text{Cost of } T_{k+1} \geq \cdots \geq \text{Cost of } T_N. \quad (2.16)$$

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

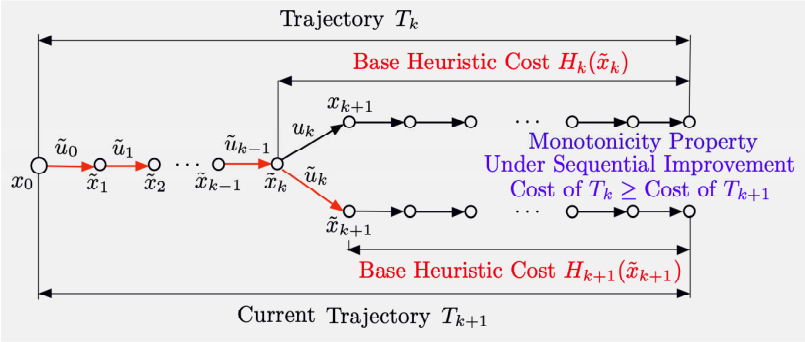


Figure 2.3.4 Proof of the monotonicity property (2.16). At $\tilde{x}_k$, the kth state generated by the rollout algorithm, we compare the "current" trajectory T_k whose cost is the sum of the cost of the current partial trajectory $(x_0, \tilde{u}_0, \tilde{x}_1, \tilde{u}_1, \ldots, \tilde{x}_k)$ and 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.

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 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

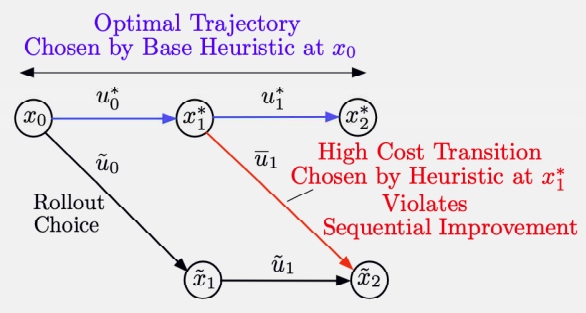


Figure 2.3.5 A 2-stage problem with states $x_1^*, \tilde{x}_1$ at stage 1, and states $x_2^*, \tilde{x}_2$ at stage 2. The controls and corresponding transitions are as shown in the figure. The rollout choice at the initial state x_0 is strictly suboptimal, while the base heuristic choice is optimal. 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.

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

$$g_0(x_0, u_0^*) + H_1(x_1^*) > g_0(x_0, \tilde{u}_0) + H_1(\tilde{x}_1), \tag{2.17}$$

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 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 (2.17) 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 (2.16) 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.

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 best end-to-end trajectory computed up to stage k of the algorithm. Initially, $\overline{T}_0$ is the trajectory generated by the base heuristic starting at the initial state x_0. The idea now is to *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

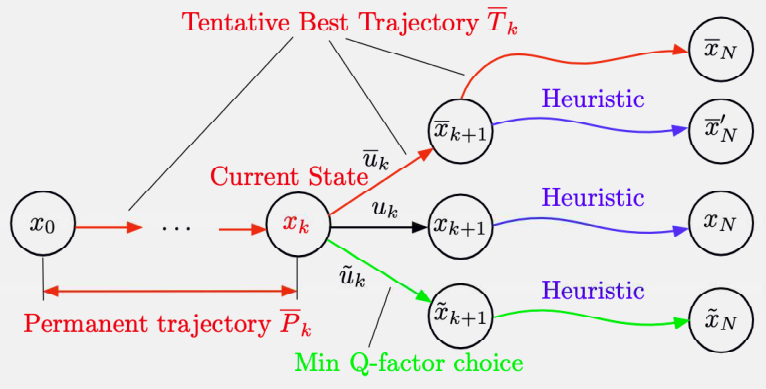


Figure 2.3.6 Schematic illustration of fortified rollout. After k steps, we have constructed the permanent trajectory

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

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: $\overline{T}_{k+1} = \overline{T}_k$.

on which of the two is smaller, chooses $\overline{u}_k$ or $\tilde{u}_k$ and moves to $\overline{x}_{k+1}$ or to $\tilde{x}_{k+1}$, respectively. In particular, if

$$C(\overline{T}_k) \leq C(\tilde{T}_k),$$

the algorithm sets the next state and corresponding tentative 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 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 sequential improvement assumption holds. In this case by maintaining 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 (2.15) 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 (2.15) 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

$$\tilde{\mu}_k(x_k)\in\arg\min_{u_k\in U_k(x_k)}\tilde{Q}_k(x_k,u_k),\tag{2.18}$$

[cf. Eqs. (2.11) and (2.12)], 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 $u_k\in\overline{U}_k(x_k)$:

$$\tilde{\mu}_k(x_k)\in\arg\min_{u_k\in\overline{U}_k(x_k)}\tilde{Q}_k(x_k,u_k).\tag{2.19}$$

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 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. (2.18)].

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:

$$\min_{u_k \in \overline{U}_k(x_k)} \tilde{Q}_k(x_k, u_k) \leq H_k(x_k). \tag{2.20}$$

This can be seen by verifying that Eq. (2.20) is sufficient to guarantee that the monotone improvement Eq. (2.16) is satisfied. The condition (2.20) 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 (2.20) 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 (2.18) by the simpler minimization (2.19) 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 (2.20), 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.

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 cost function approximation, and approximation in value space without any rollout.

2.3.6 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 solutions can be revealed by the expert, this is all that is needed.† In fact,

† 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''.

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 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 (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 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

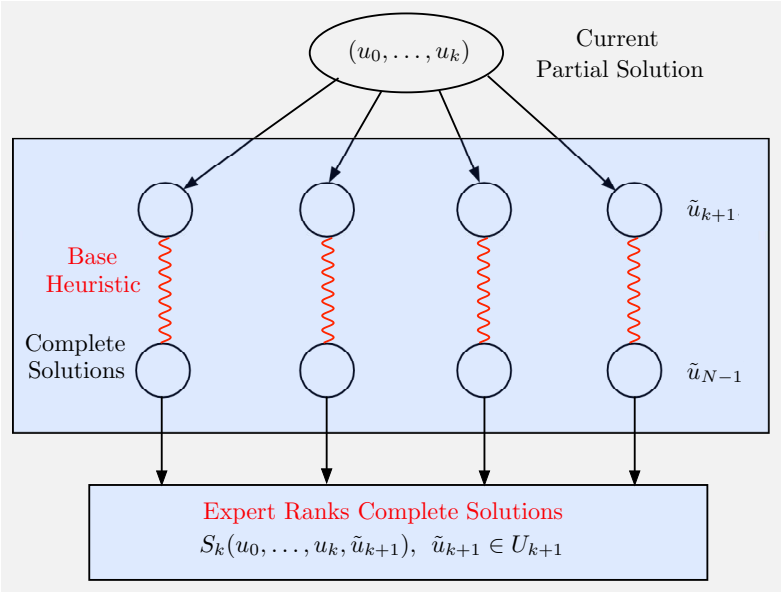


Figure 2.3.7 Schematic illustration of model-free rollout with an expert for minimizing $G(u)$ subject to

$$u \in U_0 \times \cdots \times U_{N-1}.$$

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.

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

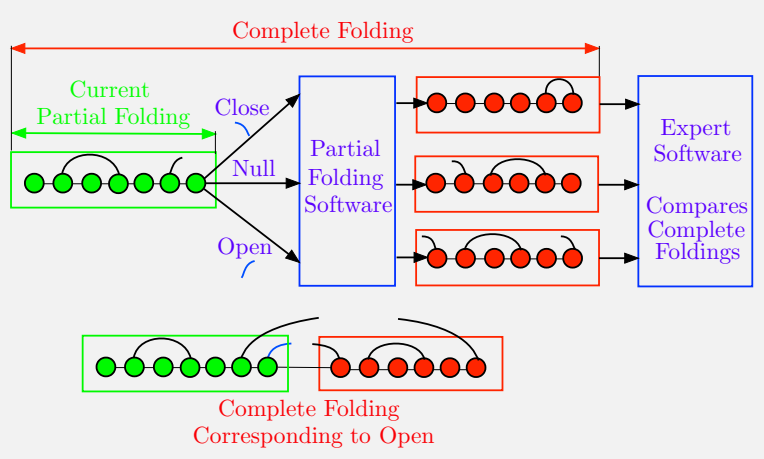


Figure 2.3.8 Schematic illustration of rollout for the RNA folding problem. The current state is the partial folding depicted on the left side. There are at most three choices for control at each state.

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

$$G(u^s) > G(\overline{u}^s), \qquad s = 1, \ldots, q, \tag{2.21}$$

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. (2.21), we generate for each $(u^s, \overline{u}^s)$, two solution-cost pairs

$$(u^s, 1), \ (\overline{u}^s, -1), \qquad s = 1, \ldots, q.$$

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

$$\tilde{Q}_k(u_0, \ldots, u_k, r_k), \qquad k = 1, \ldots, N,$$

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.

2.4 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

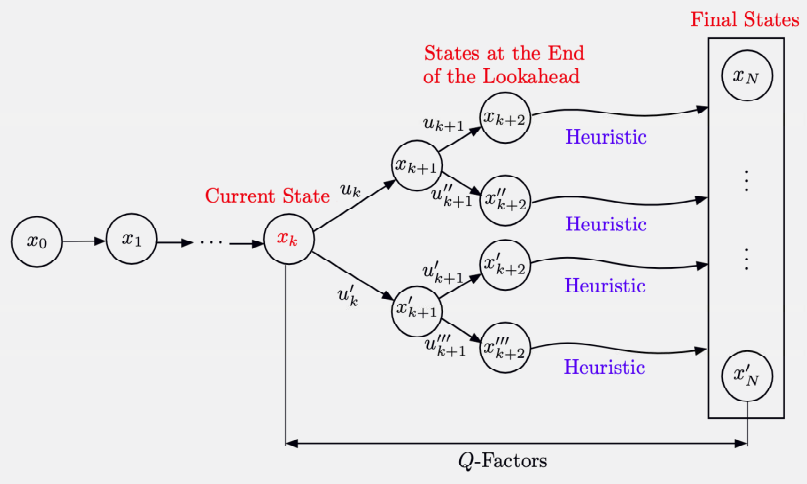


Figure 2.4.1 Illustration of multistep rollout with $\ell = 2$ for deterministic problems. We run the base heuristic from each leaf $x_{k+\ell}$ at the end of the lookahead graph. We then construct an optimal solution for the lookahead minimization problem, where the heuristic cost is used as terminal 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.

is associated with minimum cost, compute the controls $\tilde{u}_k$ and $\tilde{u}_{k+1}$ that lead from x_k to $\tilde{x}_{k+2}$, 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

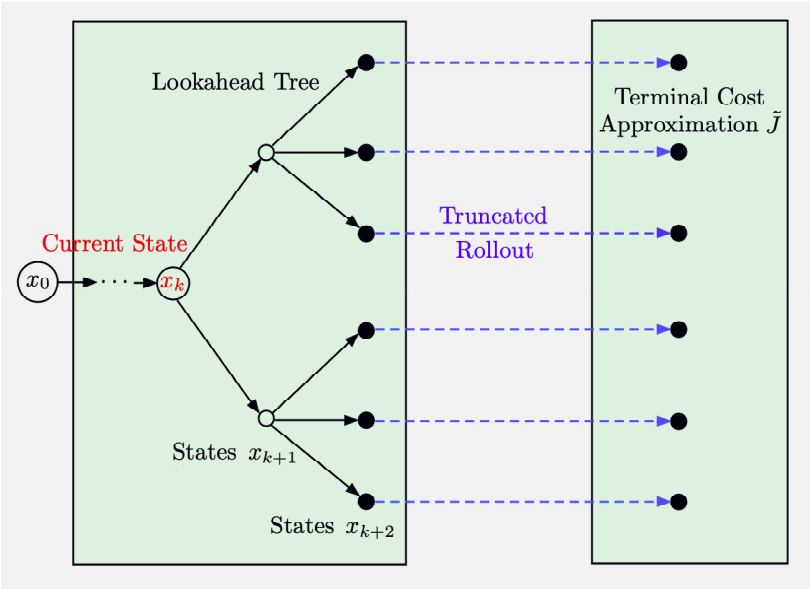


Figure 2.4.2 Illustration of truncated rollout with two-step lookahead and a terminal cost approximation $\tilde{J}$. The base heuristic is used for a limited number of steps and the terminal cost is added to compensate for the remaining steps.

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].

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 $n+1$,

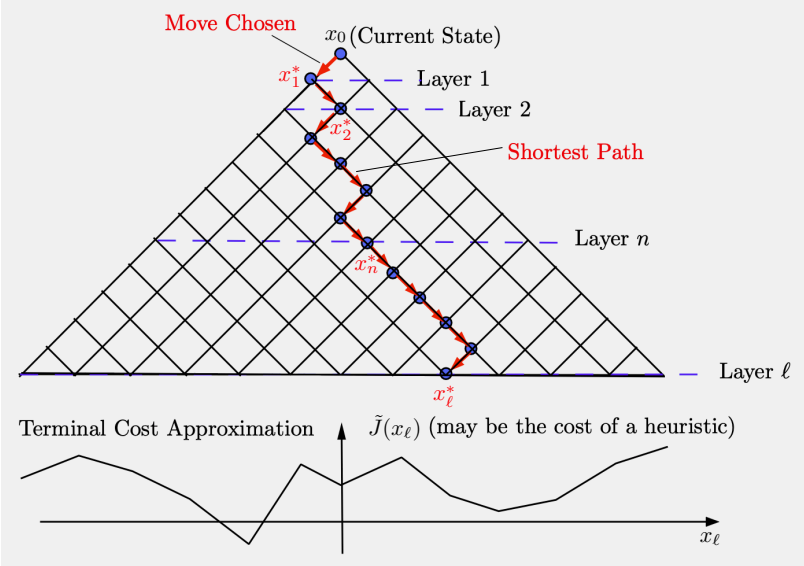


Figure 2.4.3 Illustration of the general ℓ-step approximation in value space scheme with a terminal cost approximation $\tilde{J}$ where x_0 denotes the current state. It involves an acyclic graph of ℓ layers, with layer n, $n = 1, \ldots, \ell$, consisting of all the states x_n that can be reached from 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 $x_0 \to x_1^*$.

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

$$\hat{g}_n(x_n, x_{n+1}) = \min_{\{u_n \in U_n(x_n) \mid x_{n+1} = f_n(x_n, u_n)\}} g_n(x_n, u_n). \tag{2.22}$$

For the states x_ℓ of the last layer there is also a given terminal cost approximation $\tilde{J}(x_\ell)$, possibly obtained through off-line training and/or rollout with a base policy. 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 that 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, 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 Using Truncated Rollout

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

† 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 one-step lookahead policy $\tilde{\mu}$, defined by $\tilde{J}$ using the equation†

$$\tilde{\mu}(x) \in \arg\min_{u \in U(x)} \Big[g(x,u) + \tilde{J}\big(f(x,u)\big) \Big]. \tag{2.23}$$

Let us assume that the principal computation in the minimization of Eq. (2.23) 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 appromimation 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 Tesauto and Galperin [TeG96], out of necessity, because multistep lookahead 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. Let us now calculate the corresponding number of calculations of the value of $\tilde{J}$ for scheme (b).

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 $\ell-1$ controls using the base policy (2.23) 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 conslude 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.

† For simplicity, we use stationary system notation, ommiting the time subscripts of U, g, and f.

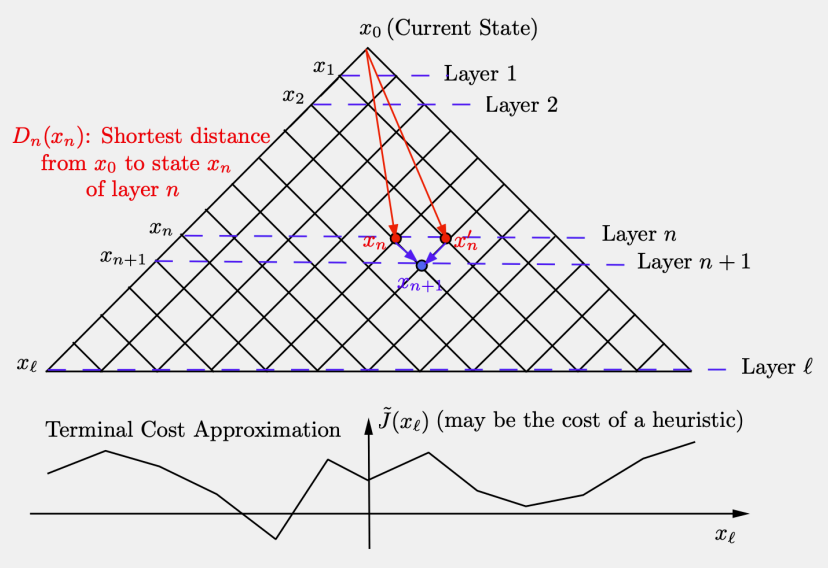


Figure 2.4.4 Illustration of the forward DP algorithm for computing the shortest distances from the current state x_0 to all the states x_n of the layers $n = 1, \ldots, \ell$. The shortest distance $D_{n+1}(x_{n+1})$ to a state x_{n+1} of layer n is obtained by minimizing over all predecessor states x_n the sum

$$\hat{g}_n(x_n, x_{n+1}) + D_n(x_n).$$

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

$$D_{n+1}(x_{n+1}) = \min_{x_n} \Big[\hat{g}_n(x_n, x_{n+1}) + D_n(x_n)\Big],$$

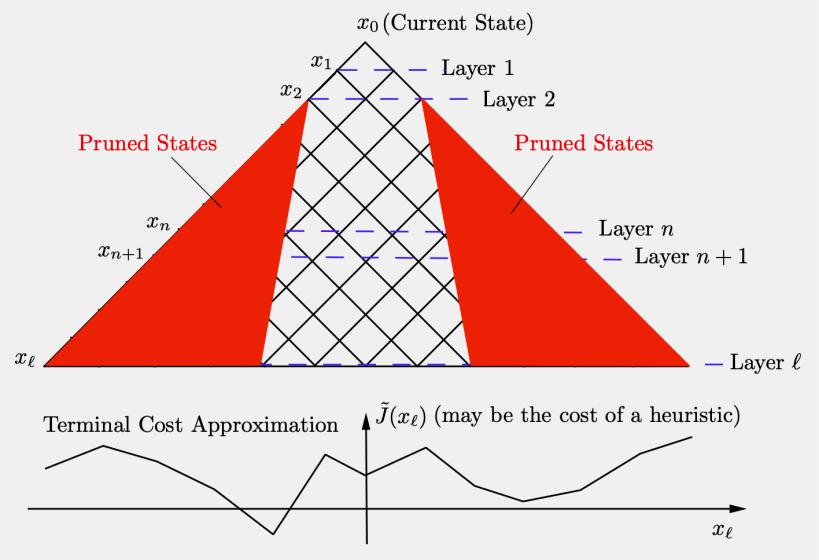


Figure 2.4.5 Illustration of iterative deepening with pruning within the context of forward DP.

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. (2.22).

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 a feasible solution to a lookahead minimization of some depth, even if it is interrupted because the limit of our computational budget has been reached. 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

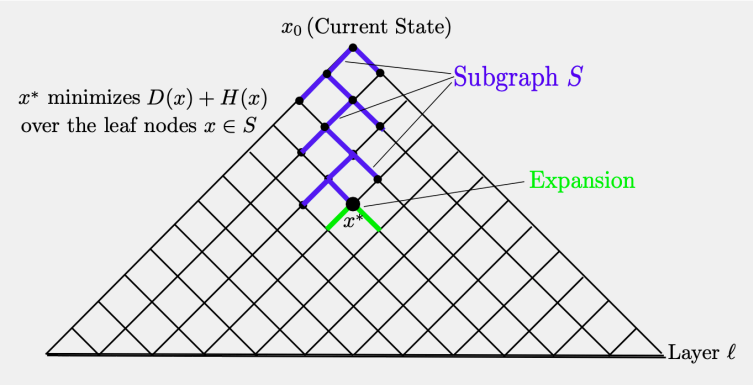


Figure 2.4.6 Illustration of the ℓ-step lookahead minimization problem and its suboptimal solution with the IMR algorithm. The algorithm maintains a connected acyclic subgraph S as shown. At each iteration it expands S by selecting a leaf node of S and by adding its neighbor nodes to S (if not already 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" $H(x)$.

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 $n+1$.

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, 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.

$H(x)$ 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 thus encourages the algorithm to "backtrack" and select nodes x^* that lie in layers closer to x_0.

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.

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

$$T = (x_0, u_0, x_1, u_1, \ldots, u_{N-1}, x_N), \tag{2.24}$$

where

$$x_{k+1} = f_k(x_k, u_k), \qquad k = 0, 1, \ldots, N-1, \tag{2.25}$$

is referred to as a *complete trajectory*. Our problem is stated succinctly as

$$\min_{T \in C} G(T), \tag{2.26}$$

where G is some cost function and C is the constraint set.

Note that G need not have the additive form

$$G(T) = g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k), \tag{2.27}$$

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. (2.25).†

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 (2.27), 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

$$g_N^m(x_N) + \sum_{k=0}^{N-1} g_k^m(x_k, u_k) \leq b^m, \qquad m = 1, \ldots, M, \tag{2.28}$$

† 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.

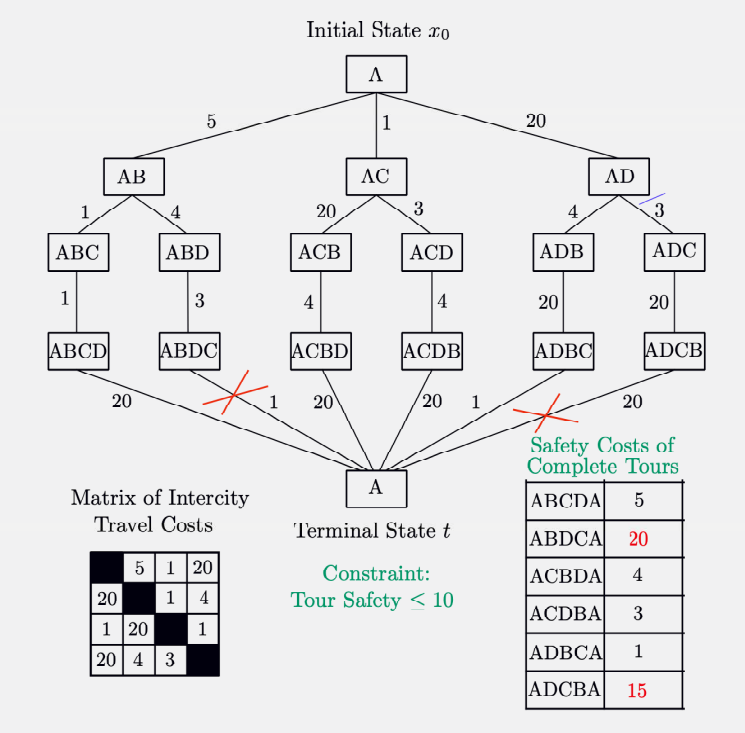


Figure 2.5.1 An example of a constrained traveling salesman problem; cf. Example 2.5.1. We want to find a minimum cost tour that has safety cost less or equal to 10. The safety costs of the six possible tours are given in the table on the right. The (unconstrained) minimum cost tour, ABDCA, does not satisfy the safety constraint. The optimal constrained tour is ABCDA.

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.

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. (2.28). 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

$$R(y_k) = (x_k, u_k, x_{k+1}, u_{k+1}, \ldots, u_{N-1}, x_N),$$

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 $T_k(y_k, u_k)$:

$$T_k(y_k, u_k) = \big(y_k, u_k, R(y_{k+1})\big). \tag{2.29}$$

This process is illustrated in Fig. 2.5.2. Note that the partial trajectory $R(y_{k+1})$ produced by the base heuristic depends on the entire partial trajectory y_{k+1}, not just the state x_{k+1}.

A complete trajectory $T_k(y_k, u_k)$ of the form (2.29) is generally feasible for only the subset $\hat{U}_k(y_k)$ of controls u_k that maintain feasibility:

$$\hat{U}_k(y_k) = \big\{u_k \mid T_k(y_k, u_k) \in C\big\}. \tag{2.30}$$

Our rollout algorithm starts from a given initial state $\tilde{y}_0 = \tilde{x}_0$, and generates successive partial trajectories $\tilde{y}_1, \ldots, \tilde{y}_N$, of the form

$$\tilde{y}_{k+1} = \big(\tilde{y}_k, \tilde{u}_k, f_k(\tilde{x}_k, \tilde{u}_k)\big), \qquad k = 0, \ldots, N-1, \tag{2.31}$$

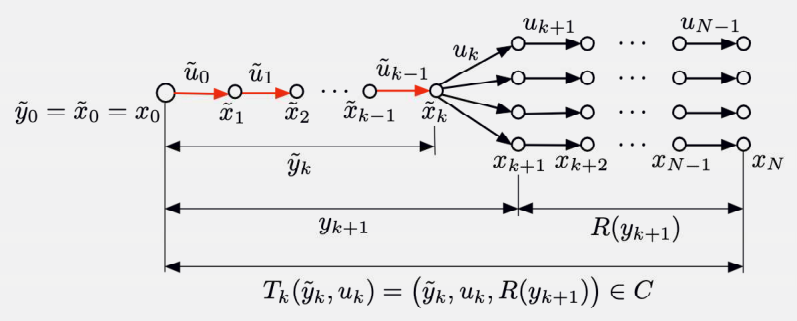


Figure 2.5.2 The trajectory generation mechanism of the rollout algorithm. At stage k, and given the current partial trajectory

$$\tilde{y}_k = (\tilde{x}_0, \tilde{u}_0, \tilde{x}_1, \ldots, \tilde{u}_{k-1}, \tilde{x}_k),$$

which starts at $\tilde{x}_0$ and ends at $\tilde{x}_k$, we consider all possible next states $x_{k+1} = f_k(\tilde{x}_k, u_k)$, run the base heuristic starting at $y_{k+1} = (\tilde{y}_k, u_k, x_{k+1})$, and form the complete trajectory $T_k(\tilde{y}_k, u_k)$. Then the rollout algorithm:

(a) Finds $\tilde{u}_k$, the control that minimizes the cost $G\big(T_k(\tilde{y}_k, u_k)\big)$ over all u_k for which the complete trajectory $T_k(\tilde{y}_k, u_k)$ is feasible.

(b) Extends $\tilde{y}_k$ by $\big(\tilde{u}_k, f_k(\tilde{x}_k, \tilde{u}_k)\big)$ to form $\tilde{y}_{k+1}$.

where $\tilde{x}_k$ is the last state component of $\tilde{y}_k$, and $\tilde{u}_k$ is a control that minimizes the heuristic cost $G\big(T_k(\tilde{y}_k, u_k)\big)$ over all u_k for which $T_k(\tilde{y}_k, u_k)$ is feasible. Thus at stage k, the algorithm forms the set $U_k(\tilde{y}_k)$ [cf. Eq. (2.30)] and selects from $U_k(\tilde{y}_k)$ a control $\tilde{u}_k$ that minimizes the cost of the complete trajectory $T_k(\tilde{y}_k, u_k)$:

$$\tilde{u}_k \in \arg \min_{u_k \in U_k(\tilde{y}_k)} G\big(T_k(\tilde{y}_k, u_k)\big); \tag{2.32}$$

see Fig. 2.5.2. The objective is to produce a feasible final complete trajectory $\tilde{y}_N$, which has a cost $G(\tilde{y}_N)$ that is no larger than the cost of $R(\tilde{y}_0)$ produced by the base heuristic starting from $\tilde{y}_0$, i.e.,

$$G(\tilde{y}_N) \leq G\big(R(\tilde{y}_0)\big).$$

Note that $T_k(\tilde{y}_k, u_k)$ is not guaranteed to be feasible for any given u_k (i.e., may not belong to C), but we will assume that the constraint set $U_k(\tilde{y}_k)$ of problem (2.32) is nonempty, so that our rollout algorithm is well-defined. We will later modify our algorithm so that it is well-defined under the weaker assumption that just *the complete trajectory generated by the base heuristic starting from the initial state $\tilde{y}_0$ is feasible*, i.e., $R(\tilde{y}_0) \in C$.

Constrained Rollout Algorithm

The algorithm starts at stage 0 and sequentially proceeds to the last stage. At the typical stage k, it has constructed a partial trajectory

$$\tilde{y}_k = (\tilde{x}_0, \tilde{u}_0, \tilde{x}_1, \ldots, \tilde{u}_{k-1}, \tilde{x}_k) \tag{2.33}$$

that starts at the given initial state $\tilde{y}_0 = \tilde{x}_0$, and is such that

$$\tilde{x}_{t+1} = f_t(\tilde{x}_t, \tilde{u}_t), \qquad t = 0, 1, \ldots, k-1.$$

The algorithm then forms the set of controls

$$U_k(\tilde{y}_k) = \big\{u_k \mid T_k(\tilde{y}_k, u_k) \in C\big\}$$

that is consistent with feasibility [cf. Eq. (2.30)], and chooses a control $\tilde{u}_k \in U_k(\tilde{y}_k)$ according to the minimization

$$\tilde{u}_k \in \arg \min_{u_k \in U_k(\tilde{y}_k)} G\big(T_k(\tilde{y}_k, u_k)\big), \tag{2.34}$$

[cf. Eq. (2.32)], where

$$T_k(\tilde{y}_k, u_k) = \Big(\tilde{y}_k, u_k, R\big(\tilde{y}_k, u_k, f_k(\tilde{x}_k, u_k)\big)\Big);$$

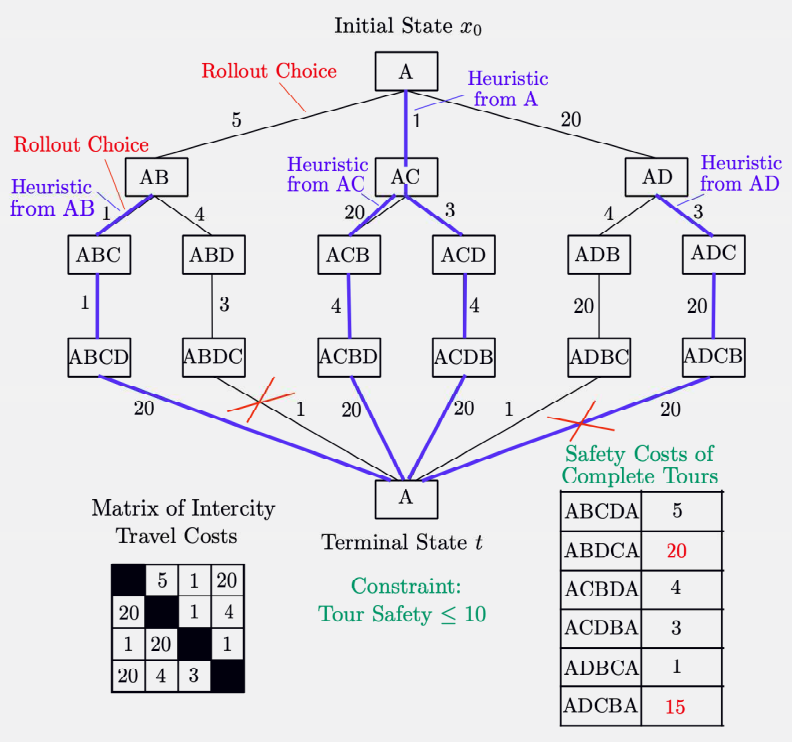
[cf. Eq. (2.29)]. Finally, the algorithm sets

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

[cf. Eq. (2.31)], thus obtaining the partial trajectory $\tilde{y}_{k+1}$ to start the next stage.

It can be seen that our constrained rollout algorithm is not much more complicated or computationally demanding than its unconstrained version where the constraint $T \in C$ is not present (as long as checking feasibility of a complete trajectory T is not computationally demanding). Note, however, that our algorithm makes essential use of the deterministic character of the problem, and does not admit a straightforward extension to stochastic problems, since checking feasibility of a complete trajectory is typically difficult in the context of these problems.

The rollout algorithm just described is illustrated in Fig. 2.5.3 for our earlier traveling salesman Example 2.5.1. Here we want to find a minimum travel cost tour that additionally satisfies a safety constraint, namely that the "safety cost" of the tour should be less than a certain threshold. Note



	5	1	20
20		1	4
1	20		1
20	4	3	

Figure 2.5.3 The constrained traveling salesman problem; cf. Example 2.5.1, and its rollout solution using the base heuristic shown, which completes a partial tour as follows:

At A it yields ACDBA.
At AB it yields ABCDA.
At AC it yields ACBDA.
At AD it yields ADCBA.

This base heuristic is not assumed to have any special structure. It is just capable of completing every partial tour without regard to any additional considerations. Thus for example the heuristic generates at A the complete tour ACDBA, and it switches to the tour ACBDA once the salesman moves to AC.

At city A, the rollout algorithm:

(a) Considers the partial tours AB, AC, and AD.

(b) Uses the base heuristic to obtain the corresponding complete tours ABCDA, ACBDA, and ADCBA.

(c) Discards ADCBA as being infeasible.

(d) Compares the other two tours, ABCDA and ACBDA, finds ABCDA to have smaller cost, and selects the partial tour AB.

(e) At AB, it considers the partial tours ABC and ABD.

(f) It uses the base heuristic to obtain the corresponding complete tours ABCDA and ABDCA, and discards ABDCA as being infeasible.

(g) It finally selects the complete tour ABCDA.

that the minimum cost tour, ABDCA, in this example does not satisfy the safety constraint. Moreover, the tour ABCDA obtained by the rollout algorithm has barely smaller cost than the tour ACDBA generated by the base heuristic starting from A. In fact if the travel cost D→A were larger, say 25, the tour produced by constrained rollout would be more costly than the one produced by the base heuristic starting from A. This points to the need for a constrained version of the notion of sequential improvement and for a fortified variant of the algorithm, which we discuss next.

Sequential Consistency, Sequential Improvement, and the Cost Improvement Property

We will now introduce sequential consistency and sequential improvement conditions guaranteeing that the control set $U_k(\tilde{y}_k)$ in the minimization (2.34) is nonempty, and that the costs of the complete trajectories $T_k(\tilde{y}_k, \tilde{u}_k)$ are improving with each k in the sense that

$$G\big(T_{k+1}(\tilde{y}_{k+1}, \tilde{u}_{k+1})\big) \leq G\big(T_k(\tilde{y}_k, \tilde{u}_k)\big), \qquad k = 0, 1, \ldots, N-1,$$

while at the first step of the algorithm we have

$$G\big(T_0(\tilde{y}_0, \tilde{u}_0)\big) \leq G\big(R(\tilde{y}_0)\big).$$

It will then follow that the cost improvement property

$$G(\tilde{y}_N) \leq G\big(R(\tilde{y}_0)\big)$$

holds.

Definition 2.5.1: We say that the base heuristic is *sequentially consistent* if whenever it generates a partial trajectory

$$(x_k, u_k, x_{k+1}, u_{k+1}, \ldots, u_{N-1}, x_N),$$

starting from a partial trajectory y_k, it also generates the partial trajectory

$$(x_{k+1}, u_{k+1}, x_{k+2}, u_{k+2}, \ldots, u_{N-1}, x_N),$$

starting from the partial trajectory $y_{k+1} = \big(y_k, u_k, x_{k+1}\big)$.

As we have noted in the context of unconstrained rollout, greedy heuristics tend to be sequentially consistent. Also any policy [a sequence of feedback control functions $\mu_k(y_k)$, $k = 0, 1, \ldots, N-1$] for the DP problem of minimizing the terminal cost $G(y_N)$ subject to the system equation

$$y_{k+1} = \big(y_k, u_k, f_k(x_k, u_k)\big)$$

and the feasibility constraint $y_N \in C$ can be seen to be sequentially consistent. For an example where sequential consistency is violated, consider the base heuristic of the traveling salesman Example 2.5.1. From Fig. 2.5.3, it can be seen that the base heuristic at A generates ACDBA, but from AC it generates ACBDA, thus violating sequential consistency.

For a given partial trajectory y_k, let us denote by $y_k \cup R(y_k)$ the complete trajectory obtained by joining y_k with the partial trajectory generated by the base heuristic starting from y_k. Thus if

$$y_k = (x_0, u_0, \ldots, u_{k-1}, x_k)$$

and

$$R(y_k) = (x_k, u_k, \ldots, u_{N-1}, x_N),$$

we have

$$y_k \cup R(y_k) = (x_0, u_0, \ldots, u_{k-1}, x_k, u_k, \ldots, u_{N-1}, x_N).$$

Definition 2.5.2: We say that the base heuristic is *sequentially improving* if for every $k = 0, 1, \ldots, N-1$ and partial trajectory y_k for which $y_k \cup R(y_k) \in C$, the set $\hat{U}_k(y_k)$ is nonempty, and we have

$$G\big(y_k \cup R(y_k)\big) \geq \min_{u_k \in \hat{U}_k(y_k)} G\big(T_k(y_k, u_k)\big). \tag{2.35}$$

Note that for a base heuristic that is not sequentially consistent, the condition $y_k \cup R(y_k) \in C$ does not imply that the set $\hat{U}_k(y_k)$ is nonempty. The reason is that starting from the next state

$$y_{k+1} = \big(y_k, u_k, f_k(x_k, u_k)\big),$$

the base heuristic may generate a different trajectory than from y_k, even if it applies u_k at y_k. Thus we need to include nonemptiness of $\hat{U}_k(y_k)$ as a requirement in the preceding definition of sequential improvement (in the fortified version of the algorithm to be discussed shortly, this requirement will be removed).

On the other hand, if the base heuristic is sequentially consistent, it is also sequentially improving. The reason is that for a sequentially consistent heuristic, $y_k \cup R(y_k)$ is equal to one of the trajectories contained in the set

$$\big\{T_k(y_k, u_k) \mid u_k \in \hat{U}_k(y_k)\big\}.$$

Our main result is contained in the following proposition.

Proposition 2.5.1: (Cost Improvement for Constrained Rollout) Assume that the base heuristic is sequentially improving and generates a feasible complete trajectory starting from the initial state $\tilde{y}_0 = \tilde{x}_0$, i.e., $R(\tilde{y}_0) \in C$. Then for each k, the set $U_k(\tilde{y}_k)$ is nonempty, and we have

$$\begin{aligned} G\big(R(\tilde{y}_0)\big) &\geq G\big(T_0(\tilde{y}_0, \tilde{u}_0)\big) \\ &\geq G\big(T_1(\tilde{y}_1, \tilde{u}_1)\big) \\ &\geq \cdots \\ &\geq G\big(T_{N-1}(\tilde{y}_{N-1}, \tilde{u}_{N-1})\big) \\ &= G(\tilde{y}_N), \end{aligned}$$

where

$$T_k(\tilde{y}_k, \tilde{u}_k) = \big(\tilde{y}_k, \tilde{u}_k, R(\tilde{y}_{k+1})\big);$$

cf. Eq. (2.29). In particular, the final trajectory $\tilde{y}_N$ generated by the constrained rollout algorithm is feasible and has no larger cost than the trajectory $R(\tilde{y}_0)$ generated by the base heuristic starting from the initial state.

Proof: Consider $R(\tilde{y}_0)$, the complete trajectory generated by the base heuristic starting from $\tilde{y}_0$. Since $\tilde{y}_0 \cup R(\tilde{y}_0) = R(\tilde{y}_0) \in C$ by assumption, it follows from the sequential improvement definition, that the set $U_0(\tilde{y}_0)$ is nonempty and we have

$$G\big(R(\tilde{y}_0)\big) \geq G\big(T_0(\tilde{y}_0, \tilde{u}_0)\big),$$

[cf. Eq. (2.35)], while $T_0(\tilde{y}_0, \tilde{u}_0) \in C$.

The preceding argument can be repeated for the next stage, by replacing $\tilde{y}_0$ with $\tilde{y}_1$, and $R(\tilde{y}_0)$ with $T_0(\tilde{y}_0, \tilde{u}_0)$. Since $\tilde{y}_1 \cup R(\tilde{y}_1) = T_0(\tilde{y}_0, \tilde{u}_0) \in C$, from the sequential improvement definition, the set $U_1(\tilde{y}_1)$ is nonempty and we have

$$G\big(T_0(\tilde{y}_0, \tilde{u}_0)\big) = G\big(\tilde{y}_1 \cup R(\tilde{y}_1)\big) \geq G\big(T_1(\tilde{y}_1, \tilde{u}_1)\big),$$

[cf. Eq. (2.35)], while $T_1(\tilde{y}_1, \tilde{u}_1) \in C$. Similarly, the argument can be successively repeated for every k, to verify that $U_k(\tilde{y}_k)$ is nonempty and that $G\big(T_k(\tilde{y}_k, \tilde{u}_k)\big) \geq G\big(T_{k+1}(\tilde{y}_{k+1}, \tilde{u}_{k+1})\big)$ for all k. **Q.E.D.**

Proposition 2.5.1 establishes the fundamental cost improvement property for constrained rollout under the sequential improvement condition. On the other hand we may construct examples where the sequential improvement condition (2.35) is violated and the cost of the solution produced by rollout is larger than the cost of the solution produced by the

base heuristic starting from the initial state (cf. the unconstrained rollout Example 2.3.3).

In the case of the traveling salesman Example 2.5.1, it can be verified that the base heuristic specified in Fig. 2.5.3 is sequentially improving. However, if the travel cost D→A were larger, say 25, then it can be verified that the definition of sequential improvement would be violated at A, and the tour produced by constrained rollout would be more costly than the one produced by the base heuristic starting from A.

The Fortified Rollout Algorithm and Other Variations

We will now discuss some variations and extensions of the constrained rollout algorithm. Let us first consider the case where the sequential improvement assumption is not satisfied. Then it may happen that given the current partial trajectory $\tilde{y}_k$, the set of controls $U_k(\tilde{y}_k)$ that corresponds to feasible trajectories $T_k(\tilde{y}_k, u_k)$ [cf. Eq. (2.30)] is empty, in which case the rollout algorithm cannot extend the partial trajectory $\tilde{y}_k$ further. To bypass this difficulty, we introduce a *fortified constrained rollout algorithm*, patterned after the fortified algorithm given earlier. For validity of this algorithm, *we require that the base heuristic generates a feasible complete trajectory $R(\tilde{y}_0)$ starting from the initial state $\tilde{y}_0$.*

The fortified constrained rollout algorithm, in addition to the current partial trajectory

$$\tilde{y}_k = (\tilde{x}_0, \tilde{u}_0, \tilde{x}_1, \ldots, \tilde{u}_{k-1}, \tilde{x}_k),$$

maintains a complete trajectory $\hat{T}_k$, called *tentative best trajectory*, which is feasible (i.e., $\hat{T}_k \in C$) and agrees with $\tilde{y}_k$ up to state $\tilde{x}_k$, i.e., $\hat{T}_k$ has the form

$$\hat{T}_k = (\tilde{x}_0, \tilde{u}_0, \tilde{x}_1, \ldots, \tilde{u}_{k-1}, \tilde{x}_k, \overline{u}_k, \overline{x}_{k+1}, \ldots, \overline{u}_{N-1}, \overline{x}_N), \tag{2.36}$$

for some $\overline{u}_k, \overline{x}_{k+1}, \ldots, \overline{u}_{N-1}, \overline{x}_N$ such that

$$\overline{x}_{k+1} = f_k(\tilde{x}_k, \overline{u}_k), \qquad \overline{x}_{t+1} = f_t(\overline{x}_t, \overline{u}_t), \quad t = k+1, \ldots, N-1.$$

Initially, $\hat{T}_0$ is the complete trajectory $R(\tilde{y}_0)$, generated by the base heuristic starting from $\tilde{y}_0$, which is assumed to be feasible. At stage k, the algorithm forms the subset $\hat{U}_k(\tilde{y}_k)$ of controls $u_k \in U_k(\tilde{y}_k)$ such that the corresponding $T_k(\tilde{y}_k, u_k)$ is not only feasible, but also has cost that is no larger than the one of the current tentative best trajectory:

$$\hat{U}_k(\tilde{y}_k) = \Big\{ u_k \in U_k(\tilde{y}_k) \mid G\big(T_k(\tilde{y}_k, u_k)\big) \leq G(\hat{T}_k) \Big\}.$$

There are two cases to consider at state k:

(1) *The set $\hat{U}_k(\tilde{y}_k)$ is nonempty.* Then the algorithm forms the partial trajectory $\tilde{y}_{k+1} = (\tilde{y}_k, \tilde{u}_k, \tilde{x}_{k+1})$, where

$$\tilde{u}_k \in \arg \min_{u_k \in \hat{U}_k(\tilde{y}_k)} G\big(T_k(\tilde{y}_k, u_k)\big), \qquad \tilde{x}_{k+1} = f_k(\tilde{x}_k, \tilde{u}_k),$$

and sets $T_k(\tilde{y}_k, \tilde{u}_k)$ as the new tentative best trajectory, i.e.,

$$\hat{T}_{k+1} = T_k(\tilde{y}_k, \tilde{u}_k).$$

(2) *The set $\hat{U}_k(\tilde{y}_k)$ is empty.* Then, the algorithm forms the partial trajectory $\tilde{y}_{k+1} = \big(\tilde{y}_k, \tilde{u}_k, \tilde{x}_{k+1}\big)$, where

$$\tilde{u}_k = \overline{u}_k, \qquad \tilde{x}_{k+1} = \overline{x}_{k+1},$$

and $\overline{u}_k, \overline{x}_{k+1}$ are the control and state subsequent to $\tilde{x}_k$ in the current tentative best trajectory $\hat{T}_k$ [cf. Eq. (2.36)], and leaves $\hat{T}_k$ unchanged, i.e.,

$$\hat{T}_{k+1} = \hat{T}_k.$$

It can be seen that the fortified constrained rollout algorithm will follow the initial complete trajectory $\hat{T}_0$, the one generated by the base heuristic starting from $\tilde{y}_0$, up to a stage k where it will discover a new feasible complete trajectory with smaller cost to replace $\hat{T}_0$ as the tentative best trajectory. Similarly, the new tentative best trajectory $\hat{T}_k$ may be subsequently replaced by another feasible trajectory with smaller cost, etc.

Note that if the base heuristic is sequentially improving, and the fortified rollout algorithm will generate the same complete trajectory as the (nonfortified) rollout algorithm given earlier, with the tentative best trajectory $\hat{T}_{k+1}$ being equal to the complete trajectory $T_k(\tilde{y}_k, \tilde{u}_k)$ for all k. The reason is that if the base heuristic is sequentially improving, the controls $\tilde{u}_k$ generated by the nonfortified algorithm belong to the set $\hat{U}_k(\tilde{y}_k)$ [by Prop. 2.5.1, case (1) above will hold].

However, it can be verified that even when the base heuristic is not sequentially improving, the fortified rollout algorithm will generate a complete trajectory that is feasible and has cost that is no worse than the cost of the complete trajectory generated by the base heuristic starting from $\tilde{y}_0$. This is because each tentative best trajectory has a cost that is no worse than the one of its predecessor, and the initial tentative best trajectory is just the trajectory generated by the base heuristic starting from the initial condition $\tilde{y}_0$.

Tree-Based Constrained Rollout Algorithms

It is possible to improve the performance of the rollout algorithm at the expense of maintaining more than one partial trajectory. In particular,

instead of the partial trajectory $\tilde{y}_k$ of Eq. (2.33), we can maintain a *tree* of partial trajectories that is rooted at $\tilde{y}_0$. These trajectories need not have equal length, i.e., they need not involve the same number of stages. At each step of the algorithm, we select a single partial trajectory from this tree, and execute the rollout algorithm's step as if this partial trajectory were the only one. Let this partial trajectory have k stages and denote it by $\tilde{y}_k$. Then we extend $\tilde{y}_k$ similar to our earlier rollout algorithm, with possibly multiple feasible trajectories. There is also a fortified version of this algorithm where a tentative best trajectory is maintained, which is the minimum cost complete trajectory generated thus far.

The aim of the tree-based algorithm is to obtain improved performance, essentially because it can go back and extend partial trajectories that were generated and temporarily abandoned at previous stages. The net result is a more flexible algorithm that is capable of examining more alternative trajectories. Note also that there is considerable freedom to select the number of partial trajectories maintained in the tree.

We finally mention a drawback of the tree-based algorithm: it is suitable for off-line computation, but it cannot be applied in an on-line context, where the rollout control selection is made after the current state becomes known as the system evolves in real-time .

2.5.1 Constrained Rollout for Discrete Optimization and Integer Programming

As noted in Section 2.1, general discrete optimization problems may be formulated as DP problems, which in turn can be addressed with rollout. The following is an example of a classical problem that involves both discrete and continuous variables. It can also be viewed as an instance of a 0-1 integer programming problem, and in fact this is the way it is usually addressed in the literature; see e.g., the book [DrH01]. The author's rollout book [Ber20a] contains additional examples.

Example 2.5.2 (Facility Location)

We are given a candidate set of N locations, and we want to place in some of these locations a "facility" that will serve the needs of a total of M "clients." Each client $i = 1, \ldots, M$ has a demand d_i for services that may be satisfied at a location $k = 0, \ldots, N-1$ at a cost a_{ik} per unit. If a facility is placed at location k, it has capacity to serve demand up to a known level c_k.

We introduce a 0-1 integer variable u_k to indicate with $u_k = 1$ that a facility is placed at location k at a cost b_k and with $u_k = 0$ that a facility is not placed at location k. Thus if y_{ik} denotes the amount of demand of client i to be served at facility k, the constraints are

$$\sum_{k=0}^{N-1} y_{ik} = d_i, \qquad i = 1, \ldots, M, \tag{2.37}$$

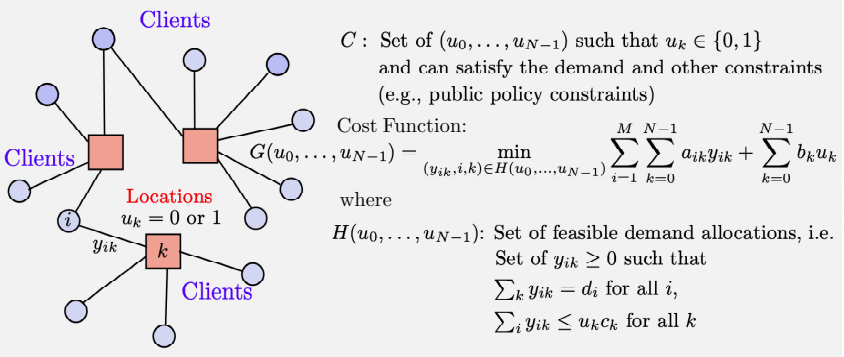


Figure 2.5.4 Schematic illustration of the facility location problem; cf. Example 2.5.2. Clients are matched to facilities, and the locations of the facilities are subject to optimization.

$$\sum_{i=1}^{M} y_{ik} \leq c_k u_k, \qquad k = 0, \ldots, N-1, \tag{2.38}$$

together with

$$y_{ik} \geq 0, \quad u_k \in \{0, 1\}, \qquad i = 1, \ldots, M, \ k = 0, \ldots, N-1. \tag{2.39}$$

We wish to minimize the cost

$$\sum_{i=1}^{M} \sum_{k=0}^{N-1} a_{ik} y_{ik} + \sum_{k=0}^{N-1} b_k u_k \tag{2.40}$$

subject to the preceding constraints; see Fig. 2.5.4. The essence of the problem is to place enough facilities at favorable locations to satisfy the clients' demand at minimum cost. This can be a very difficult mixed integer programming problem.

On the other hand, when all the variables u_k are fixed at some 0 or 1 values, the problem belongs to the class of *linear transportation* problems (see e.g., [Ber98]), and can be solved by fast polynomial algorithms. Thus the essential difficulty of the problem is how to select the integer variables u_k, $k = 0, \ldots, N-1$. This can be viewed as a discrete optimization problem of the type shown in Fig. 2.1.3. In terms of the notation of this figure, the control components are $u_0, \ldots, u_{N-1}$, where u_k can take the values 0 or 1.

To address the problem suboptimally by rollout, we must define a base heuristic at a "state" $(u_0, \ldots, u_k)$, where $u_j = 1$ or $u_j = 0$ specifies that a facility is or is not placed at location j, respectively. A suitable base heuristic at that state is to place a facility at all of the remaining locations (i.e., $u_j = 1$ for $j = k+1, \ldots, N-1$), and its cost is obtained by solving the corresponding linear transportation problem of minimizing the cost (2.40) subject to the constraints (2.37)-(2.39), with the variables u_j, $j = 0, \ldots, k$, fixed at the previously chosen values, and the variables u_j, $j = k+1, \ldots, N$, fixed at 1.

To illustrate, at the initial state where no placement decision has been made, we set $u_0 = 1$ (a facility is placed at location 0) or $u_0 = 0$ (a facility is not placed at location 0), we solve the two corresponding transportation problems, and we fix u_0, depending on which of the two resulting costs is smallest. Having fixed the status of location 0, we repeat with location 1: set the variable u_1 to 1 and to 0, solve the corresponding two transportation problems, and fix u_1, depending on which of the two resulting costs is smallest, etc.

It is easily seen that if the initial base heuristic choice (placing a facility at every candidate location) is feasible, i.e.,

$$\sum_{i=1}^{M} d_i \leq \sum_{k=0}^{N-1} c_k,$$

the rollout algorithm will yield a feasible solution with cost that is no larger than the cost corresponding to the initial application of the base heuristic. In fact it can be verified that the base heuristic here is sequentially consistent, so it is not necessary to use the fortified version of the algorithm. Regarding computational costs, the number of transportation problems to be solved is at first count $2N$, but it can be reduced to $N + 1$ by exploiting the fact that one of the two transportation problems at each stage after the first has been solved at an earlier stage.

It is worth noting, for readers that are familiar with the integer programming method of branch-and-bound, that the graph of Fig. 2.1.3 corresponds to the branch-and-bound tree for the problem, so the rollout algorithm amounts to a quick (and imperfect) method to traverse the branch-and-bound tree. This observation may be useful if we wish to use integer programming techniques to add improvements to the rollout algorithm.

We finally note that the rollout algorithm requires the solution of many linear transportation problems, which are defined by fairly similar data. It is thus important to use an algorithm that is capable of using effectively the final solution of one transportation problem as a starting point for the solution of the next. The auction algorithm for transportation problems (Bertsekas and Castañon [BeC89]) is particularly well-suited for this purpose.

Example 2.5.3 (Constrained Shortest Paths and Directed Spanning Trees)

Let us consider a spanning tree-type problem involving a directed graph with nodes $0, 1, \ldots, N$. At each node $k \in \{0, \ldots, N-1\}$ there is a set of outgoing arcs $u_k \in U_k$. Node N is special: it is viewed as a "root" node and has no outgoing arc. We are interested in collections of arcs involving a single outgoing arc per node,

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

with $u_k \in U_k$, $k = 0, \ldots, N-1$. We require that these arcs do not form a cycle, so that u specifies a directed spanning tree that is rooted at node

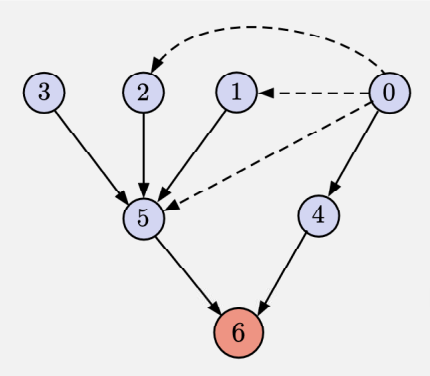


Figure 2.5.5 Schematic illustration of a constrained shortest path problem with root node $N = 6$. Given the current feasible spanning tree solution (indicated with solid line arcs), the rollout algorithm, considers a node k (in the figure $k = 0$) and the spanning tree arcs $\{u_i \mid i \neq k\}$ that are outgoing from the nodes $i \neq k$. It then considers the spanning trees that correspond to the outgoing arcs u_k from k that do not close a cycle with the set $\{u_i \mid i \neq k\}$ and are feasible [in the figure, these are the arcs indicated with broken lines, plus the arc (0,4)], and selects the arc that forms a spanning tree solution of minimum cost.

N. Note that for every node k, such a spanning tree specifies a unique path that starts at k, lies on the spanning tree, and ends at node N. We wish to find u that minimizes a given cost function $G(u)$ subject to certain additional constraints, which we do not specify further. The set of all constraints on u (including the constraint that the arcs form a directed spanning tree) is denoted abstractly as $u \in U$, so the problem comes within our constrained optimization framework of this section.

Note that this problem contains as a special case the classical shortest path problem, where we have a length for every arc and the objective is to find a tree of shortest paths to node N from all the nodes $0, \ldots, N-1$. Here U is just the constraint that the set of arcs $u = (u_0, \ldots, u_{N-1})$ form a directed spanning tree that is rooted at node N, and $G(u)$ is the sum of the lengths of all the paths specified by u, summed over all the start nodes $k = 0, \ldots, N-1$. Other shortest path-type problems, involving constraints, are included as special cases. For example, there may be a constraint that all the paths to N that are specified by the spanning tree corresponding to u contain a number of arcs that does not exceed a given upper bound.

Suppose that we have an initial solution/directed spanning tree

$$\bar{u} = (\bar{u}_0, \ldots, \bar{u}_{N-1}),$$

which is feasible (note here that finding such an initial solution may be a challenge). Let us apply the constrained rollout algorithm with a base heuristic that operates as follows: given a partial trajectory

$$y_k = (u_0, \ldots, u_{k-1}),$$

i.e., a sequence of k arcs, each outgoing from one of the nodes $0, \ldots, k-1$, it generates the complete trajectory/directed spanning tree

$$(u_0, \ldots, u_{k-1}, \bar{u}_k, \ldots, \bar{u}_{N-1}).$$

Thus the rollout algorithm, given a partial trajectory

$$\tilde{y}_k = (\tilde{u}_0, \ldots, \tilde{u}_{k-1}),$$

considers the set $\hat{U}_k(\tilde{y}_k)$ of all outgoing arcs u_k from node k, such that the complete trajectory

$$(\tilde{y}_k, u_k, \bar{u}_{k+1}, \ldots, \bar{u}_{N-1})$$

is feasible. It then selects the arc $u_k \in \hat{U}_k(\tilde{y}_k)$ that minimizes the cost

$$G(\tilde{y}_k, u_k, \bar{u}_{k+1}, \ldots, \bar{u}_{N-1});$$

see Fig. 2.5.5. It can be seen by induction, starting from $\bar{u}$, that the set of arcs $\hat{U}_k(\tilde{y}_k)$ is nonempty, and that the algorithm generates a sequence of feasible solutions/directed spanning trees, each with cost no worse than the preceding one.

Note that throughout the rollout process, a rooted spanning tree is maintained, and at each stage k, a single arc $\bar{u}_k$ that is outgoing from node k is replaced by the outgoing arc $\tilde{u}_k$. Thus two successive rooted spanning trees generated by the algorithm, differ by at most a single arc.

An interesting aspect of this rollout algorithm is that it can be applied multiple times with the final solution of one rollout application used to specify the base heuristic of the next rollout application. Moreover, a different order of nodes may be used in each rollout application. This can be viewed as a form of policy iteration, of the type that we have discussed. The algorithm will eventually terminate, in the sense that it can make no further progress. More irregular/heuristic orders of node selections are also possible; for example some nodes may be selected multiple times before others will be selected for the first time. However, there is no guarantee that the final solution thus obtained will be optimal.

2.6 SMALL STAGE COSTS AND LONG HORIZON - CONTINUOUS-TIME ROLLOUT

Let us consider the deterministic one-step approximation in value space scheme

$$\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \Big[g_k(x_k, u_k) + \tilde{J}_{k+1}\big(f_k(x_k, u_k)\big) \Big]. \tag{2.41}$$

In the context of rollout, $\tilde{J}_{k+1}\big(f_k(x_k, u_k)\big)$ is either the cost of the trajectory generated by the base heuristic starting from the next state $f_k(x_k, u_k)$, or

some approximation that may involve truncation and terminal cost function approximation, as in the truncated rollout scheme of Section 2.3.5.

There is a special difficulty within this context, which is often encountered in practice. It arises when the cost per stage $g_k(x_k, u_k)$ is either 0 or is small relative to the cost-to-go approximation $\tilde{J}_{k+1}\big(f_k(x_k, u_k)\big)$. Then there is a potential pitfall to contend with: *the cost approximation errors that are inherent in the term* $\tilde{J}_{k+1}\big(f_k(x_k, u_k)\big)$ *may overwhelm the first stage cost term* $g_k(x_k, u_k)$, with unpredictable consequences for the quality of the one-step-lookahead policy $\tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$. We will discuss this difficulty by first considering a discrete-time problem arising from discretization of a continuous-time optimal control problem.

Continuous-Time Optimal Control and Approximation in Value Space

Consider a problem that involves a vector differential equation of the form

$$\dot{x}(t) = h\big(x(t), u(t), t\big), \qquad 0 \leq t \leq T, \tag{2.42}$$

where $x(t) \in \Re^n$ is the state vector at time t, $\dot{x}(t) \in \Re^n$ is the vector of first order time derivatives of the state at time t, $u(t) \in U \subset \Re^m$ is the control vector at time t, where U is the control constraint set, and T is a given terminal time. Starting from a given initial state $x(0)$, we want to find a feasible control trajectory $\big\{u(t) \mid t \in [0, T]\big\}$, which together with its corresponding state trajectory $\big\{x(t) \mid t \in [0, T]\big\}$, minimizes a cost function of the form

$$G\big(x(T)\big) + \int_0^T g\big(x(t), u(t), t\big)dt, \tag{2.43}$$

where g represents cost per unit time, and G is a terminal cost function. This is a classical problem with a long history.

Let us consider a simple conversion of the preceding continuous-time problem to a discrete-time problem, while treading lightly over some of the associated mathematical fine points. We introduce a small discretization increment $\delta > 0$, such that $T = \delta N$ where N is a large integer, and we replace the differential equation (2.42) by

$$x_{k+1} = x_k + \delta \cdot h_k(x_k, u_k), \qquad k = 0, \ldots, N-1.$$

Here the function h_k is given by

$$h_k(x_k, u_k) = h\big(x(k\delta), u(k\delta), k\delta\big),$$

where we view $\{x_k \mid k = 0, \ldots, N-1\}$ and $\{u_k \mid k = 0, \ldots, N-1\}$ as state and control trajectories, respectively, which approximate the corresponding continuous-time trajectories:

$$x_k \approx x(k\delta), \qquad u_k \approx u(k\delta).$$

We also replace the cost function (2.43) by

$$g_N(x_N) + \sum_{k=0}^{N-1} \delta \cdot g_k(x_k, u_k),$$

where

$$g_N(x_N) = G\big(x(N\delta)\big), \qquad g_k(x_k, u_k) = g\big(x(k\delta), u(k\delta), k\delta\big).$$

Thus the approximation in value space scheme with time discretization takes the form

$$\tilde{\mu}_k(x_k) \in \arg\min_{u_k \in U} \Big[\delta \cdot g_k(x_k, u_k) + \tilde{J}_{k+1}\big(x_k + \delta \cdot h_k(x_k, u_k)\big)\Big]; \tag{2.44}$$

where $\tilde{J}_{k+1}$ is the function that approximates the cost-to-go starting from a state at time $k+1$. We note here that the ratio of the terms $\delta \cdot g_k(x_k, u_k)$ and $\tilde{J}_{k+1}\big(x_k + \delta \cdot h_k(x_k, u_k)\big)$ is likely to tend to 0 as $\delta \to 0$, since $\tilde{J}_{k+1}\big(x_k + \delta \cdot h_k(x_k, u_k)\big)$ ordinarily stays roughly constant at a nonzero level as $\delta \to 0$. This suggests that the one-step lookahead minimization may be degraded substantially by discretization, and other errors, including rollout truncation and terminal cost approximation. Note that a similar sensitivity to errors may occur in other discrete-time models that involve frequent selection of decisions, with cost per stage that is very small relative to the cumulative cost over many stages and/or the terminal cost.

To deal with this difficulty, we subtract the constant $\tilde{J}_k(x_k)$ in the one-step-lookahead minimization (2.44), and write

$$\tilde{\mu}_k(x_k) \in \arg\min_{u_k \in U} \Big[\delta \cdot g_k(x_k, u_k) + \Big(\tilde{J}_{k+1}\big(x_k + \delta \cdot h_k(x_k, u_k)\big) - \tilde{J}_k(x_k)\Big)\Big]; \tag{2.45}$$

since $\tilde{J}_k(x_k)$ does not depend on u_k, the results of the minimization are not affected. Assuming $\tilde{J}_k$ is differentiable with respect to its argument, we can write

$$\tilde{J}_{k+1}\big(x_k + \delta \cdot h_k(x_k, u_k)\big) - \tilde{J}_k(x_k) \approx \delta \cdot \nabla_x \tilde{J}_k(x_k)' h_k(x_k, u_k),$$

where $\nabla_x \tilde{J}_k$ denotes the gradient of J_k (a column vector), and prime denotes transposition. By dividing with δ, and taking informally the limit as $\delta \to 0$, we can write the one-step lookahead minimization (2.45) as

$$\tilde{\mu}(t) \in \arg\min_{u(t) \in U} \Big[g\big(x(t), u(t), t\big) + \nabla_x \tilde{J}_t\big(x(t)\big)' h\big(x(t), u(t), t\big)\Big], \tag{2.46}$$

where $\tilde{J}_t(x)$ is the continuous-time cost function approximation and $\nabla_x \tilde{J}_t(x)$ is its gradient with respect to x. This is the correct analog of the approximation in value space scheme (2.41) for continuous-time problems.

Rollout for Continuous-Time Optimal Control

In view of the value approximation scheme of Eq. (2.46), it is natural to speculate that the continuous-time analog of rollout with a base policy of the form

$$\pi = \Big\{\mu_t\big(x(t)\big) \mid 0 \leq t \leq T\Big\}, \tag{2.47}$$

where $\mu_t\big(x(t)\big) \in U$ for all $x(t)$ and t, has the form

$$\tilde{\mu}_t\big(x(t)\big) \in \arg\min_{u(t)\in U}\Big[g\big(x(t),u(t),t\big) + \nabla_x J_{\pi,t}\big(x(t)\big)'h\big(x(t),u(t),t\big)\Big]. \tag{2.48}$$

Here $J_{\pi,t}\big(x(t)\big)$ is the cost of the base policy π starting from state $x(t)$ at time t, and satisfies the terminal condition

$$J_{\pi,T}\big(x(T)\big) = G\big(x(T)\big).$$

Computationally, the inner product in the right-hand side of the above minimization can be approximated using the finite difference formula

$$\nabla_x J_{\pi,t}\big(x(t)\big)'h\big(x(t),u(t),t\big) \approx \frac{J_{\pi,t}\Big(x(t)+\delta\cdot h\big(x(t),u(t),t\big)\Big) - J_{\pi,t}\big(x(t)\big)}{\delta},$$

which can be calculated by running the base policy π starting from $x(t)$ and from $x(t)+\delta\cdot h\big(x(t),u(t),t\big)$. (This finite differencing operation may involve tricky computational issues, but we will not get into this.)

An important question is how to select the base policy π. A choice that is often sensible and convenient is to choose π to be a "short-sighted" policy, which takes into account the "short term" cost from the current state (say for a very small horizon starting from the current time t), but ignores the remaining cost. An extreme case is the *myopic* policy, given by

$$\mu_t\big(x(t)\big) \in \arg\min_{u\in U} g\big(x(t),u(t),t\big).$$

This policy is the continuous-time analog of the greedy policy that we discussed in the context of discrete-time problems, and the traveling salesman Example 1.2.3 in particular.

The following example illustrates the rollout algorithm (2.48) with a problem that has a special property: the base policy cost $J_{\pi,t}\big(x(t)\big)$ is independent of $x(t)$ (it depends only on t), so that

$$\nabla_x J_{\pi,t}\big(x(t)\big) \equiv 0.$$

In this case, in view of Eq. (2.46), the rollout policy is myopic. It turns out that the optimal policy in this example is also myopic, so that the rollout policy is optimal, even though the base policy is very poor.

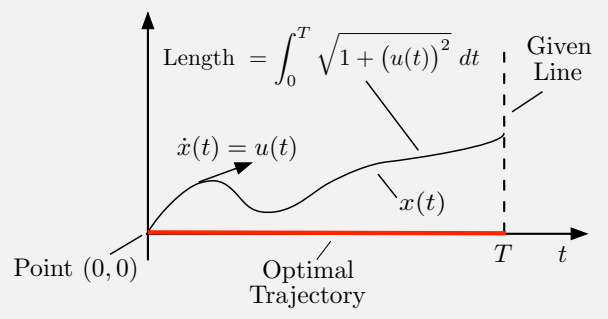


Figure 2.6.1 Problem of finding a curve of minimum length from a given point to a given line, and its formulation as a calculus of variations problem.

Example 2.6.1 (A Calculus of Variations Problem)

This is a simple example from the classical context of calculus of variations (see [Ber17a], Example 7.1.3). The problem is to find a minimum length curve that starts at a given point and ends at a given line. Without loss of generality, let $(0,0)$ be the given point, and let the given line be the vertical line that passes through $(T,0)$, as shown in Fig. 2.6.1.

Let $\big(t,x(t)\big)$ be the points of the curve, where $0 \le t \le T$. The portion of the curve joining the points $\big(t,x(t)\big)$ and $\big(t+dt,x(t+dt)\big)$ can be approximated, for small dt, by the hypotenuse of a right triangle with sides dt and $\dot{x}(t)dt$. Thus the length of this portion is

$$\sqrt{(dt)^2 + \big(\dot{x}(t)\big)^2(dt)^2},$$

which is equal to

$$\sqrt{1+\big(\dot{x}(t)\big)^2}\,dt.$$

The length of the entire curve is the integral over $[0,T]$ of this expression, so the problem is to

$$\begin{aligned} &\text{minimize} \quad \int_0^T \sqrt{1+\big(\dot{x}(t)\big)^2}\,dt \\ &\text{subject to} \quad x(0) = 0. \end{aligned}$$

To reformulate the problem as a continuous-time optimal control problem, we introduce a control u and the system equation

$$\dot{x}(t) = u(t), \qquad x(0) = 0.$$

Our problem then takes the form

$$\text{minimize} \quad \int_0^T \sqrt{1+\big(u(t)\big)^2}\,dt.$$

This is a problem that fits our continuous-time optimal control framework, with

$$h\big(x(t),u(t),t\big)=u(t),\quad g\big(x(t),u(t),t\big)=\sqrt{1+\big(u(t)\big)^2},\quad G\big(x(T)\big)=0.$$

Consider now a base policy π whereby the control depends only on t and not on x. Such a policy has the form

$$\mu_t\big(x(t)\big)=\beta(t),\qquad \text{for all } x(t),$$

where $\beta(t)$ is some scalar function. For example, $\beta(t)$ may be constant, $\beta(t)\equiv\bar{\beta}$ for some scalar $\bar{\beta}$, which yields a straight line trajectory that starts at $(0,0)$ and makes an angle ϕ with the horizontal with $\tan(\phi)=\bar{\beta}$. The cost function of the base policy is

$$J_{\pi,t}\big(x(t)\big)=\int_t^T\sqrt{1+\beta(\tau)^2}\,d\tau,$$

which is independent of $x(t)$, so that $\nabla_x J_{\pi,t}\big(x(t)\big)\equiv 0$. Thus, from the minimization of Eq. (2.48), we have

$$\tilde{\mu}_t\big(x(t)\big)\in\arg\min_{u(t)\in\Re}\sqrt{1+\big(u(t)\big)^2},$$

and the rollout policy is

$$\tilde{\mu}_t\big(x(t)\big)\equiv 0.$$

This is the optimal policy: it corresponds to the horizontal straight line that starts at $(0,0)$ and ends at $(T,0)$.

Rollout with General Base Heuristics - Sequential Improvement

An extension of the rollout algorithm (2.48) is to use a more general base heuristic whose cost function $H_t\big(x(t)\big)$ can be evaluated by simulation. This rollout algorithm has the form

$$\tilde{\mu}(t)\in\arg\min_{u(t)\in U}\Big[g\big(x(t),u(t),t\big)+\nabla_x H_t\big(x(t)\big)'h\big(x(t),u(t),t\big)\Big].$$

Here the policy cost function $J_{\pi,t}$ is replaced by a more general differentiable function H_t, obtainable through a base heuristic, which may lack the sequential consistency property that is inherent in policies.

We will now show a cost improvement property of the rollout algorithm based on the natural condition

$$H_T\big(\tilde{x}(T)\big)=G\big(\tilde{x}(T)\big),\tag{2.49}$$

and the assumption

$$\min_{u(t)\in U}\Big[g\big(x(t),u(t),t\big)+\nabla_t H_t\big(x(t)\big)+\nabla_x H_t\big(x(t)\big)'h\big(x(t),u(t),t\big)\Big]\leq 0, \tag{2.50}$$

for all $\big(x(t),t\big)$, where $\nabla_x H_t$ denotes gradient with respect to x, and $\nabla_t H_t$ denotes gradient with respect to t. This assumption is the continuous-time analog of the sequential improvement condition of Definition 2.3.2 [cf. Eq. (2.15)]. Under this assumption, we will show that

$$J_{\tilde\pi,0}\big(x(0)\big)\leq H_0\big(x(0)\big), \tag{2.51}$$

i.e., the cost of the rollout policy starting from the initial state $x(0)$ is no worse than the base heuristic cost starting from the same initial state.

Indeed, let $\big\{\tilde x(t)\mid t\in[0,T]\big\}$ and $\big\{\tilde u(t)\mid t\in[0,T]\big\}$ be the state and control trajectories generated by the rollout policy starting from $x(0)$. Then the sequential improvement condition (2.50) yields

$$g\big(\tilde x(t),\tilde u(t),t\big)+\nabla_t H_t\big(\tilde x(t)\big)+\nabla_x H_t\big(\tilde x(t)\big)'h\big(\tilde x(t),\tilde u(t),t\big)\leq 0$$

for all t, and by integration over $[0,T]$, we obtain

$$\int_0^T g\big(\tilde x(t),\tilde u(t),t\big)\,dt+\int_0^T\Big(\nabla_t H_t\big(\tilde x(t)\big)+\nabla_x H_t\big(\tilde x(t)\big)'h\big(\tilde x(t),\tilde u(t),t\big)\Big)\,dt\leq 0. \tag{2.52}$$

The second integral above can be written as

$$\int_0^T\Big(\nabla_t H_t\big(\tilde x(t)\big)+\nabla_x H_t\big(\tilde x(t)\big)'h\big(\tilde x(t),\tilde u(t),t\big)\Big)\,dt = \int_0^T\Big(\nabla_t H_t\big(\tilde x(t)\big)+\nabla_x H_t\big(\tilde x(t)\big)'\frac{d\tilde x(t)}{dt}\Big)\,dt,$$

and its integrand is the total differential with respect to time: $\frac{d}{dt}\Big(H_t\big(\tilde x(t)\big)\Big)$. Thus we obtain from Eq. (2.52)

$$\begin{aligned}\int_0^T g\big(\tilde x(t),\tilde u(t),t\big)\,dt+\int_0^T\frac{d}{dt}\Big(H_t\big(\tilde x(t)\big)\Big)\,dt\\ =\int_0^T g\big(\tilde x(t),\tilde u(t),t\big)\,dt+H_T\big(\tilde x(T)\big)-H_0\big(\tilde x(0)\big)\leq 0.\end{aligned} \tag{2.53}$$

Since $H_T\big(\tilde x(T)\big)=G\big(\tilde x(T)\big)$ [cf. Eq. (2.49)] and $\tilde x(0)=x(0)$, from Eq. (2.53) [which is a direct consequence of the sequential improvement condition (2.50)], it follows that

$$J_{\tilde\pi,0}\big(x(0)\big)=\int_0^T g\big(\tilde x(t),\tilde u(t),t\big)\,dt+G\big(\tilde x(T)\big)\leq H_0\big(x(0)\big),$$

thus proving the cost improvement property (2.51).

Note that the sequential improvement condition (2.50) is satisfied if H_t is the cost function $J_{\pi,t}$ corresponding to a base policy π. The reason is that for any policy $\pi = \{\mu_t(x(t)) \mid 0 \leq t \leq T\}$ [cf. Eq. (2.47)], the analog of the DP algorithm (under the requisite mathematical conditions) takes the form

$$0 = g\big(x(t), \mu_t(x(t)), t\big) + \nabla_t J_{\pi,t}\big(x(t)\big) + \nabla_x J_{\pi,t}\big(x(t)\big)' h\big(x(t), \mu_t(x(t)), t\big). \tag{2.54}$$

In continuous-time optimal control theory, this is known as the *Hamilton-Jacobi-Bellman equation*. It is a partial differential equation, which may be viewed as the continuous-time analog of the DP algorithm for a single policy; there is also a Hamilton-Jacobi-Bellman equation for the optimal cost function $J_t^*\big(x(t)\big)$ (see optimal control textbook accounts, such as [Ber17a], Section 7.2, and the references cited there). As illustration, the reader may verify that the cost function of the base policy used in the calculus of variations problem of Example 2.6.1 satisfies this equation. It can be seen from the Hamilton-Jacobi-Bellman Eq. (2.54) that when $H_t = J_{\pi,t}$, the sequential improvement condition (2.50) and the cost improvement property (2.51) hold.

Approximating Cost Function Differences

Let us finally note that the preceding analysis suggests that when dealing with a discrete-time problem with a long horizon N, a system equation $x_{k+1} = f_k(x_k, u_k)$, and a small cost per stage $g_k(x_k, u_k)$ relative to the optimal cost-to-go function $J_{k+1}^*\big(f_k(x_k, u_k)\big)$, it is worth considering an alternative implementation of the approximation in value space scheme. In particular, we should consider approximating the cost differences

$$D_k^*(x_k, u_k) = J_{k+1}^*\big(f_k(x_k, u_k)\big) - J_k^*(x_k)$$

instead of approximating the optimal cost-to-go functions $J_{k+1}^*\big(f_k(x_k, u_k)\big)$. The one-step-lookahead minimization (2.41) should then be replaced by

$$\tilde{\mu}_k(x_k) \in \arg\min_{u_k \in U_k(x_k)} \big[g_k(x_k, u_k) + \tilde{D}_k(x_k, u_k)\big],$$

where $\tilde{D}_k$ is the approximation to D_k^*.

Note also that while for continuous-time problems, the idea of approximating the gradient of the optimal cost function is essential and comes out naturally from the analysis, for discrete-time problems, approximating cost-to-go differences rather than cost functions is optional and should be considered in the context of a given problem. Methods along this line include advantage updating, cost shaping, biased aggregation, and the use of baselines, for which we refer to the books [BeT96], [Ber19a], and [Ber20a]. A special method to explicitly approximate cost function differences is *differential training*, which was proposed in the author's paper [Ber97b], and was also discussed in Section 4.3.4 of the book [Ber20a].

The Case of Zero Cost per Stage

The most extreme case of small stage costs arises when the cost per stage is zero for all states, while a nonzero cost may be incurred only at termination. This type of cost structure occurs, among others, in games such as chess and backgammon. It also occurs in several other contexts, including constraint programming problems (Section 2.1), where there is not even a terminal cost, just constraints to be satisfied.

Under these circumstances, the idea of approximating cost-to-go differences that we have just discussed may not be effective, and applying approximation in value space may involve serious challenges. An advisable remedy is to resort to longer lookahead, either through multistep lookahead minimization, or through some form of truncated rollout, as it is done in the AlphaZero and TD-Gammon programs. In addition, an artificial terminal cost function approximation should be introduced, possibly obtained through off-line training as in the AlphaZero and TD-Gammon programs. Another possibility is to obtain a terminal cost function by using some form of problem approximation (solving a simpler problem, in place of the original). Aggregation, discussed in Section 3.5, is one of the possbilities along this line.

2.7 STOCHASTIC ROLLOUT AND MONTE CARLO TREE SEARCH

We will now discuss the extension of the rollout algorithm to stochastic DP problems with a finite number of control and disturbances at every stage. We will restrict ourselves to the case where the base heuristic is a policy $\pi = \{\mu_0, \ldots, \mu_{N-1}\}$. The rollout policy applies at state x_k the control $\tilde{\mu}_k(x_k)$ given by the minimization

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

Equivalently, the rollout policy $\tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$ is obtained by minimization over the Q-factors $Q_{k,\pi}(x_k, u_k)$ of the base policy:

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

where

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

We first establish that the cost improvement property that we showed for deterministic problems under the sequential consistency condition carries through for stochastic problems. In particular, let us denote by $J_{k,\pi}(x_k)$

the cost corresponding to starting the base policy at state x_k, and by $J_{k,\tilde{\pi}}(x_k)$ the cost corresponding to starting the rollout algorithm at state x_k. We claim that

$$J_{k,\tilde{\pi}}(x_k) \leq J_{k,\pi}(x_k), \qquad \text{for all } x_k \text{ and } k. \tag{2.55}$$

We prove this inequality by induction similar to the deterministic case [cf. Eq. (2.14)]. Clearly it holds for $k = N$, since

$$J_{N,\tilde{\pi}} = J_{N,\pi} = g_N.$$

Assuming that it holds for index $k+1$, we have for all x_k,

$$\begin{aligned} J_{k,\tilde{\pi}}(x_k) &= E\Big\{ g_k\big(x_k, \tilde{\mu}_k(x_k), w_k\big) + J_{k+1,\tilde{\pi}}\Big(f_k\big(x_k, \tilde{\mu}_k(x_k), w_k\big)\Big)\Big\} \\ &\leq E\Big\{ g_k\big(x_k, \tilde{\mu}_k(x_k), w_k\big) + J_{k+1,\pi}\Big(f_k\big(x_k, \tilde{\mu}_k(x_k), w_k\big)\Big)\Big\} \\ &= \min_{u_k \in U_k(x_k)} E\Big\{ g_k(x_k, u_k, w_k) + J_{k+1,\pi}\big(f_k(x_k, u_k, w_k)\big)\Big\} \\ &\leq E\Big\{ g_k\big(x_k, \mu_k(x_k), w_k\big) + J_{k+1,\pi}\Big(f_k\big(x_k, \mu_k(x_k), w_k\big)\Big)\Big\} \\ &= J_{k,\pi}(x_k), \end{aligned} \tag{2.56}$$

where:

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

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

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

(d) The final equality is the DP equation for the base policy π.

The induction proof of the cost improvement property is thus complete.

The preceding cost improvement argument assumes that the cost functions $J_{k+1,\pi}$ of the base policy are calculated exactly. In practice, truncated rollout with terminal cost function approximation and limited simulation may be used to approximate $J_{k+1,\pi}$. In this case the cost function of the rollout policy can still be viewed as the result of a Newton step in the context of an approximation in value space scheme. Moreover, the cost improvement property can still be proved under some conditions that we will not discuss in these notes; see the books [Ber12], [Ber19a], and [Ber20a].

Some Rollout Examples

Similar to deterministic problems, it has been observed empirically that for stochastic problems the rollout policy not only does not deteriorate

the performance of the base policy, but also typically produces substantial cost improvement, thanks to its underlying Newton step; see also the case studies referenced at the end of the chapter. To emphasize this point, we provide here an example of an nontrivial optimal stopping problem where the rollout policy is actually optimal, despite the fact that the base policy is rather naive. Such behavior is of course special and nontypical, but highlights the nature of the cost improvement property of rollout.

Example 2.7.1 (Optimal Stopping and Rollout Optimality)

Optimal stopping problems are characterized by the availability, at each state, of a control that stops the evolution of the system. We will consider a problem with two control choices: at each stage we observe the current state of the system and decide whether to continue or to stop the process. We formulate this as an N-stage problem where stopping is mandatory at or before stage N.

Consider a stationary version of the problem (state and disturbance spaces, disturbance distribution, control constraint set, and cost per stage are the same for all times). At each state x_k and at time k, if we stop, the system moves to a termination state at a cost $C(x_k)$ and subsequently remains there at no cost. If we do not stop, the system moves to state $x_{k+1} = f(x_k, w_k)$ at cost $g(x_k, w_k)$. The terminal cost, assuming stopping has not occurred by the last stage, is $C(x_N)$. An example is a problem of optimal exercise of a financial option where x is the asset's price, $C(x) = x$, and $g(x, w) \equiv 0$.

The DP algorithm (for states other than the termination state) is given by

$$J_N^*(x_N) = C(x_N), \tag{2.57}$$

$$J_k^*(x_k) = \min \Big[C(x_k),\, E\Big\{ g(x_k, w_k) + J_{k+1}^*\big(f(x_k, w_k)\big) \Big\} \Big], \tag{2.58}$$

and it is optimal to stop at time k for states x in the set

$$S_k = \Big\{ x \;\Big|\; C(x) \leq E\Big\{ g(x, w) + J_{k+1}^*\big(f(x, w)\big) \Big\} \Big\}.$$

Consider now the rather primitive base policy π, whereby *we stop at every state* x. Thus we have for all x_k and k,

$$J_{k,\pi}(x_k) = C(x_k).$$

The rollout policy is stationary and can be computed on-line relatively easily, since $J_{k,\pi}$ is available in closed form. In particular, the rollout policy is to stop at x_k if

$$C(x_k) \leq E\Big\{ g(x_k, w_k) + C\big(f(x_k, w_k)\big) \Big\},$$

i.e., if x_k is in the set S_{N-1}, and otherwise to continue.

The rollout policy also has an intuitive interpretation: it stops at the states for which it is better to stop rather than continue for one more stage

and then stop. A policy of this type turns out to be optimal in several types of stopping applications. Let us provide a condition that guarantees its optimality.

We have from the DP Eqs. (2.57)-(2.58),

$$J^*_{N-1}(x) \leq J^*_N(x), \qquad \text{for all } x,$$

and using this fact in the DP equation (2.58), we obtain inductively

$$J^*_k(x) \leq J^*_{k+1}(x), \qquad \text{for all } x \text{ and } k.$$

Using this fact and the definition of S_k we see that

$$S_0 \subset \cdots \subset S_k \subset S_{k+1} \subset \cdots \subset S_{N-1}. \tag{2.59}$$

We will now consider a condition guaranteeing that all the stopping sets S_k are equal. Suppose that the set S_{N-1} is *absorbing* in the sense that if a state belongs to S_{N-1} and we decide to continue, the next state will also be in S_{N-1}:

$$f(x,w) \in S_{N-1}, \qquad \text{for all } x \in S_{N-1},\ w. \tag{2.60}$$

We will show that equality holds in Eq. (2.59) and for all k we have

$$S_k = S_{N-1} = \left\{ x \in S \;\middle|\; C(x) \leq E\Big\{ g(x,w) + C\big(f(x,w)\big) \Big\} \right\}.$$

Indeed, by the definition of S_{N-1}, we have

$$J^*_{N-1}(x) = C(x), \qquad \text{for all } x \in S_{N-1},$$

and using Eq. (2.60) we obtain for $x \in S_{N-1}$

$$E\Big\{ g(x,w) + J^*_{N-1}\big(f(x,w)\big) \Big\} = E\Big\{ g(x,w) + C\big(f(x,w)\big) \Big\} \geq C(x).$$

Therefore, stopping is optimal for all $x_{N-2} \in S_{N-1}$ or equivalently $S_{N-1} \subset S_{N-2}$. This together with Eq. (2.59) implies $S_{N-2} = S_{N-1}$. Proceeding similarly, we obtain $S_k = S_{N-1}$ for all k. Thus the optimal policy is to stop if and only if the state is within the set S_{N-1}, which is precisely the set of states where the rollout policy stops.

In conclusion, if condition (2.60) holds (the one-step stopping set S_{N-1} is absorbing), the rollout policy is optimal. Moreover, the preceding analysis [cf. Eq. (2.59)] can be used to show that even if the one-step stopping set S_{N-1} is not absorbing, the rollout policy stops and is optimal within the set of states $x \in \cap_k S_k$, and correctly continues within the set of states $x \notin S_{N-1}$. Contrary to the optimal policy, it also stops within the subset of states $x \in S_{N-1}$ that are not in $\cap_k S_k$. Thus, even in the absence of condition (2.60), the rollout policy is quite sensible even though the base policy is not.

We next discuss a special case of the preceding example. Again the one-step lookahead/rollout policy is optimal, despite the fact that the base policy is poor. Related examples can be found in Chapter 3 of the DP textbook [Ber17a].

Example 2.7.2 (The Rational Burglar)

A burglar may at any night k choose to retire with his accumulated earnings x_k or enter a house and bring home a random amount w_k. However, in the latter case he gets caught with probability p, and then he is forced to terminate his activities and forfeit all of his earnings thus far. The amounts w_k are independent, identically distributed with mean $\overline{w}$. The problem is to find a policy that maximizes the burglar's expected earnings over N nights.

We can formulate this problem as a stopping problem with two actions (retire or continue) and a state space consisting of the real line, the retirement state, and a special state corresponding to the burglar getting caught. The DP algorithm is given by

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

$$J_k^*(x_k) = \max\Big[x_k,\, (1-p)E\big\{J_{k+1}^*(x_k + w_k)\big\}\Big].$$

The one-step stopping set is

$$S_{N-1} = \big\{x \mid x \geq (1-p)(x+\overline{w})\big\} = \left\{x \;\middle|\; x \geq \frac{(1-p)\overline{w}}{p}\right\},$$

(more accurately this set together with the special state corresponding to the burglar's arrest). Since this set is absorbing in the sense of Eq. (2.60), we see that the one-step lookahead/rollout policy by which the burglar retires when his earnings reach or exceed $(1-p)\overline{w}/p$ is optimal. Note that the base policy of the burglar is the "timid" policy of always retiring, regardless of his accumulated earnings, which is far from optimal.

2.7.1 Simplified Rollout and Policy Iteration

The cost improvement property (2.55) also holds for the simplified version of the rollout algorithm (cf. Section 2.3.4) where the rollout policy is defined by

$$\tilde{\mu}_k(x_k) \in \arg\min_{u_k \in \overline{U}_k(x_k)} Q_{k,\pi}(x_k, u_k), \tag{2.61}$$

for a subset $\overline{U}_k(x_k) \subset U_k(x_k)$ that contains the base policy control $\mu_k(x_k)$. The proof is obtained by replacing the last inequality in the argument of Eq. (2.56),

$$\min_{u_k \in U_k(x_k)} Q_{k,\pi}(x_k, u_k) \leq Q_{k,\pi}\big(x_k, \mu_k(x_k)\big),$$

with the inequality

$$\min_{u_k \in \overline{U}_k(x_k)} Q_{k,\pi}(x_k, u_k) \leq Q_{k,\pi}\big(x_k, \mu_k(x_k)\big).$$

The simplified rollout algorithm (2.61) may be implemented in a number of ways, including control constraint discretization/approximation, a random search algorithm, or a one-agent-at-a-time minimization process, as in multiagent rollout.

The simplified rollout idea can also be used within the infinite horizon policy iteration (PI) context. In particular, instead of the minimization

$$\tilde{\mu}(x) \in \arg\min_{u \in U(x)} E_w\Big\{g(x,u,w) + \alpha J_\mu\big(f(x,u,w)\big)\Big\}, \qquad \text{for all } x, \quad (2.62)$$

in the policy improvement operation, it is sufficient for cost improvement to generate a new policy $\tilde{\mu}$ that satisfies for all x,

$$E_w\Big\{g\big(x,\tilde{\mu}(x),w\big)+\alpha J_\mu\big(f(x,\tilde{\mu}(x),w)\big)\Big\} \leq E_w\Big\{g(x,u,w)+\alpha J_\mu\big(f(x,u,w)\big)\Big\}.$$

This cost improvement property is the critical argument for proving convergence of the PI algorithm and its variations to the optimal cost function and policy; see the corresponding proofs in the books [Ber17a] and [Ber19a].

2.7.2 Certainty Equivalence Approximations

As in the case of deterministic DP problems, it is possible to use ℓ-step lookahead, with the aim to improve the performance of the policy obtained through approximation in value space. This, however, can be computationally expensive, because the lookahead graph expands fast as ℓ increases, due to the stochastic character of the problem. Using *certainty equivalence* (CE for short) is an important approximation approach for dealing with this difficulty, as it reduces the size of the ℓ-step minimization graph. Moreover, CE mitigates the potentially excessive simulation because it reduces the stochastic variance of the Q-factors calculated by the method at each stage.

In the pure but somewhat flawed version of this approach, when solving the ℓ-step lookahead minimization problem, we simply replace *all* of the uncertain quantities $w_k, w_{k+1}, \ldots, w_{k+\ell-1}, \ldots, w_{N-1}$ by some nominal value $\overline{w}$, thus making that problem fully deterministic. Unfortunately, this affects significantly the character of the approximation: when w_k is replaced by a deterministic quantity the Newton step interpretation of the underlying approximation in value space scheme is lost to a great extent.

Still, we may largely correct this difficulty, while retaining substantial simplification, by using CE for *only after the first stage* of the ℓ-step lookahead. We can do this with a CE scheme whereby only the uncertain quantities $w_{k+1}, \ldots, w_{N-1}$ are replaced by a deterministic value $\overline{w}$, while w_k is treated as a stochastic quantity.

This approach, first proposed in the paper by Bertsekas and Castañon [BeC99], has an important advantage: *it maintains the Newton step character of the approximation in value space scheme*. In particular, the function

$J_{\tilde{\mu}}$ of the ℓ-step lookahead policy $\tilde{\mu}$ obtained is generated by a Newton step, applied to the function obtained by the last $\ell - 1$ minimization steps (modified by CE, and applied to the terminal cost function approximation); see the monograph [Ber20a] for a discussion. Thus the benefit of the fast convergence of Newton's method is restored. In fact based on insights derived from this Newton step interpretation, it appears that the performance penalty for the CE approximation is typically small. At the same time the ℓ-step lookahead minimization involves only one stochastic step, the first one, and hence potentially a much "thinner" lookahead graph, than the ℓ-step minimization that does not involve any CE-type approximations; see Fig. 2.7.1. Moreover, the ideas of tree pruning and iterative deepening, which we have discussed in Section 2.4 for deterministic multistep lookahead, come into play when the CE approximation is used.

2.7.3 Simulation-Based Implementation of the Rollout Algorithm

A conceptually straightforward way to compute the rollout control at a given state x_k and time k is to consider each possible control $u_k \in U_k(x_k)$, and to generate a "large" number of simulated trajectories of the system starting from (x_k, u_k). Thus a simulated trajectory is obtained from

$$x_{i+1} = f_i\big(x_i, \mu_i(x_i), w_i\big), \qquad i = k+1, \ldots, N-1,$$

where $\{\mu_{k+1}, \ldots, \mu_{N-1}\}$ is the tail portion of the base policy, the starting state of the simulated trajectory is

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

and the disturbance sequence $\{w_k, \ldots, w_{N-1}\}$ is obtained by random sampling. The costs of the trajectories corresponding to a pair (x_k, u_k) can be viewed as samples of the Q-factor

$$Q_{k,\pi}(x_k, u_k) = E\Big\{g_k(x_k, u_k, w_k) + J_{k+1,\pi}\big(f_k(x_k, u_k, w_k)\big)\Big\},$$

where $J_{k+1,\pi}$ is the cost-to-go function of the base policy, i.e., $J_{k+1,\pi}(x_{k+1})$ is the cost of using the base policy starting from x_{k+1}. For problems with a large number of stages, it is also common to truncate the rollout trajectories and add a terminal cost function approximation as compensation for the resulting error.

By Monte Carlo averaging of the costs of the sample trajectories plus the terminal cost (if any), we obtain an approximation to the Q-factor $Q_{k,\pi}(x_k, u_k)$ for each $u_k \in U_k(x_k)$, denoted by $\tilde{Q}_{k,\pi}(x_k, u_k)$. We then compute the (approximate) rollout control $\tilde{\mu}_k(x_k)$ with the minimization

$$\tilde{\mu}_k(x_k) \in \arg\min_{u_k \in U_k(x_k)} \tilde{Q}_{k,\pi}(x_k, u_k). \tag{2.63}$$

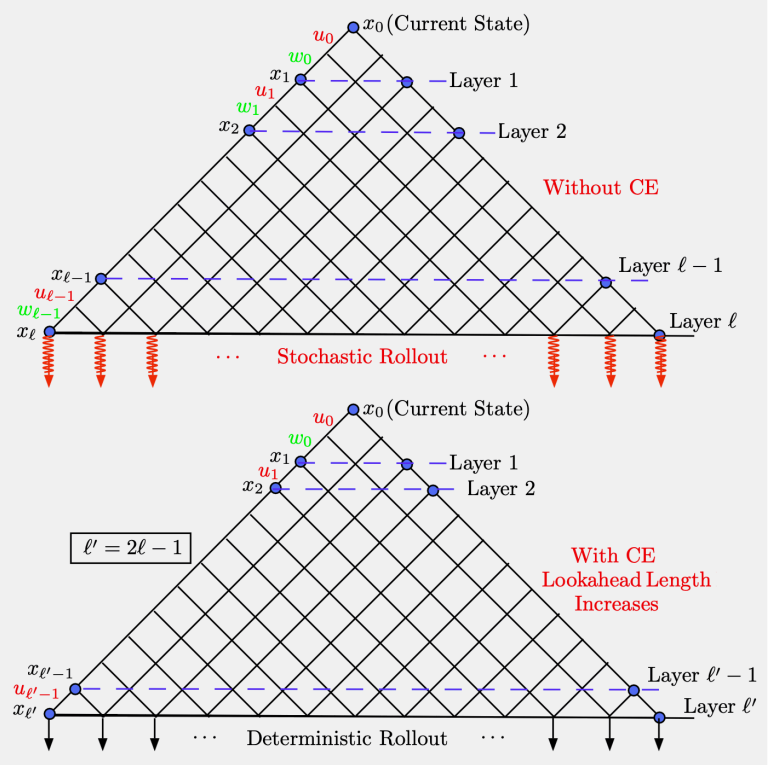


Figure 2.7.1 Illustration of multistep lookahead for stochastic problems with the CE approximation, applied at the states after the first layer of states of the multistep lookahead tree. The figure on the top (or the bottom) illustrates the lookahead tree without (or with, respectively) CE. It can be seen that with CE, the lookahead tree grows much faster (the layers contain more states). In particular, the "height" of the ℓ-step lookahead graph without the CE approximation is the same as the "height" of a ℓ'-step lookahead graph with the CE approximation, where $\ell' = 2\ell - 1$. Moreover, with a number m of controls per state, and a number n of disturbances per state-control pair, the number of leaves of the ℓ-step lookahead tree is estimated as $O(mn\ell)$ without CE and $O\big(m(n+\ell)\big)$ with CE.

Example 2.7.3 (Backgammon)

The first impressive application of rollout was given for the ancient two-player game of backgammon, in the paper by Tesauro and Galperin [TeG96]; see Fig. 2.7.2. They implemented a rollout algorithm, which attained a level of play that was better than all computer backgammon programs, and eventually better than the best humans. Tesauro had proposed earlier the use of one-step and two-step lookahead with lookahead cost function approximation provided by a neural network, resulting in a backgammon program called TD-Gammon [Tes89a], [Tes89b], [Tes92], [Tes94], [Tes95], [Tes02]. TD-Gammon was trained with an approximate policy iteration method, and was used as

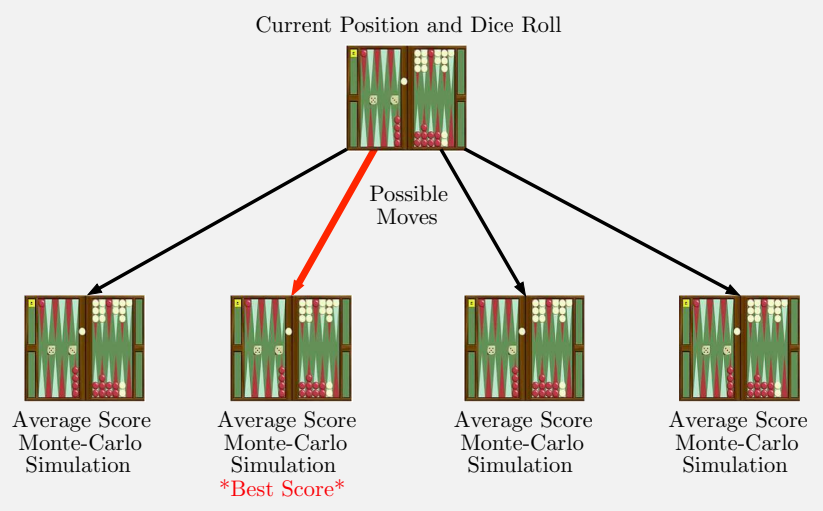


Figure 2.7.2 Illustration of rollout for backgammon. At a given position and roll of the dice, the set of all possible moves is generated, and the outcome of the game for each move is evaluated by "rolling out" (simulating to the end) many games using a suboptimal/heuristic backgammon player (the TD-Gammon player was used for this purpose in [TeG96]), and by Monte Carlo averaging the scores. The move that results in the best average score is selected for play.

the base policy (for each of the two players) to simulate game trajectories. The rollout algorithm also involved truncation of long game trajectories, using a terminal cost function approximation based on TD-Gammon's position evaluation. Game trajectories are of course random, since they involve the use of dice at each player's turn. Thus the scores of many trajectories have to be generated and Monte Carlo averaged to assess the probability of a win from a given position.

An important issue to consider here is that backgammon is a two-player game and not an optimal control problem that involves a single decision maker. While there is a DP theory for sequential zero-sum games, this theory has not been covered in these notes. Thus how are we to interpret rollout algorithms in the context of two-player games, with both players using some base policy? The answer is to view the game as a (one-player) optimal control problem, where one of the two players passively uses the base policy exclusively (TD-Gammon in the present example). The other player takes the role of the optimizer, and actively tries to improve on his base policy (TD-Gammon) by using rollout. Thus "policy improvement" in the context of the present example means that when playing against a TD-Gammon opponent, the rollout player achieves a better score on the average than if he/she were to play with the TD-Gammon strategy. In particular, the theory does not guarantee that a rollout player that is trained using TD-Gammon for both players will do better than TD-Gammon would against a non-TD-Gammon opponent. While this is a plausible practical hypothesis, it is one that can only

be tested empirically. In fact relevant counterexamples have been constructed for the game of Go using "adversarial" optimization techniques; see Wang et al. [WGB22], and also our discussion on minimax problems in Section 2.12.

Most of the currently existing computer backgammon programs descend from TD-Gammon. Rollout-based backgammon programs are the most powerful in terms of performance, consistent with the principle that a rollout algorithm performs better than its base heuristic. However, they are too time-consuming for real-time play (without parallel computing hardware), because of the extensive on-line simulation requirement at each move.† They have been used in a limited diagnostic way to assess the quality of neural network-based programs (many articles and empirical works on computer backgammon are posted on-line; see e.g., http://www.bkgm.com/articles/page07.html).

2.7.4 Variance Reduction in Rollout - Comparing Advantages

When using simulation, sampling is often organized to effect *variance reduction*. By this we mean that for a given problem, the collection and use of samples is structured so that the variance of the simulation error is made smaller, with the same amount of simulation effort. There are several methods of this type for which we refer to textbooks on simulation (see, e.g., Ross [Ros12], and Rubinstein and Kroese [RuK1]).

In this section we discuss a method to reduce the effects of the simulation error in the calculation of the Q-factors in the context of rollout. The key idea is that the selection of the rollout control depends on the values of the Q-factor differences

$$\tilde{Q}_{k,\pi}(x_k, u_k) - \tilde{Q}_{k,\pi}(x_k, \hat{u}_k)$$

for all pairs of controls $(u_k, \hat{u}_k)$. These values must be computed accurately, so that the controls u_k and $\hat{u}_k$ can be accurately compared. On the other hand, the simulation/approximation errors in the computation of the individual Q-factors $\tilde{Q}_{k,\pi}(x_k, u_k)$ may be magnified through the preceding differencing operation.

An approach to counteract this type of simulation error magnification is to approximate the Q-factor difference $\tilde{Q}_{k,\pi}(x_k, u_k) - \tilde{Q}_{k,\pi}(x_k, \hat{u}_k)$ by sampling the difference

$$C_k(x_k, u_k, \mathbf{w}_k) - C_k(x_k, \hat{u}_k, \mathbf{w}_k), \tag{2.64}$$

where $\mathbf{w}_k = (w_k, w_{k+1}, \ldots, w_{N-1})$ is the *same disturbance sequence* for the two controls u_k and $\hat{u}_k$, and

$$C_k(x_k, u_k, \mathbf{w}_k) = g_N(x_N) + g_k(x_k, u_k, w_k) + \sum_{i=k+1}^{N-1} g_i\big(x_i, \mu_i(x_i), w_i\big),$$

† The situation in backgammon is exacerbated by its high branching factor, i.e., for a given position, the number of possible successor positions is quite large, as compared for example with chess.

with $\{\mu_{k+1}, \ldots, \mu_{N-1}\}$ being the tail portion of the base policy. †

For a simple example that illustrates how this form of variance reduction works, suppose we want to calculate the difference $q_1 - q_2$ of two numbers q_1 and q_2 by subtracting two simulation samples $s_1 = q_1 + w_1$ and $s_2 = q_2 + w_2$, where w_1 and w_2 are zero mean random variables. Then $s_1 - s_2$ is unbiased in the sense that its mean is equal to $q_1 - q_2$. However, the variance of $s_1 - s_2$ decreases as the correlation of w_1 and w_2 increases. It is maximized when w_1 and w_2 are uncorrelated, and it is minimized (it is equal to 0) when w_1 and w_2 are equal.

The preceding example suggests a simulation scheme that is based on the difference (2.64) and involves a common disturbance $\mathbf{w}_k$ for u_k and $\hat{u}_k$. In particular, it may be far more accurate than the one obtained by differencing samples of $C_k(x_k, u_k, \mathbf{w}_k)$ and $C_k(x_k, \hat{u}_k, \hat{\mathbf{w}}_k)$, which involve two different disturbances $\mathbf{w}_k$ and $\hat{\mathbf{w}}_k$. Indeed, by introducing the zero mean sample errors

$$D_k(x_k, u_k, \mathbf{w}_k) = C_k(x_k, u_k, \mathbf{w}_k) - \tilde{Q}_{k,\pi}(x_k, u_k),$$

it can be seen that the variance of the error in estimating $\tilde{Q}_{k,\pi}(x_k, u_k) - \tilde{Q}_{k,\pi}(x_k, \hat{u}_k)$ with the former method will be no larger than with the latter method if and only if

$$E_{\mathbf{w}_k, \hat{\mathbf{w}}_k}\Big\{\big|D_k(x_k, u_k, \mathbf{w}_k) - D_k(x_k, \hat{u}_k, \hat{\mathbf{w}}_k)\big|^2\Big\} \geq E_{\mathbf{w_k}}\Big\{\big|D_k(x_k, u_k, \mathbf{w}_k) - D_k(x_k, \hat{u}_k, \mathbf{w}_k)\big|^2\Big\}.$$

By expanding the quadratic forms and using the fact $E\big\{D_k(x_k, u_k, \mathbf{w}_k)\big\} = 0$, we see that this condition is equivalent to

$$E\big\{D_k(x_k, u_k, \mathbf{w}_k) D_k(x_k, \hat{u}_k, \mathbf{w}_k)\big\} \geq 0; \tag{2.65}$$

i.e., the errors $D_k(x_k, u_k, \mathbf{w}_k)$ and $D_k(x_k, \hat{u}_k, \mathbf{w}_k)$ being nonnegatively correlated. A little thought should convince the reader that this property is likely to hold in many types of problems.

Roughly speaking, the relation (2.65) holds if changes in the value of u_k (at the first stage) have little effect on the value of the error $D_k(x_k, u_k, \mathbf{w}_k)$ relative to the effect induced by the randomness of $\mathbf{w}_k$. To see this, suppose that there exists a scalar $\gamma < 1$ such that, for all x_k, u_k, and $\hat{u}_k$, there holds

$$E\Big\{\big|D_k(x_k, u_k, \mathbf{w}_k) - D_k(x_k, \hat{u}_k, \mathbf{w}_k)\big|^2\Big\} \leq \gamma E\Big\{\big|D_k(x_k, u_k, \mathbf{w}_k)\big|^2\Big\}. \tag{2.66}$$

† For this to be possible, we need to assume that the probability distribution of each disturbance w_i does not depend on x_i and u_i.

Then we have, by using the generic relation $ab \geq a^2 - |a| \cdot |b-a|$ for two scalars a and b,

$$\begin{aligned} &D_k(x_k,u_k,\mathbf{w}_k)D_k(x_k,\hat{u}_k,\mathbf{w}_k) \\ &\qquad \geq \big|D_k(x_k,u_k,\mathbf{w}_k)\big|^2 \\ &\qquad\quad - \big|D_k(x_k,u_k,\mathbf{w}_k)\big| \cdot \big|D_k(x_k,\hat{u}_k,\mathbf{w}_k) - D_k(x_k,u_k,\mathbf{w}_k)\big|, \end{aligned}$$

from which we obtain

$$\begin{aligned} &E\big\{D_k(x_k,u_k,\mathbf{w}_k)D_k(x_k,\hat{u}_k,\mathbf{w}_k)\big\} \\ &\qquad \geq E\left\{\big|D_k(x_k,u_k,\mathbf{w}_k)\big|^2\right\} \\ &\qquad\quad - E\Big\{\big|D_k(x_k,u_k,\mathbf{w}_k)\big| \cdot \big|D_k(x_k,\hat{u}_k,\mathbf{w}_k) - D_k(x_k,u_k,\mathbf{w}_k)\big|\Big\} \\ &\qquad \geq E\left\{\big|D_k(x_k,u_k,\mathbf{w}_k)\big|^2\right\} - \frac{1}{2}E\left\{\big|D_k(x_k,u_k,\mathbf{w}_k)\big|^2\right\} \\ &\qquad\quad - \frac{1}{2}E\Big\{\big|D_k(x_k,\hat{u}_k,\mathbf{w}_k) - D_k(x_k,u_k,\mathbf{w}_k)\big|^2\Big\} \\ &\qquad \geq \frac{1-\gamma}{2}E\left\{\big|D_k(x_k,u_k,\mathbf{w}_k)\big|^2\right\}, \end{aligned}$$

where for the second inequality we use the generic relation

$$-|a| \cdot |b| \geq -\frac{1}{2}(a^2+b^2)$$

for two scalars a and b, and for the third inequality we use Eq. (2.66).

Thus, under the assumption (2.66), the condition (2.65) holds and guarantees that by averaging cost difference samples rather than differencing (independently obtained) averages of cost samples, the simulation error variance does not increase.

Let us finally note the potential benefit of using Q-factor differences in contexts other than rollout. In particular when approximating Q-factors $Q_{k,\pi}(x_k,u_k)$ using parametric architectures (Section 3.3 in the next chapter), it may be important to approximate and compare instead the differences

$$A_{k,\pi}(x_k,u_k) = Q_{k,\pi}(x_k,u_k) - \min_{u_k \in U_k(x_k)} Q_{k,\pi}(x_k,u_k).$$

The function $A_{k,\pi}(x_k,u_k)$ is also known as the *advantage of the pair* (x_k,u_k), and can serve just as well as $Q_{k,\pi}(x_k,u_k)$ for the purpose of comparing controls, but may work better in the presence of approximation errors. The use of advantages will be discussed further in Chapter 3.

2.7.5 Monte Carlo Tree Search

In our earlier discussion of simulation-based rollout implementation, we implicitly assumed that once we reach state x_k, we generate the same large number of trajectories starting from each pair (x_k, u_k), with $u_k \in U(x_k)$, to the end of the horizon. The drawback of this is threefold:

(a) The trajectories may be too long because the horizon length N is large (or infinite, in an infinite horizon context).

(b) Some of the controls u_k may be clearly inferior to others, and may not be worth as much sampling effort.

(c) Some of the controls u_k that appear to be promising, may be worth exploring better through multistep lookahead.

This has motivated multistep lookahead variants, generally referred to as *Monte Carlo tree search* (MCTS for short), which aim to trade off computational economy with a hopefully small risk of degradation in performance. Such variants involve, among others, early discarding of controls deemed to be inferior based on the results of preliminary calculations, and simulation that is limited in scope (either because of a reduced number of simulation samples, or because of a shortened horizon of simulation, or both).

A simple remedy for (a) above is to use rollout trajectories of reasonably limited length, with some terminal cost approximation at the end (in an extreme case, the rollout may be skipped altogether for some states, i.e., rollout trajectories have zero length). The terminal cost function may be very simple (such as zero) or may be obtained through some auxiliary calculation. In fact the base policy used for rollout may be used to construct the terminal cost function approximation, as noted for the rollout-based backgammon algorithm of Example 2.7.3. In particular, an approximation to the cost function of the base policy may be obtained by training some approximation architecture, such as a neural network (see Chapter 3), and may be used as a terminal cost function.

A simple but less straightforward remedy for (b) is to use some heuristic or statistical test to discard some of the controls u_k, as soon as this is suggested by the early results of simulation. Similarly, to implement (c) one may use some heuristic to increase the length of lookahead selectively for some of the controls u_k. This is similar to the incremental multistep rollout scheme for deterministic problems that we discussed in Section 2.4.3; see Fig. 2.4.6.

The MCTS approach can be based on sophisticated procedures for implementing and combining the ideas just described. The general idea is to use the interim results of the computation and statistical tests to focus the simulation effort along the most promising directions. Thus to implement MCTS with multistep lookahead, one needs to maintain a lookahead tree, which is expanded as the relevant Q-factors are evaluated by simulation,

and which balances *the competing desires of exploitation and exploration* (generate and evaluate controls that seem most promising in terms of performance versus assessing the potential of inadequately explored controls). Ideas that were developed in the context of multiarmed bandit problems have played an important role in the construction of this type of MCTS procedures (see the end-of-chapter references).

In the simple case of one-step lookahead, with Q-factors calculated by Monte Carlo simulation, MCTS fundamentally aims to find efficiently the minimum of the expected values of a finite number of random variables. This is illustrated in the following example.

Example 2.7.4 (Statistical Tests for Adaptive Sampling with One-Step Lookahead)

Let us consider a typical one-step lookahead selection strategy that is based on adaptive sampling. We are at a state x_k and we try to find a control $\tilde{u}_k$ that minimizes an approximate Q-factor

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

over $u_k \in U_k(x_k)$, with $\tilde{Q}_k(x_k, u_k)$ computed by averaging samples of the expression within braces. We assume that $U_k(x_k)$ contains m elements, which for simplicity are denoted $1, \ldots, m$. At the ℓth sampling period, knowing the outcomes of the preceding sampling periods, we select one of the m controls, say i_ℓ, and we draw a sample of $\tilde{Q}_k(x_k, i_\ell)$, whose value is denoted by S_{i_ℓ}. Thus after the nth sampling period we have an estimate $Q_{i,n}$ of the Q-factor of each control $i = 1, \ldots, m$ that has been sampled at least once, given by

$$Q_{i,n} = \frac{\sum_{\ell=1}^{n} \delta(i_\ell = i) S_{i_\ell}}{\sum_{\ell=1}^{n} \delta(i_\ell = i)},$$

where

$$\delta(i_\ell = i) = \begin{cases} 1 & \text{if } i_\ell = i, \\ 0 & \text{if } i_\ell \neq i. \end{cases}$$

Thus $Q_{i,n}$ is the *empirical mean* of the Q-factor of control i (total sample value divided by total number of samples), assuming that i has been sampled at least once.

After n samples have been collected, with each control sampled at least once, we may declare the control i that minimizes $Q_{i,n}$ as the "best" one, i.e., the one that truly minimizes the Q-factor $Q_k(x_k, i)$. However, there is a positive probability that there is an error: the selected control may not minimize the true Q-factor. In adaptive sampling, roughly speaking, we want to design the sample selection strategy and the criterion to stop the sampling, in a way that keeps the probability of error small (by allocating some sampling effort to all controls), and the number of samples limited (by not wasting

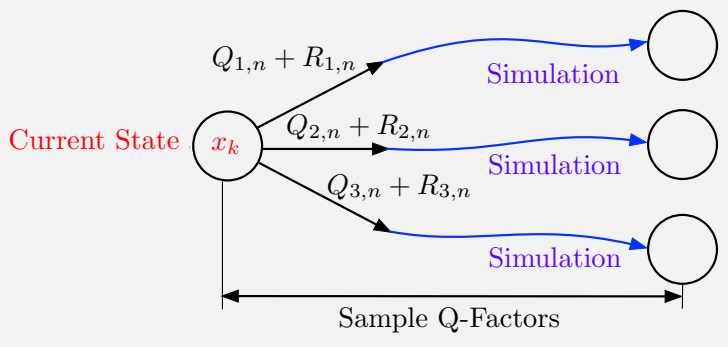


Figure 2.7.3 Illustration of one-step lookahead MCTS at a state x_k. The Q-factor sampled next corresponds to the control i with minimum sum of exploitation index (here taken to be the running average $Q_{i,n}$) and exploration index ($R_{i,n}$, possibly given by the UCB rule).

samples on controls i that appear inferior based on their empirical mean $Q_{i,n}$).

Intuitively, a good sampling policy will balance at time n the desires of exploitation and exploration (i.e., sampling controls that seem most promising, in the sense that they have a small empirical mean $Q_{i,n}$, versus assessing the potential of inadequately explored controls, those i that have been sampled a small number of times). Thus it makes sense to sample next the control i that minimizes the sum

$$T_{i,n} + R_{i,n}$$

of two indexes: an *exploitation index* $T_{i,n}$ and an *exploration index* $R_{i,n}$. Usually the exploitation index is chosen to be the empirical mean $Q_{i,n}$; see Fig. 2.7.3. The exploration index is based on a confidence interval formula and depends on the sample count

$$s_i = \sum_{\ell=1}^{n} \delta(i_\ell = i)$$

of control i. A frequently suggested choice is the UCB rule (upper confidence bound), which sets

$$R_{i,n} = -c\sqrt{\frac{\log n}{s_i}},$$

where c is a positive constant that is selected empirically (some analysis suggests values near $c = \sqrt{2}$, assuming that $Q_{i,n}$ is normalized to take values in the range $[-1, 0]$). The UCB rule, first proposed in the paper by Auer, Cesa-Bianchi, and Fischer [ACF02], has been extensively discussed in the literature both for one-step and for multistep lookahead [where it is called UCT (UCB applied to trees; see Kocsis and Szepesvari [KoS06])].†

† The paper [ACF02] refers to the rule given here as UCB1 and credits its motivation to the paper by Agrawal [Agr95]. The book by Lattimore and Szepesvari [LaS20] provides an extensive discussion of the UCB rule and its generalizations.

Its justification is based on probabilistic analyses that relate to the multiarmed bandit problem, and is beyond our scope. Alternatives to the UCB formula have been suggested, and in fact in the AlphaZero program, the exploitation term has a different form than the one above, and depends on the depth of lookahead (see Silver et al. [SHS17]).

Sampling policies for MCTS with multistep lookahead are based on similar sampling ideas to the case of one-step lookahead. A simulated trajectory is run from a node i of the lookahead tree that minimizes the sum $T_{i,n} + R_{i,n}$ of an exploitation index and an exploration index. There are several schemes of this type, but the details are beyond our scope and are often problem-dependent (see the end-of-chapter references).

A major success has been the use of MCTS in two-player game contexts, such as the AlphaGo program (Silver et al. [SHM16]), which performs better than the best humans in the game of Go. This program integrates several of the techniques discussed in these notes, including MCTS and rollout using a base policy that is trained off-line using a deep neural network. The AlphaZero program, which has performed spectacularly well against humans and other programs in the games of Go and chess (Silver et al. [SHS17]), bears some similarity with AlphaGo, and critically relies on MCTS, but does not use rollout in its on-line playing mode (it relies primarily on very long lookahead).

2.7.6 Randomized Policy Improvement by Monte Carlo Tree Search

We have described rollout and MCTS as schemes for policy improvement: start with a base policy, and compute an improved policy based on the results of one-step lookahead or multistep lookahead followed by simulation with the base policy. We have implicitly assumed that both the base policy and the rollout policy are deterministic in the sense that they map each state x_k into a unique control $\tilde{\mu}_k(x_k)$ [cf. Eq. (2.63)]. In some (even nonstochastic) contexts, success has been achieved with *randomized policies*, which map a state x_k to a probability distribution over the set of controls $U_k(x_k)$, rather than mapping onto a single control. In particular, the AlphaGo and AlphaZero programs use MCTS to generate and use for training purposes randomized policies, which specify at each board position the probabilities with which the various moves are selected.

A randomized policy can be used as a base policy in a rollout context in exactly the same way as a deterministic policy: for a given state x_k, we just generate sample trajectories and associated sample Q-factors, using probabilistically selected controls, starting from each leaf-state of the lookahead tree that is rooted at x_k. We then average the corresponding Q-factor samples. The rollout/improved policy, as described here, is a deterministic policy, i.e., it applies at x_k the control $\tilde{\mu}_k(x_k)$ that is "best" according to the results of the rollout [cf. Eq. (2.63)]. Still, however, if we

wish to generate an improved policy that is randomized, we can simply change the probabilities of different controls in the direction of the deterministic rollout policy. This can be done by increasing by some amount the probability of the "best" control $\tilde{\mu}_k(x_k)$ from its base policy level, while proportionally decreasing the probabilities of the other controls.

The use of MCTS provides a related method to "improve" a randomized policy. In the process of the adaptive simulation that is used in MCTS, we generate *frequency counts* of the different controls in $U_k(x_k)$, i.e., the proportion of rollout trajectories associated with each $u_k \in U_k(x_k)$. We can then obtain the rollout randomized policy by moving the probabilities of the base policy in the direction suggested by the frequency counts, i.e., increase the probability of high-count controls and reduce the probability of the others. This type of policy improvement is reminiscent of gradient-type methods, and has been successful in some contexts; see the end-of-chapter references for such policy improvement implementations in AlphaGo, AlphaZero, and other applications.

2.8 ROLLOUT FOR INFINITE-SPACES PROBLEMS - OPTIMIZATION HEURISTICS

We have considered so far finite control space applications of rollout, so there is a finite number of relevant Q-factors at each state x_k, which are evaluated by simulation and are exhaustively compared. When the control constraint set is infinite, to implement this approach the constraint set must be replaced by a finite set, obtained by some form of discretization or random sampling, which can be inconvenient and ineffective. In this section we will discuss an alternative approach to deal with an infinite number of controls and Q-factors at x_k. The idea is to *use a base heuristic that involves a continuous optimization*, and to rely on a linear or nonlinear programming method to solve the corresponding lookahead optimization problem.

2.8.1 Rollout for Infinite-Spaces Deterministic Problems

To develop the basic idea of how to deal with infinite control spaces, we first consider deterministic problems, involving a system

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

and a cost per stage $g_k(x_k, u_k)$. The one-step lookahead rollout minimization is

$$\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k), \tag{2.67}$$

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

$$\tilde{Q}_k(x_k, u_k) = g_k(x_k, u_k) + H_{k+1}\big(f_k(x_k, u_k)\big), \tag{2.68}$$

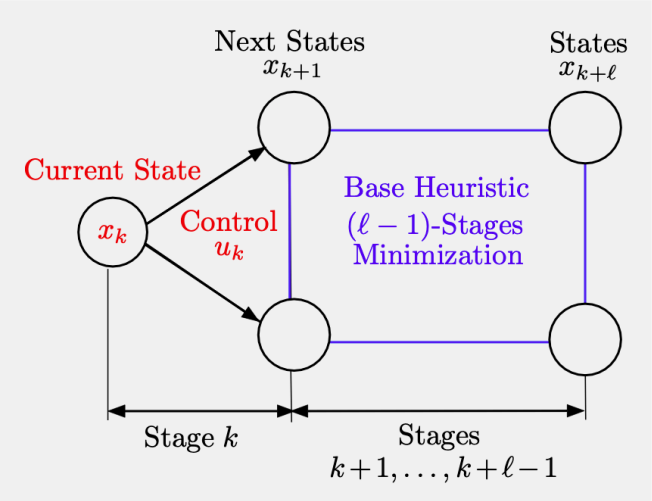


Figure 2.8.1 Schematic illustration of rollout for a deterministic problem with infinite control spaces. The base heuristic is to solve an $(\ell-1)$-stage deterministic optimal control problem, which together with the kth stage minimization over $u_k \in U_k(x_k)$, seamlessly forms an ℓ-stage continuous spaces optimal control/nonlinear programming problem that starts at state x_k.

with $H_{k+1}(x_{k+1})$ being the cost of the base heuristic starting from state x_{k+1} [cf. Eq. (2.12)]. Suppose that we have a differentiable closed-form expression for H_{k+1}, and the functions g_k and f_k are known and are differentiable with respect to u_k. Then the Q-factor $\tilde{Q}_k(x_k,u_k)$ of Eq. (2.68) is also differentiable with respect to u_k, and its minimization (2.67) may be addressed with one of the many gradient-based methods that are available for differentiable unconstrained and constrained optimization.

The preceding approach requires that the heuristic cost $H_{k+1}(x_{k+1})$ can be differentiated, so it should either be available in closed form, which is quite restrictive, or that it can be differentiated numerically, which may be inconvenient and/or unreliable. These difficulties can be circumvented by *using a base heuristic that is itself based on multistep optimization*. In particular, suppose that $H_{k+1}(x_{k+1})$ is the optimal cost of some $(\ell-1)$-stage deterministic optimal control problem that is related to the original problem. Then the rollout algorithm (2.67)-(2.68) can be implemented by solving the *ℓ-stage deterministic optimal control problem, which seamlessly concatenates the first stage minimization over u_k [cf. Eq. (2.67)], with the $(\ell-1)$-stage minimization of the base heuristic*; see Fig. 2.8.1. This ℓ-stage problem may be solvable on-line by standard continuous spaces nonlinear programming or optimal control methods.† A major paradigm of methods of this type is model predictive control, which we have discussed in Chapter

† Note, however, that for this to be possible, it is necessary to have a mathematical model of the system; a simulator is not sufficient. Another difficulty

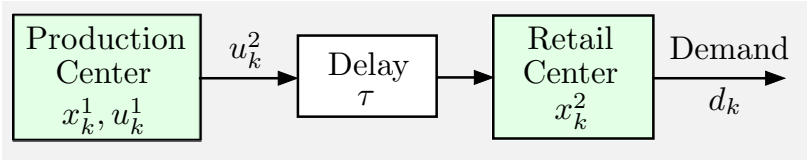


Figure 2.8.2. Illustration of a simple supply chain system for Example 2.8.1.

1 (cf. Section 1.6.7). In the present section we will discuss a few other possibilities. The following is a simple example of an important class of inventory storage and supply chain management processes.

Example 2.8.1 (Supply Chain Management)

Let us consider a supply chain system, where a certain item is produced at a production center and fulfilled at a retail center. Stock of the item is shipped from the production center to the retail center, where it arrives with a delay of $\tau \geq 1$ time units, and is used to fulfill a known stream of demands d_k over an N-stage horizon; see Fig. 2.8.2. We denote:

x_k^1: The stock at hand at the production center at time k.

x_k^2: The stock at hand at the retail center at time k, and used to fulfill demand (both positive and negative x_k^2 are allowed; a negative value indicates that there is backordered demand).

u_k^1: The amount produced at time k.

u_k^2: The amount shipped at time k (and arriving at the retail center τ time units later).

The state at time k is the stock available at the production and retail centers, x_k^1, x_k^2, plus the stock amounts that are in transit and have not yet arrived at the retail center $u_{k-\tau-1}^2, \ldots, u_{k-1}^2$. The control $u_k = (u_k^1, u_k^2)$ is chosen from some constraint set that may depend on the current state, and is subject to production capacity and transport availability constraints. The system equation is

$$x_{k+1}^1 = x_k^1 + u_k^1 - u_k^2, \qquad x_{k+1}^2 = x_k^2 + u_{k-\tau}^2 - d_k,$$

and involves the delayed control component $u_{k-\tau}^2$. Thus the exact DP algorithm involves state augmentation as introduced in Section 1.6.3, and may thus be much more complicated than in the case where there are no delays.†

occurs when the control space is the union of a discrete set and a continuous set. Then it may be necessary to use some type of mixed integer programming technique to solve the ℓ-stage problem. Alternatively, it may be possible to handle the discrete part by brute force enumeration, followed by continuous optimization.

† Despite the fact that with large delays, the size of the augmented state space can become very large (cf. Section 1.6.3), the implementation of rollout schemes is not affected much by this increase in size. For this reason, rollout can be very well suited for problems involving delayed effects of past states and controls.

The cost at time k consists of three components: a production cost that depends on x_k^1 and u_k^1, a transportation cost that depends on u_k^2, and a fulfillment cost that depends on x_k^2 [which includes positive costs for both excess inventory (i.e., $x_k^2 > d_k$) and for backordered demand (i.e., $x_k^2 < d_k$)]. The precise forms of these cost components are immaterial for the purposes of this example.

Here the control vector u_k is often continuous (or a mixture of discrete and continuous components), so it may be essential for the purposes of rollout to use the continuous optimization framework of this section. In particular, at the current stage k, we know the current state, which includes x_k^1, x_k^2, and the amounts of stock in transit together with their scheduled arrival times at the retail center. We then apply some heuristic optimization to determine the stream of future production and shipment levels over ℓ steps, and use the first component of this stream as the control applied by rollout. As an example we may use as base policy one that brings the retail inventory to some target value ℓ stages ahead, and possibly keep it at that value for a portion of the remaining periods. This is a nonlinear programming or mixed integer programming problem that may be solvable with available software far more efficiently than by a discretized form of DP.

A major benefit of rollout in the supply chain context is that it can readily incorporate on-line replanning. This is necessary when unexpected demand changes, production or transport equipment failures occur, or updated forecasts become available.

The following example deals with a common class of problems of resource allocation over time.

Example 2.8.2 (Multistage Linear and Mixed Integer Programming)

Let us consider a deterministic optimal control problem with linear system equation

$$x_{k+1} = A_k x_k + B_k u_k + d_k, \qquad k = 0, \ldots, N-1,$$

where A_k and B_k are known matrices of appropriate dimension, d_k is a known vector, and x_k and u_k are column vectors. The cost function is linear of the form

$$c_N{}' x_N + \sum_{k=0}^{N-1} (c_k{}' x_k + d_k{}' u_k),$$

where c_k and d_k are known column vectors of appropriate dimension, and a prime denotes transpose. The terminal state and state-control pairs (x_k, u_k) are constrained by

$$x_N \in T, \qquad (x_k, u_k) \in P_k, \qquad k = 0, \ldots, N-1,$$

where T and P_k, $k = 0, \ldots, N-1$, are given sets, which are specified by linear and possibly integer constraints.

As an example, consider a multi-item production system, where the state is $x_k = (x_k^1, \ldots, x_k^n)$ and x_k^i represents stock of item i available at the start of period k. The state evolves according to the system equation

$$x_{k+1}^i = \sum_{j=1}^{n} a_k^{ij} u_k^{ij} - d_k^i, \qquad i = 1, \ldots, n,$$

where u_k^{ij} is the amount of product i that is used during time k for the manufacture of product j, a_k^{ij} are known scalars that are related to the underlying production process, and d_k^i is a deterministic demand of product i that is fulfilled at time k. One constraint here is that

$$\sum_{j=1}^{n} u_k^{ij} \leq x_k^i, \qquad i = 1, \ldots, n,$$

and there are additional linear and integer constraints on (x_k, u_k), which are collected in a general constraint of the form $(x_k, u_k) \in P_k$ (e.g., nonnegativity, production capacity, storage constraints, etc). Note that the problem may be further complicated by production delays, as in the preceding supply chain Example 2.8.1. Moreover, while in this section we focus on deterministic problems, we may envision a stochastic version of the problem where the demands d_k^i are random with given probability distributions, which are subject to revisions based on randomly received forecasts.

The problem may be solved using a linear or mixed integer programming algorithm, but this may be very time-consuming when N is large. Moreover, the problem will need to be resolved on-line if some of the problem data changes and replanning is necessary. A suboptimal alternative is to use truncated rollout with an ℓ-stage mixed integer optimization, and a polyhedral terminal cost function $\tilde{J}_{k+\ell}$ to provide a terminal cost optimization. A simple possibility is no terminal cost $[\tilde{J}_{k+\ell}(x_{k+\ell}) \equiv 0]$, and another possibility is a polyhedral lower bound approximation that can be based on relaxing the integer constraints after stage $k + \ell$, or some kind of training approach that uses data.

We will next discuss how rollout can accommodate stochastic disturbances by using deterministic optimization ideas based on certainty equivalence (cf. Section 2.7.4) and the methodology of stochastic programming.

2.8.2 Rollout Based on Stochastic Programming

We have focused so far in this section on rollout that relies on deterministic continuous optimization. There is an important class of methods, known as *stochastic programming*, which can be used for stochastic optimal control, but bears a close connection to continuous spaces deterministic optimization. We will first describe this connection for two-stage problems, then discuss extensions to many-stages problems, and finally show how rollout can be brought to bear for their approximate solution.

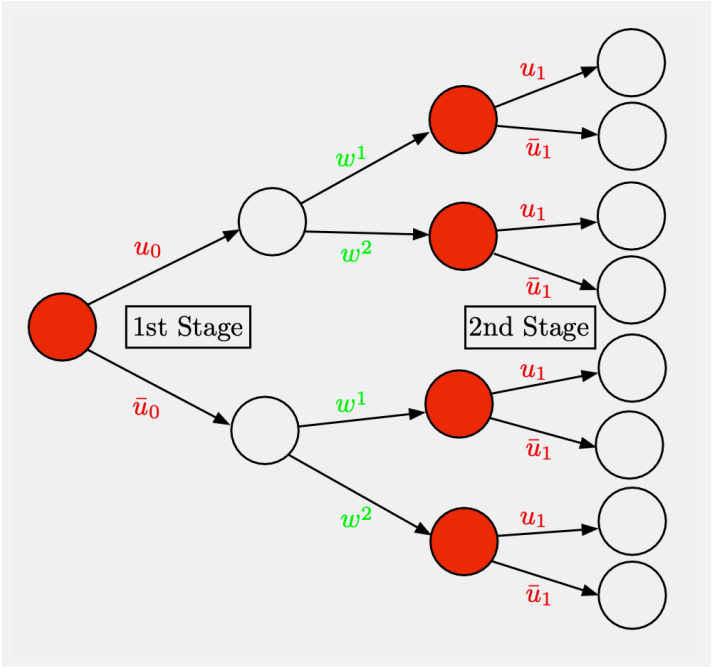


Figure 2.8.3. Illustration of the DP problem associated with two-stage stochastic programming; cf. Example 2.8.3. The figure depicts the case where each variable u_0, w_0, and u_1 can take only two values. A similar conversion to a DP problem is possible for a multistage stochastic programming problem, involving multiple choices of decisions, each followed by an uncertain event whose outcome is perfectly observed by the decision maker.

Example 2.8.3 (Two-Stage Stochastic Programming)

Consider a stochastic problem of optimal decision making over two stages: In the first stage we will choose a finite-dimensional vector u_0 from a subset U_0 with cost $g_0(u_0)$. Then an uncertain event represented by a random variable w_0 will occur, whereby w_0 will take one of the values $w^1, \ldots, w^m$ with corresponding probabilities $p^1, \ldots, p^m$. Once w_0 occurs, we will know its value w^i, and we must then choose at the second stage a vector u_1^i from a subset $U_1(u_0, w^i)$ at a cost $g_1(u_1^i, w^i)$. The objective is to minimize the expected cost

$$g_0(u_0) + \sum_{i=1}^{m} p^i g_1(u_1^i, w^i),$$

subject to

$$u_0 \in U_0, \qquad u_1^i \in U_1(u_0, w^i), \quad i = 1, \ldots, m.$$

We can view this problem as a two-stage DP problem, where $x_1 = w_0$ is the system equation, the disturbance w_0 can take the values $w^1, \ldots, w^m$ with

probabilities $p^1, \ldots, p^m$, the cost of the first stage is $g_0(u_0)$, the cost of the second stage is $g_1(x_1, u_1)$, and the terminal cost is 0. The intuitive meaning is that since at time 0 we don't know yet which of the m values w^i of w_0 will occur, we must calculate (in addition to u_0) a separate second stage decision u_1^i for each i, which will be used after we know that the value of w_0 is w^i.

However, if u_0 and u_1 take values in a continuous space such as the Euclidean spaces $\Re^{d_0}$ and $\Re^{d_1}$, respectively, we can also equivalently view the problem as a nonlinear programming problem of dimension $(d_0 + md_1)$ (the optimization variables are u_0 and u_1^i, $i = 1, \ldots, m$).

For a generalization of the preceding example, consider the stochastic DP problem of Section 1.3 for the case where there are only two stages, and the disturbances w_0 and w_1 can independently take one of the m values $w^1, \ldots, w^m$ with corresponding probabilities $p_0^1, \ldots, p_0^m$ and $p_1^1, \ldots, p_1^m$, respectively. The optimal cost function $J_0(x_0)$ is given by the two-stage DP algorithm

$$J_0(x_0) = \min_{u_0 \in U_0(x_0)} \left[\sum_{i=1}^m p_0^i \left\{ g_0(x_0, u_0, w^i) + \min_{u_1^i \in U_1(f_0(x_0,u_0,w^i))} \left[\sum_{j=1}^m p_1^j \left\{ g_1\big(f_0(x_0, u_0, w^i), u_1^i, w^j\big) + g_2\big(f_1\big(f_0(x_0, u_0, w^i), u_1^i, w^j\big)\big) \right\} \right] \right\} \right].$$

By bringing the inner minimization outside the inner brackets, we see that this DP algorithm is equivalent to solving the nonlinear programming problem

$$\begin{aligned} \text{minimize } & \sum_{i=1}^m p_0^i \left\{ g_0(x_0, u_0, w^i) + \sum_{j=1}^m p_1^j \left\{ g_1\big(f_0(x_0, u_0, w^i), u_1^i, w^j\big) + g_2\big(f_1\big(f_0(x_0, u_0, w^i), u_1^i, w^j\big)\big) \right\} \right\} \\ \text{subject to } & u_0 \in U_0(x_0), \qquad u_1^i \in U_1\big(f_0(x_0, u_0, w^i)\big), \quad i = 1, \ldots, m. \end{aligned} \tag{2.69}$$

If the controls u_0 and u_1^i are elements of $\Re^d$, this problem involves

$$d(1 + m)$$

scalar variables.

We can also consider an N-stage stochastic optimal control problem. A similar reformulation as a nonlinear programming problem is possible. It converts the N-stage stochastic problem into a deterministic optimization

problem of dimension that grows exponentially with the number of stages N. In particular, for an N-stage problem, the number of control variables expands by a factor m with each additional stage. The total number of variables is bounded by

$$d\big(1 + m + m^2 + \cdots + m^{N-1}\big),$$

where m is the maximum number of values that a disturbance can take at each stage and d is the dimension of the control vector. An example is the multi-item production problem described in Example 2.8.2 in the case where the demands w_k^i and/or the production coefficients a_k^{ij} are stochastic.

2.8.3 Stochastic Rollout with Certainty Equivalence

The dimension of the preceding nonlinear programming formulation of the multistage stochastic optimal control problem with continuous control spaces can be very large. This motivates a variant of a rollout algorithm that relies on a stochastic optimization for the current stage, and a deterministic optimization that relies on (assumed) certainty equivalence for the remaining stages, where the base policy is used. In this way, the dimension of the nonlinear programming problem to be solved by rollout is drastically reduced.

This rollout algorithm operates as follows: Given a state x_k and control $u_k \in U_k(x_k)$, we consider the next states x_{k+1}^i that correspond to the m possible values w_k^i, $i = 1, \ldots, m$, which occur with the known probabilities p_k^i, $i = 1, \ldots, m$. We then consider the approximate Q-factors

$$\tilde{Q}_k(x_k, u_k) = \sum_{i=1}^{m} p_k^i \big(g_k(x_k, u_k, w_k^i) + \tilde{H}_{k+1}(x_{k+1}^i)\big), \tag{2.70}$$

where $\tilde{H}_{k+1}(x_{k+1}^i)$ is the cost of a base policy, which starting at stage $k+1$ from

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

optimizes the cost-to-go starting from x_{k+1}^i, while assuming that the future disturbances $w_{k+1}, \ldots, w_{N-1}$, will take some nominal (nonrandom) values $\bar{w}_{k+1}, \ldots, \bar{w}_{N-1}$. The rollout control $\tilde{\mu}_k(x_k)$ computed by this algorithm is

$$\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k). \tag{2.71}$$

Note that this rollout algorithm does not have the cost improvement property, because it involves an approximation: the cost $\tilde{H}_{k+1}(x_{k+1}^i)$ used in Eq. (2.70) is an approximation to the cost of a policy. It is the cost of a policy applied to the certainty equivalent version of the original stochastic problem.

The key fact now is that the problem (2.71) can be viewed as a seamless $(N-k)$-stage deterministic optimization, which involves the control u_0, and for each value w_k^i of the disturbance w_k, the sequence of controls $(u_{k+1}^i, \ldots, u_{N-1}^i)$. If the controls are elements of $\Re^d$, this deterministic optimization involves a total of

$$d\big(1 + (N-k-1)m\big) \tag{2.72}$$

scalar variables. Currently available deterministic optimization software can deal with quite large numbers of variables, particularly in the context of linear programming, so by using rollout in combination with certainty equivalence, very large problems with continuous state and control variables may be addressed.

Another possibility is to use multistep lookahead that aims to represent better the stochastic character of the uncertainty. Here at state x_k we solve an $(N-k)$-stage optimal control problem, where the uncertainty is fully taken into account in the first ℓ stages, similar to stochastic programming, and in the remaining $N-k-\ell$ stages, the uncertainty is dealt with by certainty equivalence, by fixing the disturbances $w_{k+\ell}, \ldots, w_{N-1}$ at some nominal values (we assume here for simplicity that $\ell < N-k$). If the controls are elements of $\Re^d$, and the number of values that the disturbances $w_0, \ldots, w_{N-1}$ can take is m, the total number of control variables of this problem is

$$d\big(1 + m + \cdots + m^{\ell-1} + (N-k-\ell)m^\ell\big),$$

[this is the ℓ-step lookahead generalization of the formula (2.72)]. Once the optimal policy $\{\tilde{u}_k, \tilde{\mu}_{k+1}, \tilde{\mu}_{k+2}, \ldots\}$ for this problem is obtained, the first control component $\tilde{u}_k$ is applied at x_k and the remaining components $\{\tilde{\mu}_{k+1}, \tilde{\mu}_{k+2}, \ldots\}$ are discarded. Note also that this multistep lookahead approach may be combined with the ideas of multiagent rollout, which will be discussed in the next section.

2.9 MULTIAGENT ROLLOUT

We will now consider a special structure of the control space, whereby the control u_k consists of m components, $u_k = (u_k^1, \ldots, u_k^m)$, with a separable control constraint structure $u_k^\ell \in U_k^\ell(x_k)$, $\ell = 1, \ldots, m$. The control constraint set is the Cartesian product

$$U_k(x_k) = U_k^1(x_k) \times \cdots \times U_k^m(x_k). \tag{2.73}$$

Conceptually, each component u_k^ℓ, $\ell = 1, \ldots, m$, is chosen at stage k by a separate "agent" (a decision making entity), and for the sake of the

following discussion, we assume that each set $U_k^\ell(x_k)$ is finite. We discussed this type of problem briefly in Section 1.6.5, and we will discuss it in this section in greater detail.

Thus the one-step lookahead minimization

$$\tilde{u}_k \in \arg\min_{u_k \in U_k(x_k)} E\Big\{g_k(x_k, u_k, w_k) + J_{k+1,\pi}\big(f_k(x_k, u_k, w_k)\big)\Big\}, \quad (2.74)$$

where π is a base policy, involves as many as n^m Q-factors, where n is the maximum number of elements of the sets $U_k^\ell(x_k)$ [so that n^m is an upper bound to the number of controls in $U_k(x_k)$, in view of the Cartesian product structure (2.73)]. As a result, the standard rollout algorithm requires an exponential [order $O(n^m)$] number of base policy cost computations per stage, which can be overwhelming even for moderate values of m.

This motivates an alternative and more efficient rollout algorithm, called *multiagent rollout* also referred to as *agent-by-agent rollout*, that still achieves the cost improvement property

$$J_{k,\tilde{\pi}}(x_k) \leq J_{k,\pi}(x_k), \qquad \forall\ x_k, k, \quad (2.75)$$

where $J_{k,\tilde{\pi}}(x_k)$, $k = 0, \ldots, N$, is the cost-to-go of the rollout policy $\tilde{\pi}$ starting from state x_k. Indeed we will exploit the multiagent structure to construct an algorithm that maintains the cost improvement property at much smaller computational cost, namely requiring order $O(nm)$ base policy cost computations per stage.

A key idea here is that the computational requirements of the rollout one-step minimization (2.74) are proportional to the size of the control space and are independent of the size of the state space. We consequently reformulate the problem so that control space complexity is traded off with state space complexity, as discussed in Section 1.6.5. This is done by "unfolding" the control u_k into its m components $u_k^1, u_k^2, \ldots, u_k^m$. At the same time, between x_k and the next state $x_{k+1} = f_k(x_k, u_k, w_k)$, we introduce artificial intermediate "states" and corresponding transitions; see Fig. 2.9.1.

It can be seen 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, each policy of the reformulated problem corresponds to a policy of the original problem, with the same cost function, and reversely.†

† Policies of the original problem involve functions of x_k, while policies of the reformulated problem involve functions of the choices of the preceding agents, as well as x_k. However, by successive substitution of the control functions of the preceding agents, we can view control functions of each agent as depending exclusively on x_k. It follows that the multi-transition structure of the reformulated problem cannot be exploited to reduce the cost function beyond what can be achieved with a single-transition structure.

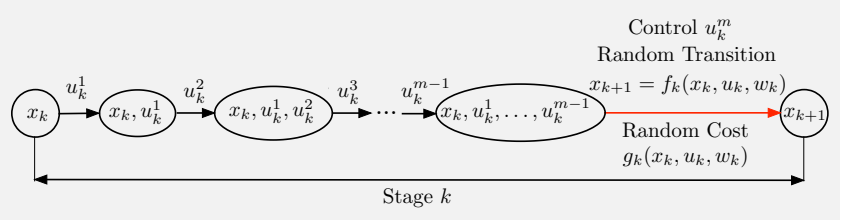


Figure 2.9.1 Equivalent formulation of the N-stage stochastic optimal control problem for the case where the control u_k consists of m components $u_k^1, u_k^2, \ldots, u_k^m$:

$$u_k = (u_k^1, \ldots, u_k^m) \in U_k^1(x_k) \times \cdots \times U_k^m(x_k).$$

The figure depicts the kth stage transitions. Starting from state x_k, we generate the intermediate states

$$(x_k, u_k^1), (x_k, u_k^1, u_k^2), \ldots, (x_k, u_k^1, \ldots, u_k^{m-1}),$$

using the respective controls $u_k^1, \ldots, u_k^{m-1}$. The final control u_k^m leads from $(x_k, u_k^1, \ldots, u_k^{m-1})$ to $x_{k+1} = f_k(x_k, u_k, w_k)$, and a stage cost $g_k(x_k, u_k, w_k)$ is incurred.

Multiagent Rollout

Consider now the standard rollout algorithm applied to the reformulated problem of Fig. 2.9.1, with a given base policy $\pi = \{\mu_0, \ldots, \mu_{N-1}\}$, which is also a policy of the original problem [so that $\mu_k = (\mu_k^1, \ldots, \mu_k^m)$, with each μ_k^ℓ, $\ell = 1, \ldots, m$, being a function of just x_k]. The algorithm involves a minimization over only one control component at the states x_k and at the intermediate states

$$(x_k, u_k^1), (x_k, u_k^1, u_k^2), \ldots, (x_k, u_k^1, \ldots, u_k^{m-1}).$$

In particular, *for each stage k, the algorithm requires a sequence of m minimizations, once over each of the agent controls $u_k^1, \ldots, u_k^m$, with the past controls determined by the rollout policy, and the future controls determined by the base policy.* Assuming a maximum of n elements in the constraint sets $U_k^\ell(x_k)$, the computation required at each stage k is of order $O(n)$ for each of the "states"

$$x_k, (x_k, u_k^1), \ldots, (x_k, u_k^1, \ldots, u_k^{m-1}),$$

for a total of order $O(nm)$ computation.

To elaborate, at $(x_k, u_k^1, \ldots, u_k^{\ell-1})$ with $\ell \leq m$, and for each of the controls $u_k^\ell \in U_k^\ell(x_k)$, we generate by simulation a number of system trajectories up to stage N, with all future controls determined by the base policy. We average the costs of these trajectories, thereby obtaining the

Q-factors corresponding to $(x_k, u_k^1, \ldots, u_k^{\ell-1}, u_k^\ell)$, for all values $u_k^\ell \in U_k^\ell(x_k)$ (with the preceding controls $u_k^1, \ldots, u_k^{\ell-1}$ held at the values computed earlier, and the future controls $u_k^{\ell+1}, \ldots, u_k^m, u_{k+1}, \ldots, u_{N-1}$ determined by the base policy). We then select the control $u_k^\ell \in U_k^\ell(x_k)$ that corresponds to the minimal Q-factor.

Prerequisite assumptions for the preceding algorithm to work in an on-line multiagent setting are:

(a) All agents have access to the current state x_k as well as the base policy (including the control functions μ_n^ℓ, $\ell = 1, \ldots, m$, $n = 0, \ldots, N-1$ of all agents).

(b) There is an order in which agents compute and apply their local controls.

(c) The agents share their information, so agent ℓ knows the local controls $u_k^1, \ldots, u_k^{\ell-1}$ computed by the predecessor agents $1, \ldots, \ell-1$ in the given order.

Note that the rollout policy obtained from the reformulated problem may be different from the rollout policy obtained from the original problem. However, the former rollout algorithm is far more efficient than the latter in terms of required computation, while still maintaining the cost improvement property (2.75).

The following spiders-and-flies example illustrates how multiagent rollout may exhibit intelligence and agent coordination that is totally lacking from the base policy. This behavior has been supported by computational experiments and analysis with larger (two-dimensional) spiders-and-flies problems.

Example 2.9.1 (Spiders and Flies)

We have two spiders and two flies moving along integer locations on a straight line. For simplicity we assume that the flies' positions are fixed at some integer locations, although the problem is qualitatively similar when the flies move randomly. The spiders have the option of moving either left or right by one unit; see Fig. 2.9.2. The objective is to minimize the time to capture both flies. The problem has essentially a finite horizon since the spiders can force the capture of the flies within a known number of steps.

The salient feature of the optimal policy here is to move the two spiders towards different flies. The minimal time to capture is the maximum of the initial distances of the two spider-fly pairs of the optimal policy.

Let us apply multiagent rollout with the base policy that directs each spider to move one unit towards the closest fly position (a tie is broken by moving towards the right-side fly). The base policy is poor because it may unnecessarily move both spiders in the same direction, when in fact only one is needed to capture the fly. This limitation is due to the lack of coordination between the spiders: each acts selfishly, ignoring the presence of the other. We will see that rollout restores a significant degree of coordination between

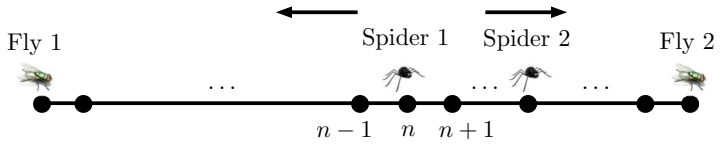


Figure 2.9.2 Illustration of the two-spiders and two-flies problem. The spiders move along integer points of a line. The two flies stay still at some integer locations. The character of the optimal policy is to move the two spiders towards two different flies.

Multiagent rollout with the given base policy starts with spider 1 at location n, and calculates the two Q-factors of moving to locations $n-1$ and $n+1$, assuming that the remaining moves of the two spiders will be made using the go-towards-the-nearest-fly base policy. The Q-factor of going to $n-1$ is smallest because it saves in unnecessary moves of spider 1 towards fly 2, so spider 1 will move towards fly 1. The trajectory generated by multiagent rollout is to move spiders 1 and 2 towards flies 1 and 2, respectively, then spider 2 first captures fly 2, and then spider 1 captures fly 1.

the spiders through an optimization that takes into account the long-term consequences of the spider moves.

According to the multiagent rollout mechanism, the spiders choose their moves one-at-a-time, optimizing over the two Q-factors corresponding to the right and left moves, while assuming that future moves will be chosen according to the base policy. Let us consider a stage, where the two flies are alive, while both spiders are closest to fly 2, as in Fig. 2.9.2. Then the rollout algorithm will start with spider 1 and calculate two Q-factors corresponding to the right and left moves, while using the base heuristic to obtain the next move of spider 2, and the remaining moves of the two spiders. Depending on the values of the two Q-factors, spider 1 will move to the right or to the left, and it can be seen that it will choose to *move away from spider 2* even if doing so increases its distance to its closest fly *contrary to what the base heuristic will do*. Then spider 2 will act similarly and the process will continue. Intuitively, at the state of Fig. 2.9.2, spider 1 moves away from spider 2 and fly 2, because it recognizes that spider 2 will capture earlier fly 2, so it might as well move towards the other fly.

Thus *the multiagent rollout algorithm induces implicit move coordination*, i.e., each spider moves in a way that takes into account future moves of the other spider. In fact it can be verified that the algorithm will produce an optimal sequence of moves starting from any initial spider positions. It can also be seen that ordinary rollout (both flies move at once) will also produce an optimal move sequence.

The example illustrates how a poor base heuristic can produce an excellent rollout solution, something that can be observed frequently in many other problems. Intuitively, the key fact is that rollout is "farsighted" in the sense that it can benefit from control calculations that reach far into future stages.

A two-dimensional generalization of the example is also interesting. Here the flies are at two corners of a square in the plane. It can be shown that the two spiders, starting from the same position within the square, will

separate under the rollout policy, with each moving towards a different spider, while under the base policy, they will move in unison along the shortest path to the closest surviving fly. Again this will happen for both standard and multiagent rollout.

Example 2.9.2 (Multi-Vehicle Routing)

Consider the multi-vehicle routing problem of Example 1.2.3, whereby m vehicles move along the arcs of a given graph, aiming to perform tasks located at the nodes of the graph; cf. Fig. 2.9.3.

For a large number of vehicles and a complicated graph, this is a non-trivial combinatorial problem. As we discussed in Example 1.2.3, the problem can be formulated as a discrete deterministic optimization problem, and addressed by approximate DP methods. The state at a given stage is the m-tuple of current positions of the vehicles together with the list of pending tasks, but the number of these states can be enormous (it increases exponentially with the number of nodes and the number of vehicles). Moreover the number of joint move choices by the vehicles also increases exponentially with the number of vehicles.

We are thus motivated to use a multiagent rollout approach. We define a base heuristic as follows: at a given stage and state (vehicle positions and pending tasks), it finds the closest pending task (in terms of number of moves needed to reach it) for each of the vehicles and moves each vehicle one step towards the corresponding closest pending task (this is a legitimate base heuristic: it assigns to each state a vehicle move for every vehicle).†

In the multiagent rollout algorithm, at a given stage and state, we take up each vehicle in the order $1, \ldots, n$, and we compare the Q-factors of the available moves to that vehicle while assuming that all the remaining moves will be made according to the base heuristic, and taking into account the moves that have been already made and the tasks that have already been performed; see the illustration of Fig. 2.9.3. In contrast to all-vehicles-at-once rollout, the one-vehicle-at-a-time rollout algorithm considers a polynomial (in m) number of moves and corresponding shortest path problems at each stage. In the example of Fig. 2.9.3, the one-vehicle-at-a-time rollout finds the optimal solution, while the base heuristic starting from the initial state does not.

† There is an alternative version of the base heuristic, which makes selections one-vehicle-at-a-time: at a given stage and state (vehicle positions and pending tasks), it finds the closest pending task (in terms of number of moves needed to reach it) for vehicle 1 and moves this vehicle one step towards this closest pending task. Then it finds the closest pending task for vehicle 2 (the pending status of the tasks, however, may have been affected by the move of vehicle 1) and moves this vehicle one step towards this closest pending task, and continues similarly for vehicles $3, \ldots, n$. There is a subtle difference between the two base heuristics: for example they may make different choices when vehicle 1 reaches a pending task in a single move, thereby changing the status of that task, and affecting the choice of the base heuristic for vehicle 2, etc.

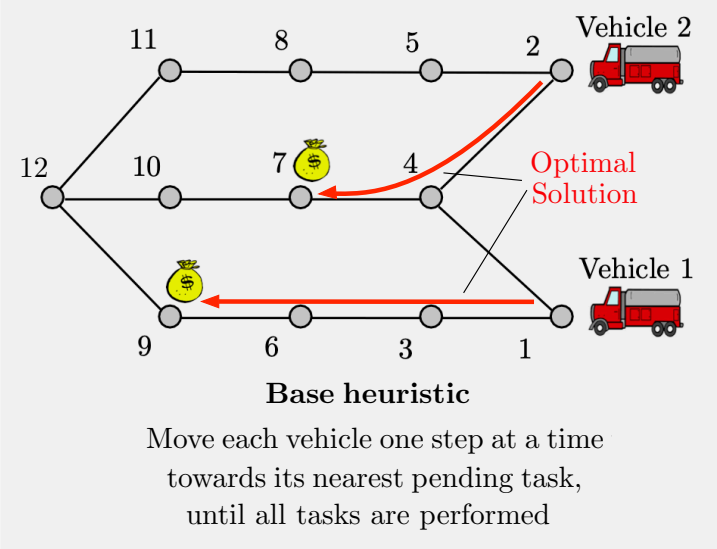


Figure 2.9.3 An instance of the vehicle routing problem of Example 2.9.2, and the multiagent rollout approach. The two vehicles aim to collectively perform the two tasks as fast as possible. Here, we should avoid sending both vehicles to node 4, towards the task at node 7; sending only vehicle 2 towards that task, while sending vehicle 1 towards the task at node 9 is clearly optimal. However, the base heuristic has "limited vision" and does not perceive this. By contrast the standard and the one-vehicle-at-a-time rollout algorithms look beyond the first move and avoid this inefficiency: they examine both moves of vehicle 1 to nodes 3 and 4, and use the base heuristic to explore the corresponding trajectories to the end of the horizon, and discover that vehicle 2 can reach quickly node 7, and that it is best to send vehicle 1 towards node 9.

In particular, the one-vehicle-at-a-time rollout algorithm will operate as follows: given the starting position pair $(1, 2)$ of the vehicles and the current pending tasks at nodes 7 and 9, we first compare the Q-factors of the two possible moves of vehicle 1 (to nodes 3 and 4), assuming that all the remaining moves will be selected by the base heuristic at the beginning of each stage. Thus vehicle 1 will choose to move to node 3. Then with knowledge of the move of vehicle 1 from 1 to 3, we select the move of vehicle 2 by comparing the Q-factors of its two possible moves (to nodes 4 and 5), taking also into account the fact that the remaining moves will be made according to the base heuristic. Thus vehicle 2 will choose to move to node 4.

We then continue at the next state [vehicle positions at (3,4) and pending tasks at nodes 7 and 9], select the base heuristic moves of vehicles 1 and 2 on the path to the closest pending tasks [(9 and 7), respectively], etc. Eventually the rollout finds the optimal solution (move vehicle 1 to node 9 in three moves and move vehicle 2 to node 7 in two moves), which has a total cost of 5. By contrast it can be seen that the base heuristic at the initial state will move both vehicles to node 4 (towards the closest pending task), and generate a trajectory that moves vehicle 1 along the path $1 \to 4 \to 7$ and vehicle 2 along the path $2 \to 4 \to 7 \to 10 \to 12 \to 9$, while incurring a total cost of 7.

The Cost Improvement Property

Generally, it is unclear how the two rollout policies (standard/all-agents-at-once and agent-by-agent) perform relative to each other in terms of attained cost.† On the other hand, both rollout policies perform no worse than the base policy, since the performance of the base policy is identical for both the reformulated and the original problems. This cost improvement property can also be shown analytically as follows by induction, by modifying the standard rollout cost improvement proof; cf. Section 2.7.

Proposition 2.9.1: (Cost Improvement for Multiagent Rollout) The rollout policy $\tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$ obtained by multiagent rollout satisfies

$$J_{k,\tilde{\pi}}(x_k) \leq J_{k,\pi}(x_k), \qquad \text{for all } x_k \text{ and } k, \tag{2.76}$$

where π is the base policy.

Proof: We will show the inequality (2.76) by induction, but for simplicity, we will give the proof for the case of just two agents, i.e., $m = 2$. Clearly the inequality holds for $k = N$, since $J_{N,\tilde{\pi}} = J_{N,\pi} = g_N$. Assuming that it holds for index $k+1$, we have for all x_k,

$$J_{k,\tilde{\pi}}(x_k) = E\Big\{ g_k\big(x_k, \tilde{\mu}_k^1(x_k), \tilde{\mu}_k^2(x_k), w_k\big) + J_{k+1,\tilde{\pi}}\Big(f_k\big(x_k, \tilde{\mu}_k^1(x_k), \tilde{\mu}_k^2(x_k), w_k\big)\Big)\Big\}$$

† For an example where the standard rollout algorithm works better, consider a single-stage problem, where the objective is to minimize the first stage cost $g_0(u_0^1, \ldots, u_0^m)$. Let $\overline{u}_0 = (\overline{u}_0^1, \ldots, \overline{u}_0^m)$ be the control applied by the base policy, and assume that $\overline{u}_0$ is not optimal. Suppose that starting at $\overline{u}_0$, the cost cannot be improved by varying any single control component. Then the multiagent rollout algorithm stays at the suboptimal $\overline{u}_0$, while the standard rollout algorithm finds an optimal control. Thus, for one-stage problems, the standard rollout algorithm will perform no worse than the multiagent rollout algorithm.

The example just given is best seen within the framework of the classical coordinate descent method for minimizing a function of m components. This method can get stuck at a nonoptimal point in the absence of appropriate conditions on the cost function, such as differentiability and/or convexity. However, within our context of multistage rollout and possibly stochastic disturbances, it appears that the consequences of such a phenomenon may not be serious. In fact, one can construct multi-stage examples where multiagent rollout performs better than the standard rollout.

$$
\begin{aligned}
&\leq E\Big\{g_k\big(x_k,\tilde{\mu}_k^1(x_k),\tilde{\mu}_k^2(x_k),w_k\big)\\
&\qquad\qquad + J_{k+1,\pi}\Big(f_k\big(x_k,\tilde{\mu}_k^1(x_k),\tilde{\mu}_k^2(x_k),w_k\big)\Big)\Big\}\\
&= \min_{u_k^2\in U_k^2(x_k)} E\Big\{g_k(x_k,\tilde{\mu}_k^1(x_k),u_k^2,w_k)\\
&\qquad\qquad + J_{k+1,\pi}\Big(f_k\big(x_k,\tilde{\mu}_k^1(x_k),u_k^2,w_k\big)\Big)\Big\}\\
&\leq E\Big\{g_k\big(x_k,\tilde{\mu}_k^1(x_k),\mu_k^2(x_k),w_k\big)\\
&\qquad\qquad + J_{k+1,\pi}\Big(f_k\big(x_k,\tilde{\mu}_k^1(x_k),\mu_k^2(x_k),w_k\big)\Big)\Big\}\\
&= \min_{u_k^1\in U_k^1(x_k)} E\Big\{g_k(x_k,u_k^1,\mu_k^2(x_k),w_k)\\
&\qquad\qquad + J_{k+1,\pi}\Big(f_k\big(x_k,u_k^1,\mu_k^2(x_k),w_k\big)\Big)\Big\}\\
&\leq E\Big\{g_k\big(x_k,\mu_k^1(x_k),\mu_k^2(x_k),w_k\big)\\
&\qquad\qquad + J_{k+1,\pi}\Big(f_k\big(x_k,\mu_k^1(x_k),\mu_k^2(x_k),w_k\big)\Big)\Big\}\\
&= J_{k,\pi}(x_k),
\end{aligned}
$$

where:

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

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

(c) The second equality holds by the definition of the rollout algorithm as it pertains to agent 2.

(d) The third equality holds by the definition of the rollout algorithm as it pertains to agent 1.

(e) The fourth equality is the DP equation for the base policy π.

The induction proof of the cost improvement property (2.76) is thus complete for the case $m = 2$. The proof for an arbitrary number of agents m is entirely similar. **Q.E.D.**

Optimizing the Agent Order in Agent-by-Agent Rollout - Multiagent Parallelization

In the multiagent rollout algorithm described so far, the agents optimize the control components sequentially in a fixed order. It is possible to improve performance by trying to optimize at each stage k the order of the agents.

An efficient way to do this is to first optimize over all single agent Q-factors, by solving the m minimization problems that correspond to each of the agents $\ell = 1,\ldots,m$ being first in the multiagent rollout order. If ℓ_1 is

the agent that produces the minimal Q-factor, we fix ℓ_1 to be the first agent in the multiagent rollout order. Then we optimize over all single agent Q-factors, by solving the $m-1$ minimization problems that correspond to each of the agents $\ell \neq \ell_1$ being second in the multiagent rollout order. Let ℓ_2 be the agent that produces the minimal Q-factor, fix ℓ_2 to be the second agent in the multiagent rollout order, and continue in this manner. In the end, after

$$m + (m-1) + \cdots + 1 = \frac{m(m+1)}{2} \tag{2.77}$$

minimizations, we obtain an agent order $\ell_1, \ldots, \ell_m$ that produces a potentially much reduced Q-factor value, as well as the corresponding rollout control component selections.

The method just described likely produces substantially better performance, and eliminates the need for guessing a good agent order, but it increases the number of Q-factor calculations needed per stage roughly by a factor $(m+1)/2$. Still this is much better than the all-agents-at-once approach, which requires an exponential number of Q-factor calculations. Moreover, the Q-factor minimizations of the above process can be parallelized, so with m parallel processors, we can perform the number of $m(m+1)/2$ minimizations derived above in just m batches of parallel minimizations, which require about the same time as in the case where the agents are selected for Q-factor minimization in a fixed order. We finally note that our earlier cost improvement proof goes through again by induction, when the order of agent selection is variable at each stage k.

Multiagent Rollout Variants

The agent-by-agent rollout algorithm admits several variants. We describe briefly a few of these variants.

(a) We may use rollout with multistep lookahead, truncated rollout, and terminal cost function approximation, as described earlier. Of course, in such cases the cost improvement property need not hold.

(b) When the control constraint sets $U_k^\ell(x_k)$ are infinite, multiagent rollout still applies, based on the tradeoff between control and state space complexity, cf. Fig. 2.9.1. In particular, when the sets $U_k^\ell(x_k)$ are intervals of the real line, each agent's lookahead minimization problem can be performed with the aid of one-dimensional search methods.

(c) When the problem is deterministic there are additional possible variants of the multiagent rollout algorithm. In particular, for deterministic problems, we may use a more general base policy, i.e., a heuristic that is not defined by an underlying policy; cf. Section 2.3.1. In this case, if the sequential improvement assumption for the modified problem of Fig. 2.9.1 is not satisfied, then the cost improvement property

may not hold. However, cost improvement may be restored by introducing fortification, as discussed in Section 2.3.2.

(d) The multiagent rollout algorithm can be simply modified to apply to infinite horizon problems. In this context, we may also consider policy iteration methods, which can be viewed as repeated rollout. These methods may involve agent-by-agent policy improvement, and value and policy approximations of intermediately generated policies (see the RL book [Ber19a], Section 5.7.3).

(e) The multiagent rollout algorithm can be simply modified to apply to deterministic continuous-time optimal control problems; cf. Section 2.6. The idea is again to simplify the minimization over $u(t)$ in the case where $u(t)$ consists of multiple components $u^1(t), \ldots, u^m(t)$.

(f) We can implement within the agent-by-agent rollout context the use of Q-factor differences. The motivation is similar: deal with the approximation errors that are inherent in the estimated cost of the base policy, $\tilde{J}_{k+1,\pi}\big(f_k(x_k,u_k)\big)$, and may overwhelm the current stage cost term $g_k(x_k,u_k)$. As noted in Section 2.3.7, this may seriously degrade the quality of the rollout policy; see also the discussion of advantage updating and differential training in Chapter 3.

Constrained Multiagent Rollout

Let us consider a special structure of the control space, where the control u_k consists of m components, $u_k = (u_k^1, \ldots, u_k^m)$, each belonging to a corresponding set $U_k^\ell(x_k)$, $\ell = 1, \ldots, m$. Thus the control space at stage k is the Cartesian product

$$U_k(x_k) = U_k^1(x_k) \times \cdots \times U_k^m(x_k).$$

We refer to this as the *multiagent case*, motivated by the special case where each component u_k^ℓ, $\ell = 1, \ldots, m$, is chosen by a separate agent ℓ at stage k.

Similar to the unconstrained case, we can introduce a modified but equivalent problem, involving one-at-a-time agent control selection. In particular, at the generic state x_k, we break down the control u_k into the sequence of the m controls $u_k^1, u_k^2, \ldots, u_k^m$, and between x_k and the next state $x_{k+1} = f_k(x_k,u_k)$, we introduce artificial intermediate "states"

$$(x_k, u_k^1), (x_k, u_k^1, u_k^2), \ldots, (x_k, u_k^1, \ldots, u_k^{m-1}),$$

and corresponding transitions. The choice of the last control component u_k^m at "state" $(x_k, u_k^1, \ldots, u_k^{m-1})$ marks the transition at cost $g_k(x_k,u_k)$ to the next state $x_{k+1} = f_k(x_k,u_k)$ according to the system equation. 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, with the same cost.

By working with the reformulated problem, we can consider a rollout algorithm that requires a sequence of m minimizations per stage, one over each of the control components $u_k^1, \ldots, u_k^m$, with the past controls already determined by the rollout algorithm, and the future controls determined by running the base heuristic. Assuming a maximum of n elements in the control component spaces $U_k^\ell(x_k)$, $\ell = 1, \ldots, m$, the computation required for the m single control component minimizations is of order $O(nm)$ per stage. By contrast the standard rollout minimization (2.34) involves the computation and comparison of as many as n^m terms $G\big(T_k(\tilde{y}_k, u_k)\big)$ per stage.

2.9.1 Asynchronous and Autonomous Multiagent Rollout

In this section we consider multiagent rollout algorithms that are distributed and asynchronous in the sense that the agents may compute their rollout controls in parallel rather than in sequence, aiming at computational speedup. An example of such an algorithm is obtained when at a given stage, agent ℓ computes the rollout control $\tilde{u}_k^\ell$ before knowing the rollout controls of some of the agents $1, \ldots, \ell - 1$, and uses the controls $\mu_k^1(x_k), \ldots, \mu_k^{\ell-1}(x_k)$ of the base policy in their place.

This algorithm may work well for some problems, but it does not possess the cost improvement property, and may not work well for other problems. In fact we can construct a simple example involving a single state, two agents, and two controls per agent, where the second agent does not take into account the control applied by the first agent, and as a result the rollout policy performs worse than the base policy for some initial states.

Example 2.9.3 (Cost Deterioration in the Absence of Adequate Agent Coordination)

Consider a problem with two agents ($m = 2$) and a single state. Thus the state does not change and the costs of different stages are decoupled (the problem is essentially static). Each of the two agents has two controls: $u_k^1 \in \{0, 1\}$ and $u_k^2 \in \{0, 1\}$. The cost per stage g_k is equal to 0 if $u_k^1 \neq u_k^2$, is equal to 1 if $u_k^1 = u_k^2 = 0$, and is equal to 2 if $u_k^1 = u_k^2 = 1$. Suppose that the base policy applies $u_k^1 = u_k^2 = 0$. Then it can be seen that when executing rollout, the first agent applies $u_k^1 = 1$, and in the absence of knowledge of this choice, the second agent also applies $u_k^2 = 1$ (thinking that the first agent will use the base policy control $u_k^1 = 0$). Thus the cost of the rollout policy is 2 per stage, while the cost of the base policy is 1 per stage. By contrast the rollout algorithm that takes into account the first agent's control when selecting the second agent's control applies $u_k^1 = 1$ and $u_k^2 = 0$, thus resulting in a rollout policy with the optimal cost of 0 per stage.

The difficulty here is inadequate coordination between the two agents. In particular, each agent uses rollout to compute the local control, each thinking that the other will use the base policy control. If instead the two agents were to coordinate their control choices, they would have applied an optimal policy.

The simplicity of the preceding example raises serious questions as to whether the cost improvement property (2.76) can be easily maintained by a distributed rollout algorithm where the agents do not know the controls applied by the preceding agents in the given order of local control selection, and use instead the controls of the base policy. One may speculate that if the agents are naturally "weakly coupled" in the sense that their choice of control has little impact on the desirability of various controls of other agents, then a more flexible inter-agent communication pattern may be sufficient for cost improvement.†

An important question is to clarify the extent to which agent coordination is essential. In what follows in this section, we will discuss a distributed asynchronous multiagent rollout scheme, which is based on the use of a signaling policy that provides estimates of coordinating information once the current state is known.

Autonomous Multiagent Rollout - Signaling Policies

An interesting possibility for autonomous control selection by the agents is to use a distributed rollout algorithm, which is augmented by a precomputed signaling policy that embodies agent coordination.‡ The idea is to assume that the agents do not communicate their computed rollout control components to the subsequent agents in the given order of local control selection. Instead, *once the agents know the state, they use precomputed (or easily computed) approximations to the control components of the preceding agents*, and compute their own control components in parallel and asyn-

† In particular, one may divide the agents in "coupled" groups, and require coordination of control selection only within each group, while the computation of different groups may proceed in parallel. Note that the "coupled" group formations may change over time, depending on the current state. For example, in applications where the agents' locations are distributed within some geographical area, it may make sense to form agent groups on the basis of geographic proximity, i.e., one may require that agents that are geographically near each other (and hence are more coupled) coordinate their control selections, while agents that are geographically far apart (and hence are less coupled) forego any coordination.

‡ The general idea of coordination by sharing information about the agents' policies arises also in other multiagent algorithmic contexts, including some that involve forms of policy gradient methods and Q-learning; see the surveys of the relevant research cited earlier. The survey by Matignon, Laurent, and Le Fort-Piat [MLL12] focuses on coordination problems from an RL point of view.

chronously. We call this algorithm *autonomous multiagent rollout*. While this type of algorithm involves a form of redundant computation, it allows for additional speedup through parallelization.

The algorithm at the kth stage uses a base policy $\mu_k = \{\mu_k^1, \ldots, \mu_k^{m-1}\}$, but also uses *a second policy* $\widehat{\mu}_k = \{\widehat{\mu}_k^1, \ldots, \widehat{\mu}_k^{m-1}\}$, called the *signaling policy*, which is computed off-line, is known to all the agents for on-line use, and is designed to play an agent coordination role. Intuitively, $\widehat{\mu}_k^\ell(x_k)$ *provides an intelligent "guess" about what agent* ℓ *will do at state* x_k. This is used in turn by all other agents $i \neq \ell$ to compute asynchronously their own rollout control components on-line.

More precisely, the autonomous multiagent rollout algorithm uses the base and signaling policies to generate a rollout policy $\tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$ as follows. At stage k and state x_k, $\tilde{\mu}_k(x_k) = \big(\tilde{\mu}_k^1(x_k), \ldots, \tilde{\mu}_k^m(x_k)\big)$, is obtained according to

$$\begin{aligned}
\tilde{\mu}_k^1(x_k) \in \arg \min_{u_k^1 \in U_k^1(x_k)} & E\Big\{ g_k\big(x_k, u_k^1, \mu_k^2(x_k), \ldots, \mu_k^m(x_k), w_k\big) \\
& \qquad + J_{k+1,\pi}\Big(f_k\big(x_k, u_k^1, \mu_k^2(x_k), \ldots, \mu_k^m(x_k), w_k\big)\Big)\Big\}, \\
\tilde{\mu}_k^2(x_k) \in \arg \min_{u_k^2 \in U_k^2(x_k)} & E\Big\{ g_k\big(x_k, \widehat{\mu}_k^1(x_k), u_k^2, \ldots, \mu_k^m(x_k), w_k\big) \\
& \qquad + J_{k+1,\pi}\Big(f_k\big(x_k, \widehat{\mu}_k^1(x_k), u_k^2, \ldots, \mu_k^m(x_k), w_k\big)\Big)\Big\}, \\
& \qquad \cdots \qquad \cdots \qquad \cdots \\
\tilde{\mu}_k^m(x_k) \in \arg \min_{u_k^m \in U_k^m(x_k)} & E\Big\{ g_k\big(x_k, \widehat{\mu}_k^1(x_k), \ldots, \widehat{\mu}_k^{m-1}(x_k), u_k^m, w_k\big) \\
& \qquad + J_{k+1,\pi}\Big(f_k\big(x_k, \widehat{\mu}_k^1(x_k), \ldots, \widehat{\mu}_k^{m-1}(x_k), u_k^m, w_k\big)\Big)\Big\}.
\end{aligned} \tag{2.78}$$

Note that the preceding computation of the controls $\tilde{\mu}_k^1(x_k), \ldots, \tilde{\mu}_k^m(x_k)$ can be done asynchronously and in parallel, and without direct agent coordination, since the signaling policy values $\widehat{\mu}_k^1(x_k), \ldots, \widehat{\mu}_k^{m-1}(x_k)$ are precomputed and are known to all the agents.

The simplest choice is to *use as signaling policy* $\widehat{\mu}$ *the base policy* μ. However, this choice does not guarantee policy improvement as evidenced by Example 2.9.3. In fact performance deterioration with this choice is not uncommon, and can be observed in more complicated examples, including the following.

Example 2.9.4 (Spiders and Flies - Use of the Base Policy for Signaling)

Consider the problem of Example 2.9.1, which involves two spiders and two flies on a line, and the base policy μ that moves a spider towards the closest surviving fly (and in case where a spider starts at the midpoint between

the two flies, moves the spider to the right). Assume that we use as signaling policy $\widehat{\mu}$ the base policy μ. It can then be verified that if the spiders start from different positions, the rollout policy will be optimal (will move the spiders in opposite directions). If, however, the spiders start *from the same position*, a completely symmetric situation is created, whereby the rollout controls move both flies in the direction of the fly *furthest away* from the spiders' position (or to the left in the case where the spiders start at the midpoint between the two flies). Thus, the flies end up oscillating around the middle of the interval between the flies and never catch the flies!

The preceding example is representative of a broad class of counterexamples that involve multiple identical agents. If the agents start at the same initial state, with a base policy that has identical components, and use the base policy for signaling, the agents will select identical controls under the corresponding multiagent rollout policy, ending up with a potentially serious cost deterioration.

This example also highlights an effect of the sequential choice of the control components $u_k^1, \ldots, u_k^m$, based on the reformulated problem of Fig. 2.9.1: it tends to break symmetries and "group think" that guides the agents towards selecting the same controls under identical conditions. Generally, any sensible multiagent policy must be able to deal in some way with this "group think" issue. One simple possibility is for each agent ℓ to randomize somehow the control choices of other agents $j \neq \ell$ when choosing its own control, particularly in "tightly coupled" cases where the choice of agent ℓ is "strongly" affected by the choices of the agents $j \neq \ell$.

An alternative idea is to choose the signaling policy $\widehat{\mu}_k$ to approximate the sequential multiagent rollout policy (the one computed with each agent knowing the controls applied by the preceding agents), or some other policy that is known to embody coordination between the agents. In particular, we may obtain $\widehat{\mu}_k$ as the multiagent rollout policy for a related but simpler problem, such as a certainty equivalent version of the original problem, whereby the stochastic system is replaced by a deterministic one.

Another interesting possibility is to compute $\widehat{\mu}_k = (\widehat{\mu}_k^1, \ldots, \widehat{\mu}_k^m)$ by off-line training of a neural network (or m networks, one per agent) with training samples generated through the sequential multiagent rollout policy. We intuitively expect that if the neural network provides a signaling policy that approximates well the sequential multiagent rollout policy, we would obtain better performance than the base policy. This expectation was confirmed in a case study involving a large-scale multi-robot repair application (see [BKB20]).

The advantage of autonomous multiagent rollout with neural network or other type of approximations is that it may lead to approximate policy improvement, while at the same time allowing asynchronous distributed agent operation without on-line agent coordination through communication of their rollout control values (but still assuming knowledge of the exact

state by all agents).

2.10 ROLLOUT FOR BAYESIAN OPTIMIZATION AND SEQUENTIAL ESTIMATION

In this section, we discuss a wide class of problems that has been studied intensively in statistics and related fields since the 1940s. Roughly speaking, in these problems we use observations and sampling for the purpose of inference, but the number and the type of observations are not fixed in advance. Instead, the outcomes of the observations are sequentially evaluated on-line with a view towards stopping or modifying the observation process. This involves sequential decision making, thus bringing to bear exact and approximate DP. A central issue here is to estimate an m-dimensional random vector θ, using optimal sequential selection of observations, which are based on feedback from preceding observations; see Fig. 2.10.1.

For a simple illustrative example, let us consider a hypothesis testing problem whereby we can make observations, at a cost C each, relating to two hypotheses. Given a new observation, we can either accept one of the hypotheses or delay the decision for one more period, pay the cost C, and obtain a new observation. At issue is trading off the cost of observation with the higher probability of accepting the wrong hypothesis. As an example, in a quality control setting, the two hypotheses may be that a certain product meets or does not meet a certain level of quality, while the observations may consist of quantitative tests of the quality of the product.

Intuitively, one expects that once the conditional probability of one of the hypotheses, given the observations thus far, gets sufficiently close to 1, we should stop the observations. Indeed classical DP analyses bear this out; see e.g., the books by Chernoff [Che72], DeGroot [DeG70], Whittle [Whi82], and the references quoted therein. In particular, the simple version of the hypothesis testing problem just described admits a simple and elegant optimal solution, known as the *sequential probability ratio test*. On the other hand more complex versions of the problem, involving for example multiple hypotheses and/or multiple types of observations, are computationally intractable, thus necessitating the use of suboptimal approaches.

An important distinction in sequential estimation problems is whether the current choice of observation affects the cost and the availability of future observations. If this is so, the problem can often be viewed most fruitfully as a *combined estimation and control problem*, and is related to a type of *adaptive control* problem that we will discuss in the next section. As an example we will consider there sequential decoding, whereby we search for a hidden code word by using a sequence of queries, in the spirit of the Wordle puzzle and the family of Mastermind games [see, e.g., the Wikipedia page for "Mastermind (board game)"].

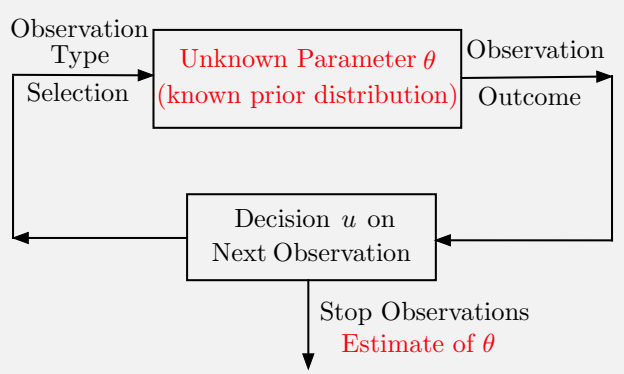


Figure 2.10.1 Illustration of sequential estimation of an unknown parameter θ. At each time a decision is made to select one of several observation types relating to θ, each of different cost, or to stop the observations and provide a final estimate of θ.

If the observation choices are "independent" and do not affect the cost or availability of future observations, the problem is substantially simplified. We will discuss problems of this type in the present section, starting with the case of surrogate and Bayesian optimization.

Surrogate Optimization

Surrogate optimization refers to a collection of methods, which address suboptimally a broad range of minimization problems, beyond the realm of DP. The idea is to minimize approximately a function that is given as a "black box." By this we mean a function whose analytical expression is unknown, and whose values at any one point may be hard-to-compute, e.g., may requite costly simulation or experimentation cost functions with "surrogates" that are easier to obtain.

Here we introduce a model of the cost function that is parametrized by a parameter θ; see Fig. 2.10.2. We observe sequentially the cost function at a few observation points, construct a model of the cost function (the surrogate) by estimating θ based on the results of the observations, and minimize the surrogate to obtain a suboptimal solution. The question is how to select observation points sequentially, using feedback from previous observations. This selection process often embodies an *exploration-exploitation tradeoff*: Observing at points likely to have near-optimal value vs observing at points in relatively unexplored areas of the search space.

Bayesian Optimization

Bayesian optimization (BO) has been used widely for the approximate optimization of functions whose values at given points can only be obtained

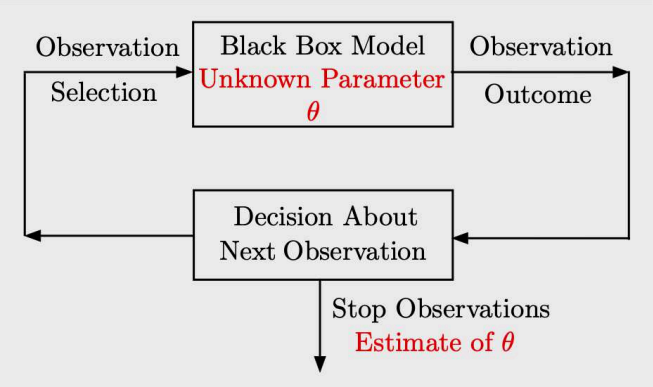


Figure 2.10.2 Illustration of construction of a surrogate for a "black box" function f whose values are hard-to-compute. We replace f with a parametric model that involves a parameter θ to be estimated by using observations at some points. The points are selected sequentially, using the results of earlier observations. Eventually, the observation process is stopped (often when an observation/computation budget limit is reached), and the final estimate of θ is used to construct the surrogate to be minimized in place of f.

through time-consuming calculation, simulation, or experimentation. A classical application from geostatistical interpolation, pioneered by the statisticians Matheron and Krige, was to identify locations of high gold distribution in South Africa based on samples from a few boreholes (the name "kriging" is often used to refer to this type of application; see the review by Kleijnen [Kle09]). As another example, BO has been used to select the hyperparameters of machine-learning models, including the architectural parameters of the deep neural network of AlphaZero; see [SHS17].

In this section, we will focus on a relatively simple BO formulation that can be viewed as the special case of surrogate optimization. In particular, we will discuss the case where the surrogate function is parametrized by the collection of its values at the points where it is defined.† Formally, we want to minimize a real-valued function f, defined over a set of m points, which we denote by $1, \ldots, m$. These m points lie in some space, which we leave unspecified for the moment.‡ The values of the function are not readily available, but can be estimated with observations that may be

† More complex forms of surrogates are obtained through linear combinations of some basis functions, with the parameter vector θ consisting of the weights of the basis functions. See the references cited later in this section.

‡ We restrict the domain of definition of f to be the finite set $\{1, \ldots, m\}$ in order to facilitate the implementation of the rollout algorithm to be discussed in what follows. However, in a more general and sometimes more convenient formulation, the domain of f can be an infinite set, such as a subset of a finite-dimensional Euclidean space.

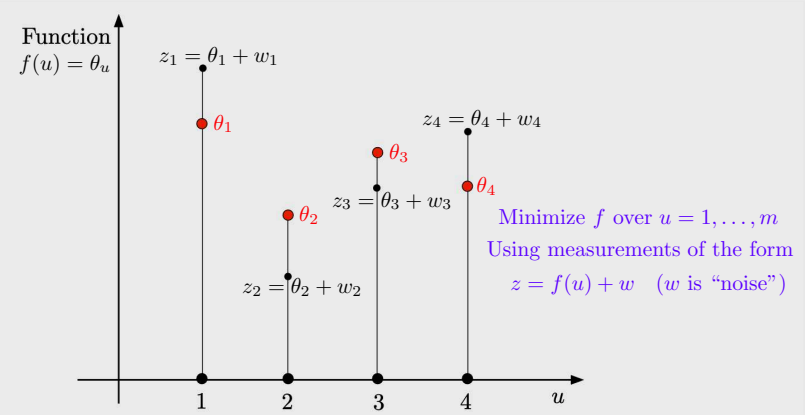


Figure 2.10.3 Illustration of a function f that we wish to estimate. The function is defined at the points $u = 1, 2, 3, 4$, and is represented by a vector $\theta = (\theta_1, \theta_2, \theta_3, \theta_4) \in \Re^4$, in the sense that $f(u) = \theta_u$ for all u. The prior distribution of θ is given, and is used to construct the posterior distribution of θ given noisy observations $z_u = \theta_u + w_u$ at some of the points u.

imperfect. However, the observations are so costly that we can only hope to observe the function at a limited number of points. Once the function has been estimated with this type of observation process, we obtain a surrogate cost function, which may be minimized to obtain an approximately optimal solution.

We denote the value of f at a point u by θ_u:

$$\theta_u = f(u), \qquad \text{for all } u = 1, \ldots, m.$$

Thus the m-dimensional vector $\theta = (\theta_1, \ldots, \theta_m)$ belongs to $\Re^m$ and represents the function f. We assume that we obtain sequentially noisy observations of values $f(u) = \theta_u$ at suitably selected points u. These values are used to estimate the vector θ (i.e., the function f), and to ultimately minimize (approximately) f over the m points $u = 1, \ldots, m$. The essence of the problem is to select points for observation based on an exploration-exploitation tradeoff (exploring the potential of relatively unexplored candidate solutions and improving the estimate of promising candidate solutions). The fundamental idea of the BO methodology is that the function value changes relatively slowly, so that observing the function value at some point provides information about the function values at neighboring points. Thus a limited number of strategically chosen observations can provide reasonable approximation to the true cost function over a large portion of the search space.

For a mathematical formulation of a BO framework, we assume that at each of N successive times $k = 1, \ldots, N$, we select a single point $u_k \in \{1, \ldots, m\}$, and observe the corresponding component θ_{u_k} of θ (i.e., the

function value at u_k) with some noise w_{u_k}, i.e.,

$$z_{u_k} = \theta_{u_k} + w_{u_k}; \tag{2.79}$$

see Fig. 2.10.3. We view the observation points $u_1, \ldots, u_N$ as the optimization variables (or controls/actions in a DP/RL context), and consider policies for selecting u_k with knowledge of the preceding observations $z_{u_1}, \ldots, z_{u_{k-1}}$ that have resulted from the selections $u_1, \ldots, u_{k-1}$. We assume that the noise random variables w_u, $u \in \{1, \ldots, m\}$ are independent and that their distributions are given. Moreover, we assume that θ has a given a priori distribution on the space of m-dimensional vectors $\Re^m$, which we denote by b_0. The posterior distribution of θ, given any subset of observations

$$\{z_{u_1}, \ldots, z_{u_k}\},$$

is denoted by b_k.

An important special case arises when b_0 and the distributions of w_u, $u \in \{1, \ldots, m\}$, are Gaussian. In this case b_0 is a multidimensional Gaussian distribution, defined by its mean (based on prior knowledge, or an equal value for all $u = 1, \ldots, m$ in case of absence of such knowledge) and its covariance matrix [implying greater correlation for pairs (u, u') that are "close" to each other in some problem-specific sense, e.g., exponentially decreasing with the Euclidean distance between u and u']. A key consequence of this assumption is that the posterior distribution b_k is multidimensional Gaussian, and can be calculated in closed form by using well-known formulas.

More generally, b_k evolves according to an equation of the form

$$b_{k+1} = B_k(b_k, u_{k+1}, z_{u_{k+1}}), \qquad k = 0, \ldots, N-1. \tag{2.80}$$

Thus given the set of observations up to time k, and the next choice u_{k+1}, resulting in an observation value $z_{u_{k+1}}$, the function B_k gives the formula for updating b_k to b_{k+1}, and may be viewed as a recursive estimator of b_k. In the Gaussian case, the function B_k can be written in closed form, using standard formulas for Gaussian random vector estimation. In other cases where no closed form expression is possible, B_k can be implemented through simulation that computes (approximately) the new posterior b_{k+1} using samples generated from the current posterior b_k.

At the end of the sequential estimation process, after the complete observation set

$$\{z_{u_1}, \ldots, z_{u_N}\}$$

has been obtained, we have the posterior distribution b_N of θ, which we can use to compute a surrogate of f. As an example we may use as surrogate the posterior mean $\hat{\theta} = (\hat{\theta}_1, \ldots, \hat{\theta}_m)$, and declare as minimizer of f over u the point u^* with minimum posterior mean:

$$u^* \in \arg\min\{\hat{\theta}_u \mid u = 1, \ldots, m\};$$

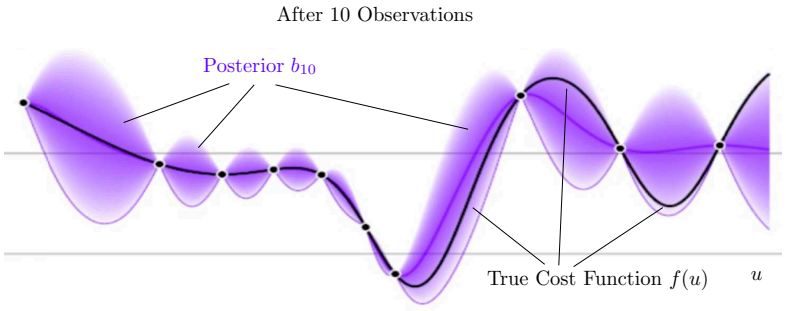


Figure 2.10.4 Illustration of the true cost function f, defined over an interval of the real line, and the posterior distribution b_{10} after noise-free measurements at 10 points. The shaded area represents the interval of the mean plus/minus the standard deviation of the posterior b_{10} at the points u. The mean of the finally obtained posterior, as a function of u, may be viewed as a surrogate cost function that can be minimized in place of f. Note that since the observations are assumed noise-free, the mean of the posterior is exact at the observation points.

see Fig. 2.10.4.

There is a large literature relating to the surrogate and Bayesian optimization methodology and its applications, particularly for the Gaussian case. We refer to the books by Rasmussen and Williams [RaW06], Powell and Ryzhov [PoR12], the highly cited papers by Saks et al. [SWM89], Jones, Schonlau, and Welch [JSW98], and Queipo et al. [QHS05], the reviews by Sasena [Sas02], Powell and Frazier [PoF08], Forrester and Keane [FoK09], Kleijnen [Kle09], Brochu, Cora, and De Freitas [BCD10], Ryzhov, Powell, and Frazier [RPF12], Ghavamzadeh, Mannor, Pineau, and Tamar [GMP15], Shahriari et al. [SSW16], and Frazier [Fra18], and the references quoted there. Our purpose here is to focus on the aspects of the subject that are most closely connected to exact and approximate DP.

A Dynamic Programming Formulation

The sequential estimation problem just described, viewed as a DP problem, involves a state at time k, which is the posterior (or belief state) b_k, and a control/action at time k, which is the point index u_{k+1} selected for observation. The transition equation according to which the state evolves, is

$$b_{k+1} = B_k(b_k, u_{k+1}, z_{u_{k+1}}), \qquad k = 0, \ldots, N-1;$$

cf. Eq. (2.80). To complete the DP formulation, we need to introduce a cost structure. To this end, we assume that observing θ_u, as per Eq. (2.79), incurs a cost $c(u)$, and that there is a terminal cost $G(b_N)$ that depends of the final posterior distribution; as an example, the function G may involve the mean and covariance corresponding to b_N.

The corresponding DP algorithm is given by

$$J_k^*(b_k) = \min_{u_{k+1}\in\{1,\ldots,m\}} \Big[c(u_{k+1}) + E_{z_{u_{k+1}}} \Big\{ J_{k+1}^*\big(B_k(b_k, u_{k+1}, z_{u_{k+1}})\big) \mid b_k, u_{k+1} \Big\} \Big], \tag{2.81}$$

and proceeds backwards from the terminal condition

$$J_N^*(b_N) = G(b_N). \tag{2.82}$$

The expected value in the right side of the DP equation (2.81) is taken with respect to the conditional distribution of $z_{u_{k+1}}$, given b_k and the choice u_{k+1}. The observation cost $c(u)$ may be 0 or a constant for all u, but it can also have a more complicated dependence on u. The terminal cost $G(b_N)$ may be a suitable measure of surrogate "fidelity" that depends on the posterior mean and covariance of θ corresponding to b_N.

Generally, executing the DP algorithm (2.81) is practically infeasible, because the space of posterior distributions is infinite-dimensional. In the Gaussian case where the a priori distribution b_0 is Gaussian and the noise variables w_u are Gaussian, the posterior b_k is m-dimensional Gaussian, so it is characterized by its mean and covariance, and can be specified by a finite set of numbers. Despite this simplification, the DP algorithm (2.81) is prohibitively time-consuming even under Gaussian assumptions, except for simple special cases. We consequently resort to approximation in value space, whereby the function J_{k+1}^* in the right side of Eq. (2.81) is replaced by an approximation $\tilde{J}_{k+1}$.

Approximation in Value Space

The most popular BO methodology makes use of a myopic/greedy policy μ_{k+1}, which at each time k and given b_k, selects a point $\hat{u}_{k+1} = \mu_{k+1}(b_k)$ for the next observation, using some calculation involving an *acquisition function*. This function, denoted $A_k(b_k, u_{k+1})$, quantifies some form of "expected benefit" for an observation at u_{k+1}, given the current posterior b_k.† The myopic policy selects the next point at which to observe, $\hat{u}_{k+1}$,

† A common type of acquisition function is the *upper confidence bound*, which has the form

$$A_k(b_k, u) = T_k(b_k, u) + \beta R_k(b_k, u),$$

where $T_k(b_k, u)$ is the negative of the mean of $f(u)$ under the posterior distribution b_k, $R_k(b_k, u)$ is the standard deviation of $f(u)$ under the posterior distribution b_k, and β is a tunable positive scalar parameter. Thus $T_k(b_k, u)$ can be

by maximizing the acquisition function:

$$\hat{u}_{k+1} \in \arg \max_{u_{k+1} \in \{1,\ldots,m\}} A_k(b_k, u_{k+1}). \tag{2.83}$$

Several ways to define suitable acquisition functions have been proposed, and an important issue is to be able to calculate economically its values $A_k(b_k, u_{k+1})$ for the purposes of the maximization in Eq. (2.83). Another important issue of course is to be able to calculate the posterior b_k economically.

Approximation in value space is an alternative approach, which is based on the DP formulation of the preceding section. In particular, in this approach we approximate the DP algorithm (2.81) by replacing J^*_{k+1} with an approximation $\tilde{J}_{k+1}$ in the minimization of the right side. Thus we select the next observation at point $\tilde{u}_{k+1}$ according to

$$\tilde{u}_{k+1} \in \arg \min_{u_{k+1} \in \{1,\ldots,m\}} Q_k(b_k, u_{k+1}), \qquad k = 0, \ldots, N-1, \tag{2.84}$$

where $Q_k(b_k, u_{k+1})$ is the Q-factor corresponding to the pair (b_k, u_{k+1}), given by

$$Q_k(b_k, u_{k+1}) = c(u_{k+1}) + E_{z_{u_{k+1}}} \Big\{ \tilde{J}_{k+1}\big(B_k(b_k, u_{k+1}, z_{u_{k+1}})\big) \;\Big|\; b_k, u_{k+1} \Big\}. \tag{2.85}$$

The expected value in the preceding equation is taken with respect to the conditional probability distribution of $z_{u_{k+1}}$ given (b_k, u_{k+1}), which can be computed using b_k and the given distribution of the noise $w_{u_{k+1}}$. Thus if b_k and $\tilde{J}_{k+1}$ are available, we may use Monte Carlo simulation to determine the Q-factors $Q_k(b_k, u_{k+1})$ for all $u_{k+1} \in \{1, \ldots, m\}$, and select as next point for observation the one that corresponds to the minimal Q-factor [cf. Eq. (2.84)].

Rollout Algorithms for Bayesian Optimization

A special case of approximation in value space is the rollout algorithm, whereby the function J^*_{k+1} in the right side of the DP Eq. (2.81) is replaced by the cost function of some base policy $\mu_{k+1}(b_k)$, $k = 0, \ldots, N-1$. Thus,

---

viewed as an *exploitation index* (encoding our desire to search within parts of the space where f takes low value), while $R_k(b_k, u)$ can be viewed as an *exploration index* (encoding our desire to search within parts of the space that are relatively unexplored). There are several other popular acquisition functions, which directly or indirectly embody a tradeoff between exploitation and exploration. A popular example is the *expected improvement* acquisition function, which is equal to the expected value of the reduction of $f(u)$ relative to the minimal value of f obtained up to time k (under the posterior distribution b_k).

given a base policy the rollout algorithm uses the cost function of this policy as the function $\tilde{J}_{k+1}$ in the approximation in value space scheme (2.84)-(2.85). The values of $\tilde{J}_{k+1}$ needed for the Q-factor calculations in Eq. (2.85) can be computed or approximated by simulation. Greedy/myopic policies based on an acquisition function [cf. Eq. (2.83)] have been suggested as base policies in various rollout proposals.†

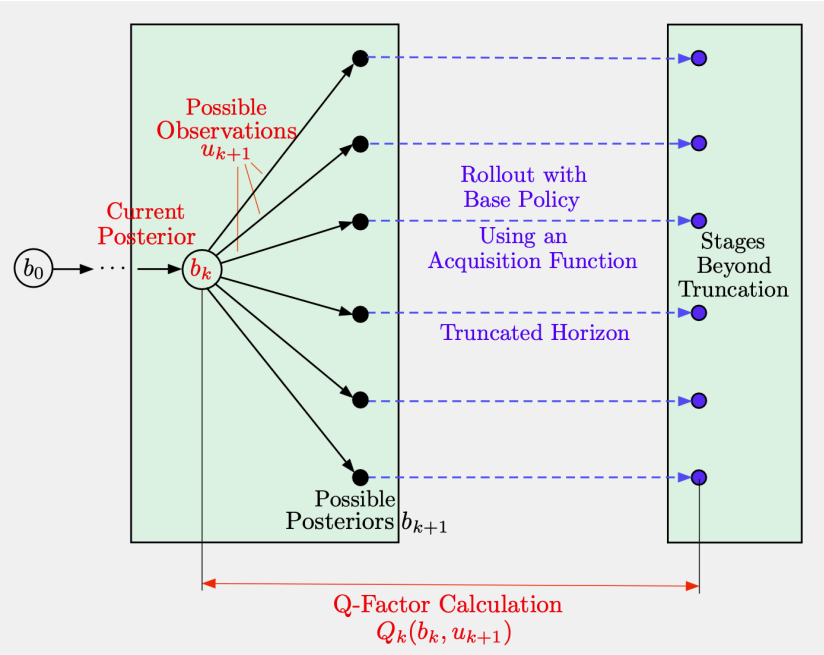


Figure 2.10.5 Illustration of rollout at the current posterior b_k. For each $u_{k+1} \in \{1, \ldots, m\}$, we compute the Q-factor $Q_k(b_k, u_{k+1})$ by using Monte-Carlo simulation with samples from $w_{u_{k+1}}$ and a base heuristic that uses an acquisition function starting from each possible posterior b_{k+1}. The rollout may extend to the end of the horizon N, or it may be truncated after a few steps.

In particular, given b_k, the rollout algorithm computes for each $u_{k+1} \in$

† The rollout algorithm for BO was first proposed under Gaussian assumptions by Lam, Wilcox, and Wolpert [LWW16]. It was further discussed by Jiang et al. [JJB20], [JCG20], Lee at al. [LEC20], Lee [Lee20], Yue and Kontar [YuK20], Lee et al. [LEP21], Paulson, Sorouifar, and Chakrabarty [PSC22], where it is also referred to as "nonmyopic BO" or "nonmyopic sequential experimental design." For related work, see Gerlach, Hoffmann, and Charlish [GHC21]. These papers also discuss various approximations to the rollout approach, and generally report encouraging computational results. Section 3.5 of the author's book [Ber20a] focuses on rollout algorithms for surrogate and Bayesian optimization.

$\{1, \ldots, m\}$ a Q-factor value $Q_k(b_k, u_{k+1})$ by simulating the base policy for multiple time steps starting from all possible posteriors b_{k+1} that can be generated from (b_k, u_{k+1}), and by accumulating the corresponding cost [including a terminal cost such as $G(b_N)$]; see Fig. 2.10.5. It then selects the next point $\tilde{u}_{k+1}$ for observation by using the Q-factor minimization of Eq. (2.84).

Note that the equation

$$b_{k+1} = B_k(b_k, u_{k+1}, z_{u_{k+1}}), \qquad k = 0, \ldots, N-1,$$

which governs the evolution of the posterior distribution (or belief state), is stochastic because $z_{u_{k+1}}$ involves the stochastic noise $w_{u_{k+1}}$. Thus some Monte Carlo simulation is unavoidable in the calculation of the Q-factors $Q_k(b_k, u_{k+1})$. On the other hand, one may greatly reduce the Monte Carlo computational burden by employing a certainty equivalence approximation, which at stage k, treats only the noise $w_{u_{k+1}}$ as stochastic, and replaces the noise variables $w_{u_{k+2}}, w_{u_{k+3}}, \ldots$, after the first stage of the calculation, by deterministic quantities such as their means $\hat{w}_{u_{k+2}}, \hat{w}_{u_{k+3}}, \ldots$.

The simulation of the Q-factor values may also involve other approximations, some of which have been suggested in various proposals for rollout-based BO. For example, if the number of possible observations m is very large, we may compute and compare the Q-factors of only a subset. In particular, at a given time k, we may rank the observations by using an acquisition function, select a subset U_{k+1} of most promising observations, compute their Q-factors $Q_k(b_k, u_{k+1})$, $u_{k+1} \in U_{k+1}$, and select the observation whose Q-factor is minimal; this idea has been used in the case of the Wordle puzzle in the paper by Bhambri, Bhattacharjee, and Bertsekas [BBB22], which will be discussed in the next section.

Multiagent Rollout for Bayesian Optimization

In some BO applications there arises the possibility of simultaneously performing multiple observations before receiving feedback about the corresponding observation outcomes. This occurs, among others, in two important contexts:

(a) In parallel computation settings, where multiple processors are used to perform simultaneously expensive evaluations of the function f at multiple points u. These evaluations may involve some form of truncated simulation, so they yield evaluations of the form $z_u = \theta_u + w_u$, where w_u is the simulation noise.

(b) In distributed sensor systems, where a number of sensors provide in parallel relevant information about the random vector θ that we want to estimate; see e.g., the recent paper by Li, Krakow, and Gopalswamy [LKG21], which describes related multisensor estimation problems, based on the multiagent rollout methodology of Section 2.9.

Of course in such cases we may treat the entire set of simultaneous observations as a single observation within an enlarged Cartesian product space of observations, but there is a fundamental difficulty: the size of the observation space (and hence the number of Q-factors to be calculated by rollout at each time step) grows exponentially with the number of simultaneous observations. This in turn greatly increases the computational requirements of the rollout algorithm.

To address this difficulty, we may employ the methodology of multiagent rollout whereby the policy improvement is done one-agent-at-a-time in a given order, with (possibly partial) knowledge of the choices of the preceding agents in the order. As a result, the amount of computation for each policy improvement grows linearly with the number of agents, as opposed to exponentially for the standard all-agents-at-once method. At the same time the theoretical cost improvement property of the rollout algorithm can be shown to be preserved, while the empirical evidence suggests that great computational savings are achieved with hardly any performance degradation.

Generalization to Sequential Estimation of Random Vectors

Aside from BO, there are several other types of simple sequential estimation problems, which involve "independent sampling," i.e., problems where the choice of an observation type does not affect the quality, cost, or availability of observations of other types. A common class of problems that contains BO as a special case and admits a similar treatment, is to sequentially estimate an m-dimensional random vector $\theta = (\theta_1, \ldots, \theta_m)$ by using N linear observations of θ of the form

$$z_u = a_u'\theta + w_u, \qquad u \in \{1, \ldots, n\},$$

where n is some integer. Here w_u are independent random variables with given probability distributions, the m-dimensional vectors a_u are known, and $a_u'\theta$ denotes the inner product of a_u and θ. Similar to the case of BO, the problem simplifies if the given a priori distribution of θ is Gaussian, and the random variables w_u are independent and Gaussian. Then, the posterior distribution of θ, given any subset of observations, is Gaussian (thanks to the linearity of the observations), and can be calculated in closed form.

Observations are generated sequentially at times $1, \ldots, N$, one at a time and with knowledge of the outcomes of the preceding observations, by choosing an index $u_k \in \{1, \ldots, n\}$ at time k, at a cost $c(u_k)$. Thus u_k are the optimization variables, and affect both the quality of estimation of θ and the observation cost. The objective, roughly speaking, is to select N observations to estimate θ in a way that minimizes an appropriate cost function; for example, one that penalizes some form of estimation

error plus the cost of the observations. We can similarly formulate the corresponding optimization problem in terms of N-stage DP, and develop rollout algorithms for its approximate solution.

2.11 ADAPTIVE CONTROL BY ROLLOUT WITH A POMDP FORMULATION

In this section, we discuss various approaches for the approximate solution of Partially Observed Markovian Decision Problems (POMDP) with a special structure, which is well-suited for adaptive control, as well as other contexts that involve search for a hidden object.† It is well known that POMDP are among the most challenging DP problems, and nearly always require the use of approximations for (suboptimal) solution.

The application and implementation of rollout and approximate PI methods to general finite-state POMDP is described in the author's RL book [Ber19a] (Section 5.7.3). Here we will focus attention on a special class of POMDP where the state consists of two components:

(a) A perfectly observed component x_k that evolves over time according to a discrete-time equation.

(b) An unobserved component θ that 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, hence the connection with adaptive control. Thus we have

$$x_{k+1} = f_k(x_k, \theta, u_k, w_k), \tag{2.86}$$

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). We will assume that θ can take one of m known values $\theta^1, \ldots, \theta^m$:

$$\theta \in \{\theta^1, \ldots, \theta^m\}.$$

see Fig. 2.11.1.

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, we assume that the information vector

$$I_k = \{x_0, \ldots, x_k, u_0, \ldots, u_{k-1}\}$$

† In Section 1.6.6, we discussed the indirect adaptive control approach, which enforces a separation of the controller into a system identification algorithm and a policy reoptimization algorithm. The POMDP approach of this section (also summarized in Section 1.6.6), does not assume such an a priori separation, and is thus founded on a more principled algorithmic framework.

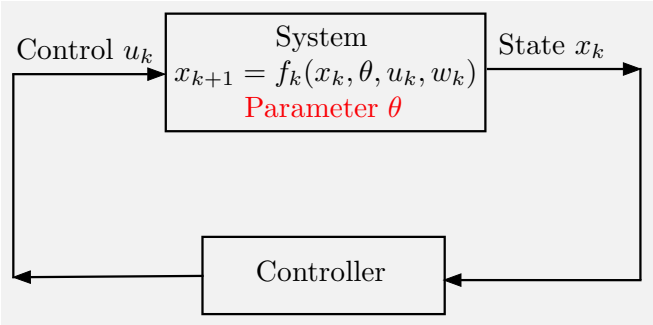


Figure 2.11.1 Illustration of an adaptive control scheme involving perfect state observation of a system with an unknown parameter θ. At each time a decision is made to select a control and (possibly) one of several observation types, each of different cost.

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) that is commonly called the *belief state* of the POMDP at time k.

Note that according to the classical methodology of POMDP (see e.g., [Ber17a], Chapter 4), the belief component b_{k+1} is determined by the belief state (x_k, b_k), the control u_k, and the observation obtained at time $k+1$, i.e., x_{k+1}. Thus b_k can be updated according to an equation of the form

$$b_{k+1} = B_k(x_k, b_k, u_k, x_{k+1}),$$

where B_k is an appropriate function, which can be viewed as a recursive estimator of θ. There are several approaches to implement this estimator (perhaps with some approximation error), including the use of Bayes' rule and the simulation-based method of particle filtering.

The preceding mathematical model forms the basis for a classical adaptive control formulation, where each θ^i represents a set of system parameters, and the computation of the belief probabilities $b_{k,i}$ can be viewed as the outcome of a system identification algorithm. In this context, the problem becomes one of *dual control*, a combined identification and control problem, whose optimal solution is notoriously difficult.

Another interesting context arises in search problems, where θ specifies the locations of one or more objects of interest within a given space. Some puzzles, including the popular Wordle game, fall within this category, as we will discuss briefly later in this section.

The Exact DP Algorithm - Approximation in Value Space

We will now describe an exact DP algorithm that operates in the space of information vectors I_k. To describe this algorithm, let us denote by $J_k(I_k)$ the optimal cost starting at information vector I_k at time k. We can view I_k as a state of the POMDP, which evolves over time according to the equation

$$I_{k+1} = (I_k, x_{k+1}, u_k) = \big(I_k, f_k(x_k, \theta, u_k, w_k), u_k\big),$$

Viewing this as a system equation, whose right hand side involves the state I_k, the control u_k, and the disturbance w_k, the DP algorithm takes the form

$$\begin{aligned} J_k^*(I_k) &= \min_{u_k \in U_k(x_k)} E_{\theta, w_k}\Big\{ g_k(x_k, \theta, u_k, w_k) + J_{k+1}^*(I_{k+1}) \mid I_k, u_k \Big\} \\ &= \min_{u_k \in U_k(x_k)} E_{\theta, w_k}\Big\{ g_k(x_k, \theta, u_k, w_k) + \\ &\qquad\qquad J_{k+1}^*\big(I_k, f_k(x_k, \theta, u_k, w_k), u_k\big) \mid I_k, u_k \Big\}, \end{aligned} \tag{2.87}$$

for $k = 0, \ldots, N-1$, with $J_N(I_N) = g_N(x_N)$; see e.g., the DP textbook [Ber17a], Section 4.1.

By using the law of iterated expectations,

$$E_{\theta, w_k}\{\cdot \mid I_k, u_k\} = E_\theta\big\{E_{w_k}\{\cdot \mid I_k, \theta, u_k\} \mid I_k, u_k\big\},$$

we can rewrite this DP algorithm as

$$\begin{aligned} J_k^*(I_k) &= \min_{u_k \in U_k(x_k)} \sum_{i=1}^m b_{k,i} E_{w_k}\Big\{ g_k(x_k, \theta^i, u_k, w_k) + J_{k+1}^*(I_{k+1}) \mid I_k, \theta^i, u_k \Big\} \\ &= \min_{u_k \in U_k(x_k)} \sum_{i=1}^m b_{k,i} E_{w_k}\Big\{ g_k(x_k, \theta^i, u_k, w_k) + \\ &\qquad\qquad J_{k+1}^*\big(I_k, f_k(x_k, \theta^i, u_k, w_k), u_k\big) \mid I_k, \theta^i, u_k \Big\}. \end{aligned} \tag{2.88}$$

The summation over i above represents the expected value of θ conditioned on I_k and u_k.

The algorithm (2.88) is typically very hard to implement, in part because of the dependence of J_{k+1}^* on the entire information vector I_{k+1}, which expands in size according to

$$I_{k+1} = (I_k, x_{k+1}, u_k).$$

To address this implementation difficulty, we may use approximation in value space, based on replacing $J_{k+1}^*(I_{k+1})$ in the DP algorithms (2.87) and (2.88) with some function $\tilde{J}_{k+1}(I_{k+1})$ such that the expected value

$$E_{w_k}\Big\{ \tilde{J}_{k+1}(I_{k+1}) \mid I_k, \theta^i, u_k \Big\}$$

can be obtained (either off-line or on-line) with a tractable computation for any (I_k, θ^i, u_k).

Here we will focus on functions $\tilde{J}_{k+1}$ with a special structure that facilitates the implementation of the corresponding approximaion in value space scheme. One such possibility is to use the optimal cost functions corresponding to the m parameters θ^i,

$$\hat{J}^i_{k+1}(x_{k+1}), \qquad i = 1, \ldots, m. \tag{2.89}$$

In particular, $\hat{J}^i_{k+1}(x_{k+1})$ is the optimal cost that would be obtained starting from state x_{k+1} under the assumption that $\theta = \theta^i$; this corresponds to a perfect state information problem. Then an approximation in value space scheme with one-step lookahead minimization is given by

$$\tilde{u}_k \in \arg \min_{u_k \in U_k(x_k)} \sum_{i=1}^{m} b_{k,i} E_{w_k} \Big\{ g_k(x_k, \theta^i, u_k, w_k) + \hat{J}^i_{k+1}\big(f_k(x_k, \theta^i, u_k, w_k)\big) \mid x_k, \theta^i, u_k \Big\}. \tag{2.90}$$

In particular, instead of the optimal control, which minimizes the optimal Q-factor of (I_k, u_k) appearing in the right side of Eq. (2.87), we apply control $\tilde{u}_k$ that minimizes the expected value over θ of the optimal Q-factors that correspond to fixed values of θ.

In the case where the horizon is infinite, it is reasonable to expect that the estimate of the parameter θ improves over time, and that with a suitable estimation scheme, it converges asymptotically to the correct value of θ, call it θ^*, i.e.,

$$\lim_{k \to \infty} b_{k,i} = \begin{cases} 1 & \text{if } \theta^i = \theta^*, \\ 0 & \text{if } \theta^i \neq \theta^*. \end{cases}$$

Then it can be seen that the generated one-step lookahead controls $\tilde{u}_k$ are asymptotically obtained from the Bellman equation that corresponds to the correct parameter θ^*, and are typically optimal in some asymptotic sense. Schemes of this type have been extensively discussed in the adaptive control literature since the 70s; see the end-of-chapter references.

Generally, the optimal costs $\hat{J}^i_{k+1}(x_{k+1})$ that correspond to the different parameter values θ^i [cf. Eq. (2.89)] may be hard to compute, despite their perfect state information structure.† An alternative possibility is to use off-line trained feature-based or neural network-based approximations to $\hat{J}^i_{k+1}(x_{k+1})$. Another possibility, described next, is to use a rollout approach.

† In favorable special cases, such as linear quadratic problems, the optimal costs $\hat{J}^i_{k+1}(x_{k+1})$ may be easily calculated in closed form. Still, however, even in such cases the calculation of the belief probabilities $b_{k,i}$ may not be simple, and may require the use of a system identification algorithm.

Rollout and Cost Improvement

A simpler possibility for approximation in value space is to use the costs of given policies π^i in place of the optimal costs $\hat{J}^i_{k+1}(x_{k+1})$. In this case the one-step lookahead scheme (2.90) takes the form

$$\tilde{u}_k \in \arg\min_{u_k \in U_k(x_k)} \sum_{i=1}^{m} b_{k,i} E_{w_k} \Big\{ g_k(x_k, \theta^i, u_k, w_k) + \hat{J}^i_{k+1,\pi^i}\big(f_k(x_k, \theta^i, u_k, w_k)\big) \mid x_k, \theta^i, u_k \Big\}, \tag{2.91}$$

with $\pi^i = \{\mu^i_0, \ldots, \mu^i_{N-1}\}$, $i = 1, \ldots, m$, being known policies, *with components μ^i_k that depend only on x_k*. Here, the term

$$\hat{J}^i_{k+1,\pi^i}\big(f_k(x_k, \theta^i, u_k, w_k)\big)$$

in Eq. (2.91) is the cost of the base policy π^i, calculated starting from the next state

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

under the assumption that θ will stay fixed at the value $\theta = \theta^i$ until the end of the horizon. Note that the cost function of π^i, conditioned on $\theta = \theta^i$, x_k, and u_k, which is needed in Eq. (2.91), can be calculated by Monte Carlo simulation. This is made possible by the fact that the components μ^i_k of π^i depend only on x_k [rather than I_k or the belief state (x_k, b_k)].

The preceding scheme has the character of a rollout algorithm, but strictly speaking, it does not qualify as a rollout algorithm because the policy components μ^i_k involve a dependence on i in addition to the dependence on x_k. On the other hand if we restrict all the policies π^i to be the same for all i, the corresponding functions μ_k depend only on x_k and not on i, thus defining a legitimate base policy. It is then possible to view the rollout policy as being generated from the base policy through a policy iteration scheme. As a result, a cost improvement property can be shown.

Within our rollout context, a policy π such that $\pi^i = \pi$ for all i must be a *robust* policy, in the sense that it should work adequately well for all parameter values θ^i. The choice of such a policy is likely problem-dependent. On the other hand robust policies have a long history in the context of adaptive control, and have been discussed widely (see e.g., the book by Jiang and Jiang [JiJ17], and the references quoted therein).

The Case of a Deterministic System

Let us now consider the case where the system (2.86) is deterministic of the form

$$x_{k+1} = f_k(x_k, \theta, u_k). \tag{2.92}$$

Then, while the problem still has a stochastic character due to the uncertainty about the value of θ, the DP algorithm (2.88) and its approximation in value space counterparts are greatly simplified because there is no expectation over w_k to contend with. Indeed, given a state x_k, a parameter θ^i, and a control u_k, the on-line computation of the control of the rollout-like algorithm (2.91), takes the form

$$\tilde{u}_k \in \arg \min_{u_k \in U_k(x_k)} \sum_{i=1}^{m} b_{k,i} \Big(g_k(x_k, \theta^i, u_k) + \hat{J}^i_{k+1,\pi^i} \big(f_k(x_k, \theta^i, u_k) \big) \Big). \quad (2.93)$$

The computation of $\hat{J}^i_{k+1,\pi^i} \big(f_k(x_k, \theta^i, u_k) \big)$ involves a deterministic propagation from the state x_{k+1} of Eq. (2.92) up to the end of the horizon, using the base policy π^i, while assuming that θ is fixed at the value θ^i.

In particular, the term

$$Q_k(x_k, u_k, \theta^i) = g_k(x_k, \theta^i, u_k) + \hat{J}^i_{k+1,\pi^i} \big(f_k(x_k, \theta^i, u_k) \big) \quad (2.94)$$

appearing on the right side of Eq. (2.93) is viewed as a Q-factor that must be computed for every pair (u_k, θ^i), $u_k \in U_k(x_k)$, $i = 1, \ldots, m$, using the base policy π^i. The expected value of this Q-factor,

$$\hat{Q}_k(x_k, u_k) = \sum_{i=1}^{m} b_{k,i} Q_k(u_k, \theta^i),$$

must then be calculated for every $u_k \in U_k(x_k)$, and the computation of the rollout control $\tilde{u}_k$ is obtained from the minimization

$$\tilde{u}_k \in \arg \min_{u_k \in U_k(x_k)} \hat{Q}_k(x_k, u_k); \quad (2.95)$$

cf. Eq. (2.93). This computation is illustrated in Fig. 2.11.2.

The case of a deterministic system is particularly interesting because we can typically expect that the true parameter θ^* is identified in a finite number of stages, since at each stage k, we are receiving a noiseless measurement relating to θ, namely the state x_k. Once this happens, the problem becomes one of perfect state information.

An illustration similar to the one of Fig. 2.11.2 applies to the rollout scheme (2.91) for the case of a stochastic system. In this case, a Q-factor

$$Q_k(x_k, u_k, \theta^i, w_k) = g_k(x_k, \theta^i, u_k, w_k) + \hat{J}^i_{k+1,\pi^i} \big(f_k(x_k, \theta^i, u_k, w_k) \big)$$

must be calculated for every triplet (u_k, θ^i, w_k), using the base policy π^i. The rollout control $\tilde{u}_k$ is obtained by minimizing the expected value of this Q-factor [averaged using the distribution of (θ, w_k)]; cf. Eq. (2.91).

An interesting and intuitive example that demonstrates the deterministic system case is the popular Worlde puzzle.

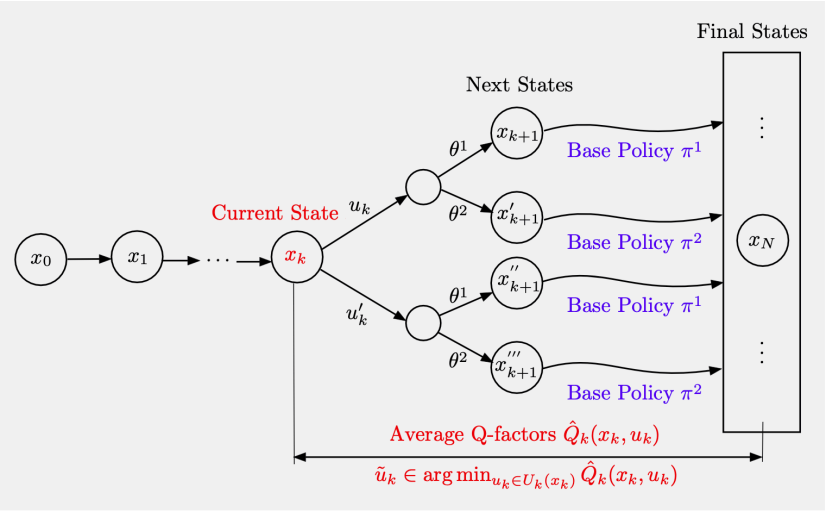


Figure 2.11.2 Schematic illustration of adaptive control by rollout for deterministic systems; cf. Eqs. (2.94) and (2.95). The Q-factors $Q_k(x_k, u_k, \theta^i)$ are averaged over θ^i, using the current belief distribution b_k, and the control applied is the one that minimizes the averaged Q-factor

$$\hat{Q}_k(x_k, u_k) = \sum_{i=1}^{m} b_{k,i} Q_k(x_k, u_k, \theta^i)$$

over $u_k \in U_k(x_k)$.

Example 2.11.1 (The Wordle Puzzle)

In the classical form of this puzzle, we try to guess a mystery word θ^* out of a known finite collection of 5-letter words. This is done with sequential guesses each of which provides additional information on the correct word θ^*, by using certain given rules to shrink the current mystery list (the smallest list that contains θ^*, based on the currently available information). The objective is to minimize the number of guesses to find θ^* (using more than 6 guesses is considered to be a loss). This type of puzzle descends from the classical family of Mastermind puzzles that centers around decoding a secret sequence of objects (e.g., letters or colors) using partial observations.

The rules for shrinking the mystery list relate to the common letters between the word guesses and the mystery word θ^*, and they will not be described here (there is a large literature regarding the Wordle puzzle). Moreover, θ^* is assumed to be chosen from the initial collection of 5-letter words according to a uniform distribution. Under this assumption, it can be shown that the belief distribution b_k at stage k continues to be uniform over the mystery list. As a result, we may use as state x_k the mystery list at stage k, which evolves deterministically according to an equation of the form (2.92), where u_k is the guess word at stage k. There are several base policies to

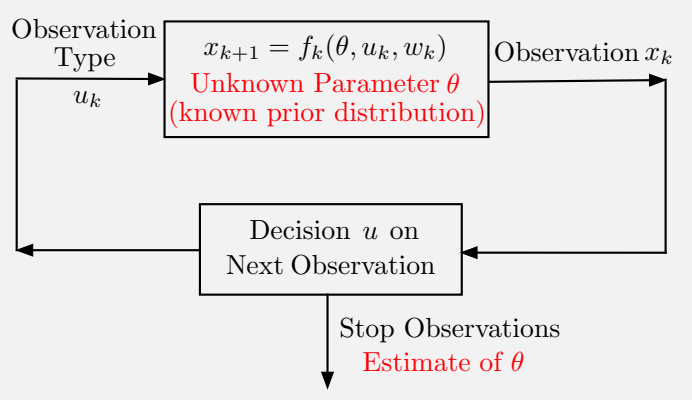


Figure 2.11.3 A view of sequential estimation as an adaptive control problem. The system function f_k does not depend on the current state x_k, so the system provides a decision-dependent noisy observation of θ.

use in the rollout-like algorithm (2.93), which are described in the paper by Bhambri, Bhattacharjee, and Bertsekas [BBB22], together with computational results, which show that the corresponding rollout algorithm (2.93) performs remarkably close to optimal.

The rollout approach also applies to several variations of the Wordle puzzle. Such variations may include for example a larger length $\ell > 5$ of mystery words, and/or a known nonuniform distribution over the initial collection of ℓ-letter words; see [BBB22].

The Case of Sequential Estimation - Alternative Base Policies

We finally note that the adaptive control framework of this section contains as a special case the sequential estimation framework of the preceding section. This special case corresponds to a dynamic system of the form

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

where the state x_{k+1} is the observation at time $k+1$ and exhibits no explicit dependence on the preceding observation x_k, but depends on the stochastic disturbance w_k, and on the decision u_k; cf. Figs. 2.11.1 and 2.11.3. This decision may involve a cost and determines the type of next obsevation out of a collection of possible types.

While the rollout methodology of the present section applies to sequential estimation problems, other rollout algorithms may also be used, depending on the problem's detailed structure. In particular, the rollout algorithms for Bayesian optimization of the works noted in Section 2.10 involve base policies that depend on the current belief state b_k, rather than the current state x_k. Another example of rollout for adaptive control, which uses a base policy that depends on the current belief state is given

in Section 6.7 of the book [Ber22a]. For work on related stochastic optimal control problems that involve observation costs and the rollout approach, see Antunes and Heemels [AnH14], and Khashooei, Antunes, and Heemels [KAH15].

2.12 ROLLOUT FOR MINIMAX CONTROL

The problem of optimal control of uncertain systems is usually treated within a stochastic framework, whereby all disturbances $w_0, \ldots, w_{N-1}$ are described by probability distributions, and the expected value of the cost is minimized. However, in many practical situations a stochastic description of the disturbances may not be available, but one may have information with less detailed structure, such as bounds on their magnitude. In other words, one may know a set within which the disturbances are known to lie, but may not know the corresponding probability distribution. Under these circumstances one may use a minimax approach, whereby the worst possible values of the disturbances within the given set are assumed to occur. Within this context, we take the view that the disturbances are chosen by an antagonistic opponent. The minimax approach is also connected with two-player games, when in lack of information about the opponent, we adopt a worst case viewpoint during on-line play, as well as with contexts where we wish to guard against adversarial attacks.†

To be specific, consider a finite horizon context, and assume that the disturbances $w_0, w_1, \ldots, w_{N-1}$ do not have a probabilistic description but rather are known to belong to corresponding given sets $W_k(x_k, u_k) \subset D_k$, $k = 0, 1, \ldots, N-1$, which may depend on the current state x_k and control u_k. The minimax control problem is to find a policy $\pi = \{\mu_0, \ldots, \mu_{N-1}\}$ with $\mu_k(x_k) \in U_k(x_k)$ for all x_k and k, which minimizes the cost function

$$J_\pi(x_0) = \max_{\substack{w_k \in W_k(x_k, \mu_k(x_k)) \\ k=0,1,\ldots,N-1}} \left[g_N(x_N) + \sum_{k=0}^{N-1} g_k\big(x_k, \mu_k(x_k), w_k\big) \right].$$

The DP algorithm for this problem takes the following form, which resembles the one corresponding to the stochastic DP problem (maximization is used in place of expectation):

$$J_N^*(x_N) = g_N(x_N), \tag{2.96}$$

† The minimax approach to decision and control has its origins in the 50s and 60s. It is also referred to by other names, depending on the underlying context, such as *robust control*, *robust optimization*, *control with a set membership description of the uncertainty*, and *games against nature*. In this book, we will be using the minimax control name.

$$J_k^*(x_k) = \min_{u_k \in U(x_k)} \max_{w_k \in W_k(x_k,u_k)} \Big[g_k(x_k,u_k,w_k) + J_{k+1}^*\big(f_k(x_k,u_k,w_k)\big) \Big]. \tag{2.97}$$

This algorithm can be explained by using a principle of optimality type of argument. In particular, we consider the tail subproblem whereby we are at state x_k at time k, and we wish to minimize the "cost-to-go"

$$\max_{\substack{w_t \in W_t(x_t,\mu_t(x_t)) \\ t=k,k+1,\ldots,N-1}} \left[g_N(x_N) + \sum_{t=k}^{N-1} g_t\big(x_t,\mu_t(x_t),w_t\big) \right].$$

We argue that if $\pi^* = \{\mu_0^*, \mu_1^*, \ldots, \mu_{N-1}^*\}$ is an optimal policy for the minimax problem, then the tail of the policy $\{\mu_k^*, \mu_{k+1}^*, \ldots, \mu_{N-1}^*\}$ is optimal for the tail subproblem. The optimal cost of this subproblem is $J_k^*(x_k)$, as given by the DP algorithm (2.96)-(2.97). The algorithm expresses the intuitive fact that when at state x_k at time k, then regardless of what happened in the past, we should choose u_k that minimizes the worst/maximum value over w_k of the sum of the current stage cost plus the optimal cost of the tail subproblem that starts from the next state. This argument requires a mathematical proof, which turns out to involve a few fine points. For a detailed mathematical derivation, we refer to the author's textbook [Ber17a], Section 1.6. However, the DP algorithm (2.96)-(2.97) is correct assuming finite state and control spaces, among other cases.

Approximation in Value Space and Minimax Rollout

The approximation ideas for stochastic optimal control are also relevant within the minimax context. In particular, approximation in value space with one-step lookahead applies at state x_k a control

$$\tilde{u}_k \in \arg \min_{u_k \in U_k(x_k)} \max_{w_k \in W_k(x_k,u_k)} \Big[g_k(x_k,u_k,w_k) + \tilde{J}_{k+1}\big(f_k(x_k,u_k,w_k)\big) \Big], \tag{2.98}$$

where $\tilde{J}_{k+1}(x_{k+1})$ is an approximation to the optimal cost-to-go $J_{k+1}^*(x_{k+1})$ from state x_{k+1}.

Rollout is obtained when this approximation is the tail cost of some base policy $\pi = \{\mu_0, \ldots, \mu_{N-1}\}$:

$$\tilde{J}_{k+1}(x_{k+1}) = J_{k+1,\pi}(x_{k+1}).$$

Given π, we can compute $J_{k+1,\pi}(x_{k+1})$ by solving a *deterministic maximization* DP problem with the disturbances $w_{k+1}, \ldots, w_{N-1}$ playing the role of "optimization variables/controls." For finite state, control, and disturbance spaces, this is a longest path problem defined on an acyclic graph, since the control variables $u_{k+1}, \ldots, u_{N-1}$ are determined by the base policy. It is then straightforward to implement rollout: at x_k we generate all next states of the form

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

corresponding to all possible values of $u_k \in U_k(x_k)$ and $w_k \in W_k(x_k, u_k)$. We then run the maximization/longest path problem described above to compute $\tilde{J}_{k+1}(x_{k+1})$ from each of these possible next states x_{k+1}. Finally, we obtain the rollout control $\tilde{u}_k$ by solving the minimax problem in Eq. (2.98). Moreover, it is possible to use truncated rollout to approximate the tail cost of the base policy.†

Note that like all rollout algorithms, the minimax rollout algorithm is well-suited for on-line replanning in problems where data may be changing or may be revealed during the process of control selection.

We mentioned earlier that deterministic problems allow a more general form of rollout, whereby we may use a base heuristic that need not be a legitimate policy, i.e., it need not be sequentially consistent. For cost improvement it is sufficient that the heuristic be sequentially improving. A similarly more general view of rollout is not easily constructed for stochastic problems, but is possible for minimax control.

In particular, suppose that at any state x_k there is a heuristic that generates a sequence of feasible controls and disturbances, and corresponding states,

$$\{u_k, w_k, x_{k+1}, u_{k+1}, w_{k+1}, x_{k+2}, \ldots, u_{N-1}, w_{N-1}, x_N\},$$

with corresponding cost

$$H_k(x_k) = g_k(x_k, u_k, w_k) + \cdots + g_{N-1}(x_{N-1}, u_{N-1}, w_{N-1}) + g_N(x_N).$$

Then the rollout algorithm applies at state x_k a control

$$\tilde{u}_k \in \arg \min_{u_k \in U_k(x_k)} \max_{w_k \in W_k(x_k, u_k)} \Big[g_k(x_k, u_k, w_k) + H_{k+1}\big(f_k(x_k, u_k, w_k)\big) \Big].$$

This does not preclude the possibility that the disturbances $w_k, \ldots, w_{N-1}$ are chosen by an antagonistic opponent, but allows more general choices of disturbances, obtained for example, by some form of approximate maximization. For example, when the disturbance involves multiple components, $w_k = (w_k^1, \ldots, w_k^m)$, corresponding to multiple opponent agents, *the heuristic may involve an agent-by-agent maximization strategy*.

The sequential improvement condition, similar to the deterministic case, is that for all x_k and k,

$$\min_{u_k \in U_k(x_k)} \max_{w_k \in W_k(x_k, w_k)} \Big[g_k(x_k, u_k, w_k) + H_{k+1}\big(f_k(x_k, u_k, w_k)\big) \Big] \leq H_k(x_k).$$

It guarantees cost improvement, i.e., that for all x_k and k, the rollout policy

$$\tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$$

† For a more detailed discussion of this implementation, see the author's paper [Ber19b] (Section 5.4).

satisfies

$$J_{k,\tilde{\pi}}(x_k) \leq H_k(x_k).$$

Thus, generally speaking, minimax rollout is fairly similar to rollout for deterministic as well as stochastic DP problems. The main difference with deterministic (or stochastic) problems is that to compute the Q-factor of a control u_k, we need to solve a maximization problem, rather than carry out a deterministic (or Monte-Carlo, respectively) simulation with the given base policy.

Example 2.12.1 (Pursuit-Evasion Problems)

Consider a pursuit-evasion problem with state $x_k = (x_k^1, x_k^2)$, where x_k^1 is the location of the minimizer/pursuer and x_k^2 is the location of the maximizer/evader, at stage k, in a (finite node) graph defined in two- or three-dimensional space. There is also a cost-free and absorbing termination state that consists of a subset of pairs (x^1, x^2) that includes all pairs with $x^1 = x^2$. The pursuer chooses one out of a finite number of actions $u_k \in U_k(x_k)$ at each stage k, when at state x_k, and if the state is x_k and the pursuer selects u_k, the evader may choose from a known set $X_{k+1}(x_k, u_k)$ of next states x_{k+1}, which depends on (x_k, u_k). The objective of the pursuer is to minimize a nonnegative terminal cost $g(x_N^1, x_N^2)$ at the end of N stages (or reach the termination state, which has cost 0 by assumption). A reasonable base policy for the pursuer can be precomputed by DP as follows: given the current (nontermination) state $x_k = (x_k^1, x_k^2)$, make a move along the path that starts from x_k^1 and minimizes the terminal cost after $N - k$ stages, under the assumption that the evader will stay motionless at his current location x_k^2. (In a variation of this policy, the DP computation is done under the assumption that the evader will follow some nominal sequence of moves.)

For the on-line computation of the rollout control, we need the maximal value of the terminal cost that the evader can achieve starting from every $x_{k+1} \in X_{k+1}(x_k, u_k)$, assuming that the pursuer will follow the base policy (which has already been computed). We denote this maximal value by $\tilde{J}_{k+1}(x_{k+1})$. The required values $\tilde{J}_{k+1}(x_{k+1})$ can be computed by an $(N-k)$-stage DP computation involving the optimal choices of the evader, while assuming the pursuer uses the (already computed) base policy. Then the rollout control for the pursuer is obtained from the minimization

$$\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \max_{x_{k+1} \in X_{k+1}(x_k, u_k)} \tilde{J}_{k+1}(x_{k+1}).$$

Note that the preceding algorithm can be adapted for the imperfect information case where the pursuer knows x_k^2 imperfectly. This is possible by using a form of assumed certainty equivalence: the pursuer's base policy and the evader's maximization can be computed by using an estimate of the current location x_k^2 instead of the unknown true location.

In the preceding pursuit-evasion example, the choice of the base policy was facilitated by the special structure of the problem. Generally, however,

finding a suitable base policy that can be conveniently implemented is an important problem-dependent issue.

Variants of Minimax Rollout

Several of the variants of rollout discussed earlier have analogs in the minimax context, e.g., truncation with terminal cost approximation, multistep and selective step lookahead, and multiagent rollout. In particular, in the ℓ-step lookahead variant, we solve the ℓ-stage problem

$$\min_{u_k,\mu_{k+1},\ldots,\mu_{k+\ell-1}} \max_{\substack{w_k\in W_k(x_k,u_k)\\ w_t\in W_t(x_t,\mu_t(x_t))\\ t=k+1,\ldots,N-1}} \Big\{ g_k(x_k,u_k,w_k) + \sum_{t=k+1}^{k+\ell-1} g_t\big(x_t,\mu_t(x_t),w_t\big) + H_{k+\ell}(x_{k+\ell}) \Big\},$$

we find an optimal solution $\tilde{u}_k, \tilde{\mu}_{k+1}, \ldots, \tilde{\mu}_{k+\ell-1}$, and we apply the first component $\tilde{u}_k$ of that solution. As an example, this type of problem is solved at each move of chess programs like AlphaZero, where the terminal cost function is encoded through a position evaluator. In fact when multistep lookahead is used, special techniques such as *alpha-beta pruning* may be used to accelerate the computations by eliminating unnecessary portions of the lookahead graph. These techniques are well-known in the context of the two-person computer game methodology, and are used widely in games such as chess.

It is interesting to note that, contrary to the case of stochastic optimal control, there is an on-line *constrained form of rollout* for minimax control. Here there are some additional trajectory constraints of the form

$$(x_0, u_0, \ldots, u_{N-1}, x_N) \in C,$$

where C is an arbitrary set. The modification needed is similar to the one of Section 6.6: at partial trajectory

$$\tilde{y}_k = (\tilde{x}_0, \tilde{u}_0, \ldots, \tilde{u}_{k-1}, \tilde{x}_k),$$

generated by rollout, we use a heuristic with cost function H_{k+1} to compute the Q-factor

$$\tilde{Q}_k(\tilde{x}_k,u_k) = \max_{w_k,\ldots,w_{N-1}} \Big[g_k(\tilde{x}_k,u_k,w_k) + H_{k+1}\big(f_k(\tilde{x}_k,u_k,w_k),w_{k+1},\ldots,w_{N-1}\big) \Big]$$

for each u_k in the set $\tilde{U}_k(\tilde{y}_k)$ that guarantee feasibility [we can check feasibility here by running some algorithm that verifies whether the future

disturbances $w_k, \ldots, w_{N-1}$ can be chosen to violate the constraint under the base policy, starting from $(\tilde{y}_k, u_k)$]. Once the set of "feasible controls" $\tilde{U}_k(\tilde{y}_k)$ is computed, we can obtain the rollout control by the Q-factor minimization:

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

We may also use fortified versions of the unconstrained and constrained rollout algorithms, which guarantee a feasible cost-improved rollout policy. This requires the assumption that the base heuristic at the initial state produces a trajectory that is feasible for all possible disturbance sequences. Similar to the deterministic case, there are also truncated and multiagent versions of the minimax rollout algorithm.

Example 2.12.2 (Multiagent Minimax Rollout)

Let us consider a minimax problem where the minimizer's choice involves the collective decision of m agents, $u = (u^1, \ldots, u^m)$, with u^ℓ corresponding to agent ℓ, and constrained to lie within a finite set U^ℓ. Thus u must be chosen from within the set

$$U = U^1 \times \ldots \times U^m,$$

which is finite but grows exponentially in size with m. The maximizer's choice w is constrained to belong to a finite set W. We consider multiagent rollout for the minimizer, and for simplicity, we focus on a two-stage problem. However, there are straightforward extensions to a more general multistage framework.

In particular, we assume that the minimizer knowing an initial state x_0, chooses $u = (u^1, \ldots, u^m)$, with $u^\ell \in U^\ell$, $\ell = 1, \ldots, m$, and a state transition

$$x_1 = f_0(x_0, u)$$

occurs with cost $g_0(x_0, u)$. Then the maximizer, knowing x_1, chooses $w \in W$, and a terminal state

$$x_2 = f_1(x_1, w)$$

is generated with cost

$$g_1(x_1, w) + g_2(x_2).$$

The problem is to select $u \in U$, to minimize

$$g_0(x_0, u) + \max_{w \in W} \big[g(x_1, w) + g_2(x_2)\big].$$

The exact DP algorithm for this problem is given by

$$J_1^*(x_1) = \max_{w \in W} \Big[g_1(x_1, w) + g_2\big(f_1(x_1, w)\big)\Big],$$

$$J_0^*(x_0) = \min_{u \in U} \Big[g_0(x_0, u) + J_1^*\big(f_0(x_0, u)\big)\Big].$$

This DP algorithm is computationally intractable for large m. The reason is that the set of possible minimizer choices u grows exponentially with m, and for each of these choices the value of $J_1^*\big(f_0(x_0,u)\big)$ must be computed.

However, the problem can be solved approximately with multiagent rollout, using a base policy $\mu = (\mu^1,\ldots,\mu^m)$. Then the number of times $J_1^*\big(f_0(x_0,u)\big)$ needs to be computed is dramatically reduced. This computation is done sequentially, one-agent-at-a-time, as follows:

$$\tilde{u}^1 \in \arg\min_{u^1\in U^1}\Big[g_0\big(x_0,u^1,\mu^2(x_0),\ldots,\mu^m(x_0)\big) + J_1^*\Big(f_0\big(x_0,u^1,\mu^2(x_0),\ldots,\mu^m(x_0)\big)\Big)\Big],$$

$$\tilde{u}^2 \in \arg\min_{u^2\in U^2}\Big[g_0\big(x,\tilde{u}^1,u^2,\mu^3(x_0),\ldots,\mu^m(x_0)\big) + J_1^*\Big(f_0\big(x_0,\tilde{u}^1,u^2,\mu^3(x_0),\ldots,\mu^m(x_0)\big)\Big)\Big],$$

$$\cdots \qquad \cdots \qquad \cdots \qquad \cdots$$

$$\tilde{u}^m \in \arg\min_{u^m\in U^m}\Big[g_0\big(x_0,\tilde{u}^1,\tilde{u}^2,\ldots,\tilde{u}^{m-1},u^m\big) + J_1^*\Big(f_0\big(x_0,\tilde{u}^1,\tilde{u}^2,\ldots,\tilde{u}^{m-1},u^m\big)\Big)\Big].$$

In this algorithm, the number of times for which $J_1^*\big(f_0(x_0,u)\big)$ must be computed grows linearly with m.

When the number of stages is larger than two, a similar algorithm can be used. Essentially, the one-stage maximizer's cost function J_1^* must be replaced by the optimal cost function of a multistage maximization problem, where the minimizer is constrained to use the base policy (see also the paper [Ber19b], Section 5.4).

An interesting question is how do various algorithms work when approximations are used in the min-max and max-min problems? We can certainly improve the minimizer's policy *assuming a fixed policy for the maximizer*. However, it is unclear how to improve both the minimizer's and the maximizer's policies simultaneously. In practice, in *symmetric games*, like chess, a common policy is trained for both players. In particular, in the AlphaZero and TD-Gammon programs this strategy is computationally expedient and has worked well. However, there is no reliable theory to guide the simultaneous training of policies for both maximizer and minimizer, and it is quite plausible that unusual behavior may arise in exceptional cases.† Even *exact* policy iteration methods for Markov games

† Indeed such exceptional cases have been reported for the AlphaGo program in late 2022, when humans defeated an AlphaGo look-alike, KataGo, "by using adversarial techniques that take advantage of KataGo's blind spots" (according to the reports); see Wang et al. [WGB22].

encounter serious convergence difficulties, and need to be modified for reliable behavior. The author's paper [Ber21c] and book [Ber22b] (Chapter 5) address these convergence issues with modified versions of the policy iteration method, and give many earlier references.

We finally note another source of difficulty in minimax control: Newton's method applied to solution of the Bellman equation for minimax problems exhibits more complex behavior than its expected value counterpart. The reason is that the Bellman operator T for infinite horizon problems, given by

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

is neither convex nor concave as a function of J. To see this, note that the function

$$\max_{w \in W(x,u)} \Big[g(x,u,w) + \alpha J\big(f(x,u,w)\big) \Big],$$

viewed as a function of J [for fixed (x,u)], is convex, and when minimized over $u \in U(x)$, it becomes neither convex nor concave. As a result there are special difficulties in connection with convergence of Newton's method and the natural form of policy iteration, given by Pollatschek and Avi-Itzhak [PoA69]; see also Chapter 5 of the author's abstract DP book [Ber22a].

Minimax Control and Zero-Sum Game Theory

Zero-sum game problems are viewed as fundamental in the field of economics, and there is an extensive and time-honored theory around them. In the case where the game involves a dynamic system

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

and a cost function

$$g_k(x_k, u_k, w_k),$$

there are two players, the minimizer choosing $u_k \in U_k(x_k)$, and the maximizer choosing $w_k \in W_k(x_k)$, at each stage k. Such zero-sum games involve *two* minimax control problems:

(a) The *min-max problem*, where the minimizer chooses a policy first and the maximizer chooses a policy second with knowledge of the minimizer's policy. The DP algorithm for this problem has the form

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

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

(b) The *max-min problem*, where the maximizer chooses policy first and the minimizer chooses policy second with knowledge of the maximizer's policy. The DP algorithm for this problem has the form

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

$$\hat{J}_k(x_k) = \max_{w_k \in W_k(x_k)} \min_{u_k \in U_k(x_k)} \Big[g_k(x_k, u_k, w_k) + \hat{J}_{k+1}\big(f_k(x_k, u_k, w_k)\big) \Big].$$

A basic and easily seen fact is that

$$\text{Max-Min optimal value} \leq \text{Min-Max optimal value}.$$

Game theory is particularly interested on conditions that guarantee that

$$\text{Max-Min optimal value} = \text{Min-Max optimal value}. \tag{2.99}$$

However, this question is of limited interest in engineering contexts that involve worst case design. Moreover, the validity of the minimax equality (2.99) is beyond the range of practical RL. This is so primarily because once approximations are introduced, the delicate assumptions that guarantee this equality are disrupted.

2.13 NOTES, SOURCES, AND EXERCISES

Section 2.1: In this chapter, we have placed emphasis on finite horizon problems, possibly involving a nonstationary system and cost per stage. However, the insights that can be obtained from the infinite horizon/stationary context fully apply. These include the interpretation of approximation in value space as a Newton step, and of rollout as a single step of the policy iteration method. The reason is that an N-step finite horizon/nonstationary problem can be converted to an infinite horizon/stationary problem with a termination state to which the system moves at the Nth stage; see Section 1.6.2.

Section 2.2: Approximation in value space has been considered in an ad hoc manner since the early days of DP, motivated by the curse of dimensionality. Moreover, the idea of ℓ-step lookahead minimization with horizon truncation beyond the ℓ steps has a long history and is often referred to as "rolling horizon" or "receding horizon" optimization. Approximation in value space was reframed in the late 80s and was coupled with model-free simulation methods that originated in artificial intelligence.

Section 2.3: The main idea of rollout algorithms, obtaining an improved policy starting from some other suboptimal policy, has appeared in several DP contexts, including games; see e.g., Abramson [Abr90], and Tesauro

and Galperin [TeG96]. The name "rollout" was coined by Tesauro [TeG96] in the context of backgammon; see Example 2.7.3. The use of the name "rollout" has gradually expanded beyond its original context; for example the samples collected through trajectory simulation are referred to as "rollouts" by some authors.

In the present notes, we will adopt the original intended meaning: rollout is an algorithm that provides policy improvement starting from a base policy, which is evaluated with some form of Monte Carlo simulation, perhaps augmented by some other calculation that may include a terminal cost function approximation. The author's rollout book [Ber20a] provides a more extensive discussion.

There has been a lot of research on rollout algorithms, which we list selectively in chronological order: Christodouleas [Chr97], Bertsekas and Castañon [BeC99], Duin and Voss [DuV99], Secomandi [Sec00], [Sec01], [Sec03], Ferris and Voelker [FeV02], [FeV04], McGovern, Moss, and Barto [MMB02], Savagaonkar, Givan, and Chong [SGC02], Wu, Chong, and Givan [WCG02], [WCG03], Bertsimas and Popescu [BeP03], Guerriero and Mancini [GuM03], Tu and Pattipati [TuP03], Meloni, Pacciarelli, and Pranzo [MPP04], Yan et al. [YDR04], Han, Lai, and Spivakovsky [HLS06], Besse and Chaib-draa [BeC08], Sun et al. [SZL08], Mishra et al. [MCT10], Bertazzi et al. [BBG13], Sun et al. [SLJ13], Tesauro et al. [TGL13], Antunes and Heemels [AnH14], Beyme and Leung [BeL14], Goodson, Thomas, and Ohlmann [GTO15], [GTO17], Khashooei, Antunes, and Heemels [KAH15], Li and Womer [LiW15], Mastin and Jaillet [MaJ15], Huang, Jia, and Guan [HJG16], Simroth, Holfeld, and Brunsch [SHB15], Lan, Guan, and Wu [LGW16], Lam, Willcox, and Wolpert [LWW16], Gommans et al. [GTA17], Lam and Willcox [LaW17], Ulmer [Ulm17], Bertazzi and Secomandi [BeS18], Zhang, Ohlmann, and Thomas [ZOT18], Sarkale et al. [SNC18], Ulmer at al. [UGM18], Arcari, Hewing, and Zeilinger [AHZ19], Chu, Xu, and Li [CXL19], Goodson, Bertazzi, and Levary [GBL19], Guerriero, Di Puglia, and Macrina [GDM19], Ho, Liu, and Zabinsky [HLZ19], Liu et al. [LLL19], Nozhati et al. [NSE19], Singh and Kumar [SiK19], Yu et al. [YYM19], Yuanhong [Yua19], Andersen, Stidsen, and Reinhardt [ASR20], Durasevic and Jakobovic [DuJ20], Issakkimuthu, Fern, and Tadepalli [IFT20], Lee et al. [LEC20], Li et al. [LZS20], Lee [Lee20], Montenegro et al. [MLM20], Meshram and Kaza [MeK20], Schope, Driessen, and Yarovoy [SDY20], Yan, Wang, and Xu [YWX20], Yue and Kontar [YuK20], Zhang, Kafouros, and Yu [ZKY20], Hoffman et al. [HCR21], Houy and Flaig [HoF21], Li, Krakow, and Gopalswamy [LKG21], Liu et al. [LPS21], Nozhati [Noz21], Rimélé et al. [RGG21], Tuncel et al. [TBP21], Xie, Li, and Xu [XLX21], Bertsekas [Ber22d], Paulson, Sonouifar, and Chakrabarty [PSC22], Bai et al. [BLJ23], Rusmevichientong et al. [RST23].

These references collectively include a large number of computational studies, discuss variants and problem-specific adaptations of rollout algorithms for a broad variety of practical problems, and consistently report

favorable computational experience. The size of the cost improvement over the base policy is often impressive, evidently owing to the fast convergence rate of Newton's method that underlies rollout. Moreover these works illustrate some of the other important advantages of rollout: reliability, simplicity, suitability for on-line replanning, and the ability to interface with other RL techniques, such as neural network training, which can be used to provide suitable base policies and/or approximations to their cost functions.

The adaptation of rollout algorithms to discrete deterministic optimization problems, the notions of sequential consistency, sequential improvement, fortified rollout, and the use of multiple heuristics for parallel rollout were first given in the paper by Bertsekas, Tsitsiklis, and Wu [BTW97], and were also discussed in the neuro-dynamic programming book [BeT96]. Rollout algorithms for stochastic problems were further formalized in the papers by Bertsekas [Ber97b], and Bertsekas and Castañon [BeC99]. Extensions to constrained rollout were first given in the author's papers [Ber05a], [Ber05b]. A survey of rollout in discrete optimization was given by the author in [Ber13a].

The model-free rollout algorithm, in the form given here, was first discussed in the RL book [Ber19a]. It is inspired by the method of comparison training, proposed by Tesauro [Tes89a], [Tes89b], [Tes01], and subsequently used by several other authors (see [DNW16], [TCW19]). This is a general method for training an approximation architecture to choose between two alternatives, using a dataset of expert choices in place of an explicit cost function.

Section 2.4: Our discussion of rollout, iterative deepening, and pruning in the context of multistep approximation in value space for deterministic problems contains some original ideas particularly in connection with the incremental multistep rollout algorithms.

Section 2.5: Constrained forms of rollout were introduced in the author's papers [Ber05a] and [Ber05b]. The paper [Ber05a] also discusses rollout and approximation in value space for stochastic problems in the context of so-called *restricted structure policies*. The idea here is to simplify the problem by selectively restricting the information and/or the controls available to the controller, thereby obtaining a restricted but more tractable problem structure, which can be used conveniently in a one-step lookahead context. An example of such a structure is one where fewer observations are obtained, or one where the control constraint set is restricted to a single or a small number of given controls at each state.

Section 2.6: Rollout for continuous-time optimal control was first discussed in the author's rollout book [Ber20a]. A related discussion of policy iteration, including the motivation for approximating the gradient of the optimal cost-to-go $\nabla_x J_t$ rather than the optimal cost-to-go J_t, has been

given in Section 6.11 of the neuro-dynamic programming book [BeT96]. This discussion also includes the use of value and policy networks for approximate policy evaluation and policy improvement for continuous-time optimal control. The underlying ideas have long historical roots, which are recounted in detail in the book [BeT96].

Section 2.7: The idea of the certainty equivalence approximation in the context of rollout for stochastic systems (Section 2.7.3) was proposed in the paper by Bertsekas and Castañon [BeC99], together with extensive empirical justification. However, the associated theoretical insight into this idea was established more recently, through the interpretation of approximation in value space as a Newton step, which suggests that the lookahead minimization after the first step can be approximated with small degradation of performance.

The idea of variance reduction in the context of rollout (Section 2.7.4) was proposed by the author in the paper [Ber97b]; see also the DP textbook [Ber17a], Section 6.5.2. The paper by Chang, Hu, Fu, and Marcus [CHF05], and the 2007 first edition of their monograph proposed and analyzed adaptive sampling in connection with DP, as well as early forms of Monte Carlo tree search, including statistical tests to control the sampling process (a second edition, [CHF13], appeared in 2013). The name "Monte Carlo tree search" has become popular, and in its current use, it encompasses a variety of methods that involve adaptive sampling, rollout, and extensions to sequential games. We refer to the papers by Coulom [Cou06], and Chang et al. [CHF13], the discussion by Fu [Fu17], and the survey by Browne et al. [BPW12].

Statistical tests for adaptive sampling has been inspired by works on multiarmed bandit problems; see Lai and Robbins [LaR85], Agrawal [Agr95], Burnetas and Katehakis [BuK97], Meuleau and Bourgine [MeB99], Auer, Cesa-Bianchi, and Fischer [ACF02], Kocsis and Szepesvari [KoS06], Dimitrakakis and Lagoudakis [DiL08], Audibert, Munos, and Szepesvari [AMS09], and Munos [Mun14]. The book by Lattimore and Szepesvari [LaS20] focuses on multiarmed bandit methods, and provides an extensive account of the UCB rule.

Adaptive sampling and MCTS may be viewed within the context of a broader class of on-line lookahead minimization techniques, sometimes called *on-line search* methods. These techniques are based on a variety of ideas, such as random search and intelligent pruning of the lookahead tree. One may naturally combine them with approximation in value space and (possibly) rollout, although it is not necessary to do so (the multistep minimization horizon may extend to the terminal time N). For representative works, some of which apply to continuous spaces problems, including POMDP, see Hansen and Zilberstein [HaZ01], Kearns, Mansour, and Ng [KMN02], Peret and Garcia [PeG04], Ross et al. [RPP08], Silver and Veness [SiV10], Hostetler, Fern, and Dietterich [HFD17], and Ye et al. [YSH17].

The multistep lookahead approximation ideas of Section 2.4 may also be viewed within the context of on-line search methods.

Another rollout idea for stochastic problems, which we have not discussed in these notes, is the *open-loop feedback controller* (OLFC), a suboptimal control scheme that dates to the 60s; see Dreyfus [Dre65]. The OLFC applies to POMDP as well, and uses an open-loop optimization over the future evolution of the system. In particular, it uses the current information vector I_k to determine the belief state b_k. It then solves the open-loop problem of minimizing

$$E\left\{ g_N(x_N) + \sum_{i=k}^{N-1} g_i(x_i, u_i, w_i) \;\Big|\; I_k \right\}$$

subject to the constraints

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

and applies the first control $\overline{u}_k$ in the optimal open-loop control sequence $\{\overline{u}_k, \overline{u}_{k+1}, \ldots, \overline{u}_{N-1}\}$. It is easily seen that the OLFC is a rollout algorithm that uses as base policy the optimal open-loop policy for the problem (the one that ignores any state or observation feedback).

For a detailed discussion of the OLFC, we refer to the author's survey paper [Ber05a] (Section 4) and DP textbook [Ber17a] (Section 6.4.4). The survey [Ber05a] discusses also a generalization of the OLFC, called *partial open-loop-feedback-control*, which calculates the control input on the basis that *some* (but not necessarily all) of the observations will in fact be taken in the future, and the remaining observations will not be taken. This method often allows one to deal with those observations that are troublesome and complicate the solution, while taking into account the future availability of other observations that can be reasonably dealt with. A computational case study for hydrothermal power system scheduling is given by Martinez and Soares [MaS02]. A variant of the OLFC, which also applies to minimax control problems, is given in the author's paper [Ber72b], together with a proof of a cost improvement property over the optimal open-loop policy.

Section 2.8: The role of stochastic programming in providing a link between stochastic DP and continuous spaces deterministic optimization (cf. Section 2.8) is well known; see the texts by Birge and Louveaux [BiL97], Kall and Wallace [KaW94], and Prekopa [Pre95], and the survey by Ruszczynski and Shapiro [RuS03]. Stochastic programming has been applied widely, and there is much to be gained from its combination with RL. The material of this section comes from the author's rollout book [Ber20a].

Section 2.9: The multiagent rollout algorithm was proposed in the author's papers [Ber19c], [Ber20b]. The paper [Ber21a] provides an extensive overview of this research. See also the notes and sources for Chapter 1.

Section 2.10: The material on rollout for Bayesian optimization and sequential estimation comes from a recent paper by the author [Ber22d]. This paper is also the basis for the adaptive control material of Section 2.11, and has been included in the book [Ber22a]. The paper by Bhambri, Bhattacharjee, and Bertsekas [BBB22] discusses this material for the case of a deterministic system, applies rollout to sequential decoding in the context of the challenging Wordle puzzle, and provides an implementation using some popular base heuristics, with performance that is very close to optimal. For related work see Loxley and Cheung [LoC23].

Section 2.11: The POMDP framework for adaptive control dates to the 60s, and has stimulated substantial theoretical investigations; see Mandl [Man74], Doshi and Shreve [DoS80], Kumar and Lin [KuL82], and the survey by Kumar [Kum85]. Some of the pitfalls of performing parameter identification while simultaneously applying adaptive control have been described by Borkar and Varaiya [BoV79], and by Kumar [Kum83]; see [Ber17a], Section 6.8 for a related discussion.

Section 2.12: The treatment of sequential minimax problems by DP (cf. Section 2.12) has a long history. For some early influential works, see Blackwell and Girshick [BlG54], Shapley [Sha53], and Witsenhausen [Wit66]. In minimax control problems, the maximizer is assumed to make choices with perfect knowledge of the minimizer's policy. If the roles of maximizer and minimizer are reversed, i.e., the maximizer has a policy (a sequence of functions of the current state) and the minimizer makes choices with perfect knowledge of that policy, the minimizer gains an advantage, the problem may genuinely change, and the optimal value may be reduced. Thus "min-max" and "max-min" are generally two different problems. In classical two-person zero-sum game theory, however, the main focus is on situations where the min-max and max-min are equal. By contrast, in engineering worst case design contexts, the min-max and max-min values are typically unequal.

There is substantial literature on sequential zero-sum games in the context of DP, often called *Markov games*. The classical paper by Shapley [Sha53] addresses discounted infinite horizon games. A PI algorithm for finite-state Markov games was proposed by Pollatschek and Avi-Itzhak [PoA69], and was interpreted as a Newton method for solving the associated Bellman equation. They have also shown that the algorithm may not converge to the optimal cost function. Computational studies have verified that the Pollatschek and Avi-Itzhak algorithm converges much faster than its competitors, *when it converges* (see Breton et al. [BFH86], and also Filar and Tolwinski [FiT91], who proposed a modification of the algorithm). Related methods have been discussed for Markov games by van der Wal [Van78] and Tolwinski [Tol89]. The paper by Raghavan and Filar [RaF91], and the textbook by Filar and Vrieze [FiV96] provide extensive surveys of the research up to that time.

The paper by Yu [Yu14] provides an analysis of stochastic shortest path games, where the termination state may not be reachable under some policies, following the earlier paper by Patek and Bertsekas [PaB99]. The paper [Yu14] also includes a rigorous analysis of the Q-learning algorithm for stochastic shortest path games (without any cost function approximation). The papers by Perolat et al. [PSP15], [PPG16], and the survey by Zhang, Yang, and Basar [ZYB21] discuss alternative RL methods for games. The author's paper [Ber19b] develops VI, PI, and Dijkstra-like finitely terminating algorithms for exact solution of shortest path minimax problems. It also discusses related rollout algorithms for approximate solution.

The author's paper [Ber21b] has explained the reason behind the unreliable behavior of the Pollatschek and Avi-Itzhak algorithm, based on the Newton step interpretation of PI given in Chapter 1: in the case of Markov games, the Bellman operator does not have the concavity property that is typical of one-player games. This paper has also provided a modified algorithm with solid convergence properties under a totally asynchronous implementation, which applies to very general types of sequential zero-sum games and minimax control. Related aggregation-based RL algorithms were also given. The algorithms, variations, and analysis of the paper [Ber21b] were incorporated as Chapter 5 in the 3rd edition of the abstract DP book [Ber22b].

E X E R C I S E S

2.1 (A Traveling Salesman Rollout Example with a Sequentially Improving Heuristic)

Consider the traveling salesman problem of Example 1.2.3 and Fig. 1.2.11, and the rollout algorithm starting from city A.

(a) Assume that the base heuristic is chosen to be the farthest neighbor heuristic, which completes a partial tour by successively moving to the farthest neighbor city not visited thus far. Show that this base heuristic is sequentially consistent. What are the tours produced by this base heuristic and the corresponding rollout algorithm? *Answer*: The base heuristic will produce the tour A→AD→ADB→ADBC→A with cost 45. The rollout algorithm will produce the tour A→AB→ABD→ABDC→A with cost 13.

(b) Assume that the base heuristic at city A is the nearest neighbor heuristic, while at the partial tours AB, AC, and AD it is the farthest neighbor heuristic. Show that this base heuristic is sequentially improving but not sequentially consistent. Compute the final tour generated by rollout.

Solution of part (b): Clearly the base heuristic is not sequentially consistent, since from A it generates

$$\text{A}\rightarrow\text{AC}\rightarrow\text{ACD}\rightarrow\text{ACDB}\rightarrow\text{A},$$

but from AC it generates

$$\text{AC}\rightarrow\text{ACB}\rightarrow\text{ACBD}\rightarrow\text{A}.$$

However, it is seen that the sequential improvement criterion (2.15) holds at each of the states A, AB, AC, and AD (and also trivially for the remaining states).

The base heuristic at A is the nearest neighbor heuristic so it generates

$$\text{A}\rightarrow\text{AC}\rightarrow\text{ACD}\rightarrow\text{ACDB}\rightarrow\text{A with cost 28.}$$

The rollout algorithm at state A looks at the three successor states AB, AC, AD, and runs the farthest neighbor heuristic from each, and generates:

$$\text{A}\rightarrow\text{AB}\rightarrow\text{ABD}\rightarrow\text{ABDC}\rightarrow\text{A with cost 13,}$$

$$\text{A}\rightarrow\text{AC}\rightarrow\text{ACB}\rightarrow\text{ACBD}\rightarrow\text{A with cost 45,}$$

$$\text{A}\rightarrow\text{AD}\rightarrow\text{ADB}\rightarrow\text{ADBC}\rightarrow\text{A with cost 45,}$$

so the rollout algorithm will move from A to AB.

Then the rollout algorithm looks at the two successor states ABC, ABD, and runs the base heuristic (whatever that may be; it does not matter) from each. The paths generated are:

$$\text{AB}\rightarrow\text{ABC}\rightarrow\text{ABCD}\rightarrow\text{A with cost 26,}$$

AB→ABD→ABDC→A with cost 13,

so the rollout algorithm will move from AB to ABD.

Thus the final tour generated by the rollout algorithm is

A→AB→ABD→ABDC→A, with cost 13.

2.2 (A Generic Example of a Base Heuristic that is not Sequentially Improving)

Consider a rollout algorithm for a deterministic problem with a base heuristic that produces an optimal control sequence at the initial state x_0, and uses the (optimal) first control u_0 of this sequence to move to the (optimal) next state x_1. Suppose that the base heuristic produces a strictly suboptimal sequence from every successor state $x_2 = f_1(x_1, u_1)$, $u_1 \in U_1(x_1)$, so that the rollout yields a control u_1 that is strictly suboptimal. Show that the trajectory produced by the rollout algorithm starting from the initial state x_0 is strictly inferior to the one produced by the base heuristic starting from x_0, while the sequential improvement condition does not hold.

2.3 (Computational Exercise - Parking with Problem Approximation and Rollout)

In this computational exercise we consider a more complex, imperfect state information version of the one-directional parking problem of Example 1.6.1. Recall that in this problem a driver is looking for a free parking space in an area consisting of N spaces arranged in a line, with a garage at the end of the line (space N). The driver starts at space 0 and traverses the parking spaces sequentially, i.e., from each space he/she goes to the next space, up to when he/she decides to park in space k at cost $c(k)$, if space k is free. Upon reaching the garage, parking is mandatory at cost C.

In Example 1.6.1, we assumed that the driver knows the probabilities $p(k+1), \ldots, p(N-1)$ of the parking spaces $(k+1), \ldots, (N-1)$, respectively, being free. Under this assumption, the state at stage k is either the termination state t (if already parked), or it is F (location k free), or it is $\overline{F}$ (location k taken), and the DP algorithm has the form

$$J_k^*(F) = \begin{cases} \min\Big[c(k),\, p(k+1)J_{k+1}^*(F) + \big(1-p(k+1)\big)J_{k+1}^*(\overline{F})\Big] & \text{if } k < N-1, \\ \min\big[c(N-1),\, C\big] & \text{if } k = N-1, \end{cases} \tag{2.100}$$

$$J_k^*(\overline{F}) = \begin{cases} p(k+1)J_{k+1}^*(F) + \big(1-p(k+1)\big)J_{k+1}^*(\overline{F}) & \text{if } k < N-1, \\ C & \text{if } k = N-1, \end{cases} \tag{2.101}$$

for the states other than the termination state t, while for t we have $J_k^*(t) = 0$ for all k.

We will now consider the more complex variant of the problem where the probabilities $p(0), \ldots, p(N-1)$ do not change over time, but are unknown to

the driver, so that he/she cannot use the exact DP algorithm (2.100)-(2.101). Instead, the driver considers a one-step lookahead approximation in value space scheme, which uses empirical estimates of these probabilities that are based on the ratio $\frac{f_k}{k+1}$, where f_k is the number of free spaces seen up to space k, after the free/taken status of spaces $0,\ldots,k$ has been observed. In particular, these empirical estimates are given by

$$b_k(m,f_k)=\gamma\bar{p}(m)+(1-\gamma)\frac{f_k}{k+1},\qquad m=k+1,\ldots,N-1, \tag{2.102}$$

where f_k is the number of free spaces seen up to space k, and γ and $\bar{p}(m)$ are fixed numbers between 0 and 1. Of course the values f_k observed by the driver evolve according to the true (and unknown) probabilities $p(0),\ldots,p(N-1)$ according to

$$f_{k+1}=\begin{cases} f_k+1 & \text{with probability } p(k+1),\\ f_k & \text{with probability } 1-p(k+1).\end{cases} \tag{2.103}$$

For the solution of this exercise you may assume any reasonable values you wish for N, $p(m)$, $\bar{p}(m)$, and γ. Recommended values are $N\geq 100$, and probabilities $p(m)$ and $\bar{p}(m)$ that are nonincreasing with m.

The decision made by the approximation in value space scheme is to park at space k if and only if it is free and in addition

$$c(k)\leq b_k(k+1,f_k)\tilde{J}_{k+1}(F)+\big(1-b_k(k+1,f_k)\big)\tilde{J}_{k+1}(\overline{F}), \tag{2.104}$$

where $\tilde{J}_{k+1}(F)$ and $\tilde{J}_{k+1}(\overline{F})$ are the cost-to-go approximations from stage $k+1$. Consider the following two different methods to compute $\tilde{J}_{k+1}(F)$ and $\tilde{J}_{k+1}(\overline{F})$ for use in Eq. (2.104):

(1) Here the approximate cost function values $\tilde{J}_{k+1}(F)$ and $\tilde{J}_{k+1}(\overline{F})$ are obtained by using problem approximation, whereby at time k it is assumed that the probabilities of free/taken status at the future spaces $m=k+1,\ldots,N-1$ are $b_k(m,f_k)$, $m=k+1,\ldots,N-1$, as given by Eq. (2.102).

More specifically, $\tilde{J}_{k+1}(F)$ and $\tilde{J}_{k+1}(\overline{F})$ are obtained by solving optimally the problem whereby we use the probabilities $b_k(m,f_k)$ of Eq. (2.102) in place of the unknown $p(m)$ in the DP algorithm (2.100)-(2.101):

$$\tilde{J}_{k+1}(F)=\hat{J}_{k+1}(F),\qquad \tilde{J}_{k+1}(\overline{F})=\hat{J}_{k+1}(\overline{F}),$$

where $\hat{J}_{k+1}(F)$ and $\hat{J}_{k+1}(\overline{F})$ are given at the last step of the DP algorithm

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

$$\hat{J}_m(F)=\min\big[c(m),\,b_k(m+1,f_k)\hat{J}_{m+1}(F)+\big(1-b_k(m+1,f_k)\big)\hat{J}_{m+1}(\overline{F})\big],$$
$$\text{if } k<m<N-1,$$

$$\hat{J}_m(\overline{F})=b_k(m+1,f_k)\hat{J}_{m+1}(F)+\big(1-b_k(m+1,f_k)\big)\hat{J}_{m+1}(\overline{F}),$$
$$\text{if } k<m<N-1.$$

(2) Here for each k, the approximate cost function values $\tilde{J}_{k+1}(F)$ and $\tilde{J}_{k+1}(\overline{F})$ are obtained by using rollout with a greedy base heuristic (park as soon as possible), and Monte Carlo simulation. In particular, according to this greedy heuristic, we have $\tilde{J}_{k+1}(F) = c(k+1)$. To compute $\tilde{J}_{k+1}(\overline{F})$ we generate many random trajectories by running the greedy heuristic forward from space $k+1$ assuming the probabilities $b_k(m+1, f_k)$ of Eq. (2.102) in place of the unknown $p(m+1)$, $m = k+1, \ldots, N-1$, and we average the cost results obtained.

(a) Use Monte Carlo simulation to compute the expected cost from spaces $0, \ldots, N-1$, when using each of the two schemes (1) and (2).

(b) Compare the performance of the schemes of part (a) with the following:

(i) The optimal expected costs $J_k^*(F)$ and $J_k^*(\overline{F})$ from $k = 0, \ldots, N-1$, using the DP algorithm (2.100)-(2.101), and the probabilities $p(m)$, $m = 0, \ldots, N-1$, that you used for the random generation of the numbers of free spaces f_k [cf. Eq. (2.103)].

(ii) The expected costs $\hat{J}_k(F)$ and $\hat{J}_k(\overline{F})$ from $k = 0, \ldots, N-1$ that are attained by using the greedy base heuristic. Argue that these are given by

$$\hat{J}_k(F) = c(k), \qquad k = 0, \ldots, N-1,$$

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

$$\hat{J}_{N-1}(\overline{F}) = C.$$

(c) Argue that scheme (1) becomes superior to scheme (2) in terms of cost attained as $\gamma \approx 1$ and $\bar{p}(m) \approx p(m)$. Are your computational results in rough agreement with this assertion?

(d) Argue that as $\gamma \approx 0$ and $N >> 1$, scheme (1) becomes superior to scheme (2) in terms of cost attained from parking spaces $k >> 1$.

(e) What happens if the probabilities $p(m)$ do not change much with m?

3

Learning Values and Policies

Contents

In this chapter, we will discuss the methods and objectives of off-line training through the use of parametric approximation architectures such as neural networks. We begin with a general discussion of parametric architectures and their training in Section 3.1. We then consider the training of neural networks in Section 3.2, and their use in the context of finite horizon approximate DP in Section 3.3. In Section 3.4, we discuss the training of policies. Finally, in Section 3.5, we discuss aggregation methods.

3.1 PARAMETRIC APPROXIMATION ARCHITECTURES

As we have noted earlier, for the success of approximation in value space, it is important to select a class of lookahead function approximations $\tilde{J}_k$ that is suitable for the problem at hand. In the preceding two chapters we discussed several methods for choosing $\tilde{J}_k$, based mostly on some form of rollout. We will now discuss how $\tilde{J}_k$ can be obtained by off-line training from a parametric class of functions, possibly involving a neural network, with the parameters "optimized" with the use of some algorithm.

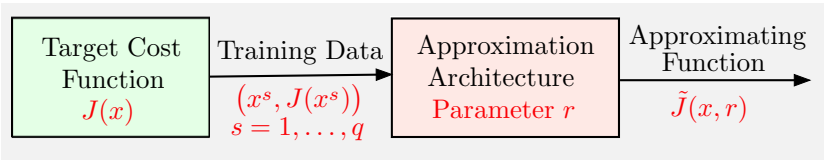


Figure 3.1.1 The general structure for parametric cost approximation. We approximate the target cost function $J(x)$ with a member from a parametric class $\tilde{J}(x, r)$ that depend on a parameter vector r. We use training data $\big(x^s, J(x^s)\big)$, $s = 1, \ldots, q$, and a form of optimization 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. 3.1.1. 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 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 aims to minimize 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

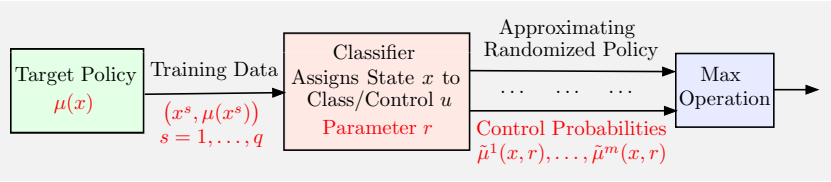


Figure 3.1.2 A general structure for parametric policy approximation for the case where the control space is finite, $U = \{u^1, \ldots, u^m\}$, and its relation to a classification scheme. It produces a randomized policy of the form (3.1), which is converted to a nonrandomized policy through the maximization operation (3.2).

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.

However, 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

$$\{\tilde{\mu}(u^1, r), \ldots, \tilde{\mu}(u^m, r)\}, \tag{3.1}$$

which is a randomized policy in our policy approximation context; see Fig. 3.1.2. 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 (3.1) can be converted to a nonrandomized policy using a maximization operation: associate x with the control of maximum probability (cf. Fig. 3.1.2),

$$\tilde{\mu}(x,r) \in \arg \max_{i=1,\ldots,m} \tilde{\mu}^i(x,r). \tag{3.2}$$

The use of classification methods for approximation in policy space will be discussed more fully in Section 3.4.

3.1.1 Cost Function Approximation

For the remainder of this section, as well as Sections 3.2 and 3.3, we will focus on approximation in value space schemes, where the approximate cost functions are selected from a parametric class of functions $\tilde{J}_k(x_k, r_k)$ that

for each k, depend on the current state x_k and a vector $r_k = (r_{1,k}, \ldots, r_{m_k,k})$ of m_k "tunable" scalar parameters. By adjusting the parameters, one can change the "shape" of $\tilde{J}_k$ so that it is a reasonably good approximation to some target function, usually the true optimal cost-to-go function J_k^*, or the cost-to-go function $J_{k,\pi}$ of some policy π. The class of functions $\tilde{J}_k(x_k, r_k)$ is called an *approximation architecture*, and the process of choosing the parameter vectors r_k is commonly called *training* or *tuning* the architecture. We will focus initially on approximation of cost functions, hence the use of the $\tilde{J}_k$ notation. In Section 3.4 we will consider the other major use of parametric approximation architectures, of the form $\tilde{\mu}_k(x_k, r_k)$, where the target function is a control function μ_k that is part of some policy.

The simplest training approach for parametric architectures is to do some form of semi-exhaustive or semi-random search in the space of parameter vectors and adopt the parameters that result in best performance of the associated one-step lookahead controller (according to some criterion). There are methods of this type that have been used primarily in cases where the number of parameters is relatively small.

Random search and Bayesian optimization methods have also been used to tune *hyperparameters* of an approximation architecture; for example, the number of layers in a neural network, or the number of clusters in the context of partitioning discrete spaces into clusters, etc. We refer to the research literature for further discussion.

Other systematic approaches are based on numerical optimization, such as a least squares fit that aims to match the cost approximation produced by the architecture to a "training set," i.e., a large number of pairs of state and cost values that are obtained through some form of sampling process. Throughout Sections 3.1-3.3 we will focus primarily on this approach.

3.1.2 Feature-Based Architectures

There is a large variety of approximation architectures, based for example on polynomials, wavelets, radial basis functions, discretization/interpolation schemes, neural networks, and others. A particularly interesting type of cost approximation involves *feature extraction*, a process that maps the state x_k into some vector $\phi_k(x_k)$, called the *feature vector* associated with x_k at time k. The vector $\phi_k(x_k)$ consists of scalar components

$$\phi_{1,k}(x_k), \ldots, \phi_{m_k,k}(x_k),$$

called *features*. A feature-based cost approximation has the form

$$\tilde{J}_k(x_k, r_k) = \hat{J}_k\big(\phi_k(x_k), r_k\big),$$

where r_k is a parameter vector and $\hat{J}_k$ is some function. Thus, the cost approximation depends on the state x_k through its feature vector $\phi_k(x_k)$.

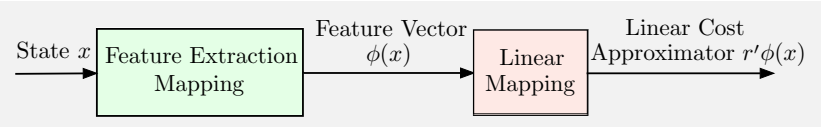


Figure 3.1.3 The structure of a linear feature-based architecture. At time k, we use a feature extraction mapping to generate an input $\phi_k(x_k)$ to a linear mapping defined by a parameter vector r_k.

Note that we are allowing for different features $\phi_k(x_k)$ and different parameter vectors r_k for each stage k. This is necessary for nonstationary problems (e.g., if the state space changes over time), and also to capture the effect of proximity to the end of the horizon. On the other hand, for stationary problems with a long or infinite horizon, where the state space does not change with k, it is common to use the same features and parameters for all stages. The subsequent discussion can easily be adapted to infinite horizon methods, as we will discuss later.

Features are often handcrafted, based on whatever human intelligence, insight, or experience is available, and are meant to capture the most important characteristics of the current state. There are also systematic ways to construct features, including the use of data and neural networks, which we will discuss shortly. In this section, we provide a brief and selective presentation of architectures.

One idea behind using features is that the optimal cost-to-go functions J_k^* may be complicated nonlinear mappings, so it is sensible to try to break their complexity into smaller, less complex pieces. In particular, if the features encode much of the nonlinearity of J_k^*, we may be able to use a relatively simple architecture $\hat{J}_k$ to approximate J_k^*. For example, with a well-chosen feature vector $\phi_k(x_k)$, a good approximation to the cost-to-go is often provided by *linearly* weighting the features, i.e.,

$$\tilde{J}_k(x_k, r_k) = \hat{J}_k\big(\phi_k(x_k), r_k\big) = \sum_{\ell=1}^{m_k} r_{\ell,k}\phi_{\ell,k}(x_k) = r_k'\phi_k(x_k), \tag{3.3}$$

where $r_{\ell,k}$ and $\phi_{\ell,k}(x_k)$ are the ℓth components of r_k and $\phi_k(x_k)$, respectively, and $r_k'\phi_k(x_k)$ denotes the inner product of r_k and $\phi_k(x_k)$, viewed as column vectors of $\Re^{m_k}$ (a prime denotes transposition, so r_k' is a row vector); see Fig. 3.1.3.

This is called a *linear feature-based architecture*, and the scalar parameters $r_{\ell,k}$ are also called *weights*. Among other advantages, these architectures admit simpler training algorithms than their nonlinear counterparts; see the NDP book [BeT96]. Mathematically, the approximating function $\tilde{J}_k(x_k, r_k)$ can be viewed as a member of the subspace spanned by the features $\phi_{\ell,k}(x_k)$, $\ell = 1, \ldots, m_k$, which for this reason are also referred to as *basis functions*. We provide a few examples, where for simplicity we drop the index k.

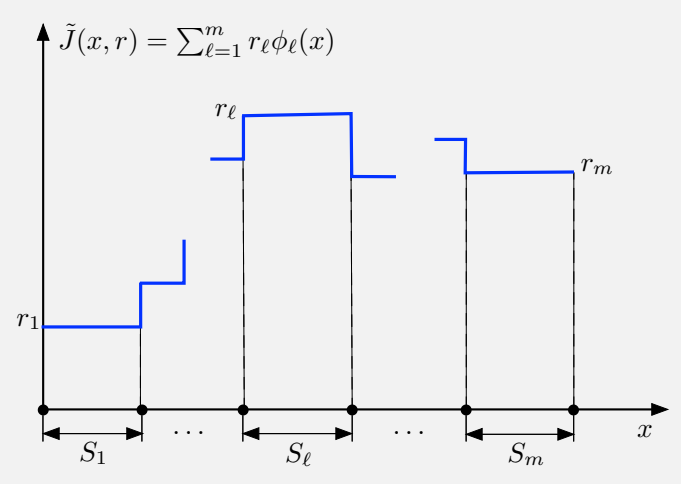


Figure 3.1.4 Illustration of a piecewise constant architecture. The state space is partitioned into subsets $S_1, \ldots, S_m$, with each subset S_ℓ defining the feature

$$\phi_\ell(x) = \begin{cases} 1 & \text{if } x \in S_\ell, \\ 0 & \text{if } x \notin S_\ell, \end{cases} \qquad \ell = 1, \ldots, m,$$

with its own weight r_ℓ.

Example 3.1.1 (Piecewise Constant Approximation)

Suppose that the state space is partitioned into subsets $S_1, \ldots, S_m$, so that every state belongs to one and only one subset. Let the ℓth feature be defined by membership to the set S_ℓ, i.e.,

$$\phi_\ell(x) = \begin{cases} 1 & \text{if } x \in S_\ell, \\ 0 & \text{if } x \notin S_\ell, \end{cases} \qquad \ell = 1, \ldots, m.$$

Consider the architecture

$$\tilde{J}(x,r) = \sum_{\ell=1}^{m} r_\ell \phi_\ell(x),$$

where r is the vector consists of the m scalar parameters $r_1, \ldots, r_m$. It can be seen that $\tilde{J}(x,r)$ is the piecewise constant function that has value r_ℓ for all states within the set S_ℓ; see Fig. 3.1.4.

The piecewise constant approximation is an example of a linear feature-based architecture that involves exclusively *local features*. These are features that take a nonzero value only for a relatively small subset of states. Thus a change of a single weight causes a change of the value of $\tilde{J}(x,r)$ for relatively few states x. At the opposite end we have linear

feature-based architectures that involve *global features*. These are features that take nonzero values for a large number of states. The following is a common example.

Example 3.1.2 (Polynomial Approximation)

An important case of linear architecture is one that uses polynomial basis functions. Suppose that the state consists of n components $x^1, \ldots, x^n$, each taking values within some range of integers. For example, in a queueing system, x^i may represent the number of customers in the ith queue, where $i = 1, \ldots, n$. Suppose that we want to use an approximating function that is quadratic in the components x^i. Then we can define a total of $1 + n + n^2$ basis functions that depend on the state $x = (x^1, \ldots, x^n)$ via

$$\phi_0(x) = 1, \qquad \phi_i(x) = x^i, \qquad \phi_{ij}(x) = x^i x^j, \quad i, j = 1, \ldots, n.$$

A linear approximation architecture that uses these functions is given by

$$\tilde{J}(x, r) = r_0 + \sum_{i=1}^{n} r_i x^i + \sum_{i=1}^{n} \sum_{j=1}^{n} r_{ij} x^i x^j,$$

where the parameter vector r has components r_0, r_i, and r_{ij}, with $i, j = 1, \ldots, n$. Indeed, any kind of approximating function that is polynomial in the components $x^1, \ldots, x^n$ can be constructed similarly.

A more general polynomial approximation may be based on some other known features of the state. For example, we may start with a feature vector

$$\phi(x) = \big(\phi_1(x), \ldots, \phi_m(x)\big)',$$

and transform it with a quadratic polynomial mapping. In this way we obtain approximating functions of the form

$$\tilde{J}(x, r) = r_0 + \sum_{i=1}^{m} r_i \phi_i(x) + \sum_{i=1}^{m} \sum_{j=1}^{m} r_{ij} \phi_i(x) \phi_j(x),$$

where the parameter r has components r_0, r_i, and r_{ij}, with $i, j = 1, \ldots, m$. This can also be viewed as a linear architecture that uses the basis functions

$$w_0(x) = 1, \qquad w_i(x) = \phi_i(x), \qquad w_{ij}(x) = \phi_i(x)\phi_j(x), \quad i, j = 1, \ldots, m.$$

The preceding example architectures are generic in the sense that they can be applied to many different types of problems. Other architectures rely on problem-specific insight to construct features, which are then combined into a relatively simple architecture. We present two examples involving games.

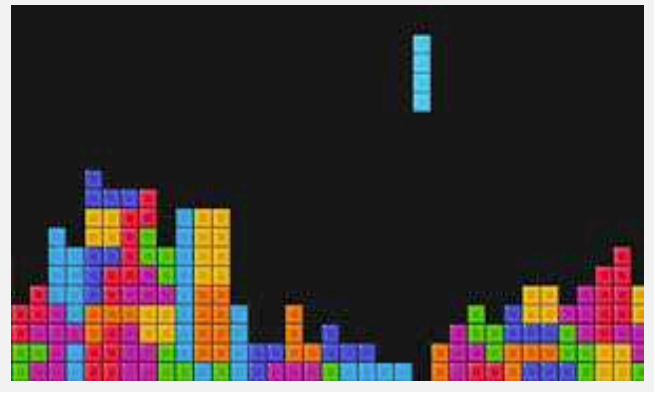


Figure 3.1.5 The board of the tetris game. The squares fill up as blocks of different shapes fall from the top of the grid and are added to the top of the wall. The shapes are generated according to some stochastic process. As a given block falls, the player can move horizontally and rotate the block in all possible ways, subject to the constraints imposed by the sides of the grid and the top of the wall. When a row of full squares is created, this row is removed, the bricks lying above this row move one row downward, and the player scores a point. The player's objective is to maximize the score attained (total number of rows removed) within N steps or up to termination of the game, whichever occurs first.

Example 3.1.3 (Tetris)

Let us consider the game of tetris, which we formulated in Example 1.6.2 as a stochastic shortest path problem with the termination state being the end of the game (see Fig. 3.1.5). The state is the pair of the board position x and the shape of the current falling block y. We viewed as control, the horizontal positioning and rotation applied to the falling block. The optimal cost-to-go function is a vector of huge dimension (there are 2^{200} board positions in a "standard" tetris board of width 10 and height 20). However, it has been successfully approximated in practice by low-dimensional linear architectures.

In particular, the following features have been proposed in the paper by Bertsekas and Ioffe [BeI96]: the heights of the columns, the height differentials of adjacent columns, the wall height (the maximum column height), the number of holes of the board, and the constant 1 (the unit is often included as a feature in cost approximation architectures, as it allows for a constant shift in the approximating function). These features are readily recognized by tetris players as capturing important aspects of the board position.† There

† The use of feature-based approximate DP methods for the game of tetris was first suggested in the paper by Tsitsiklis and Van Roy [TsV96], which introduced just two features (in addition to the constant 1): the wall height and the number of holes of the board. Most studies have used the set of features of [BeI96] described here, but other sets of features have also been used; see [ThS09] and the discussion in [GGS13].

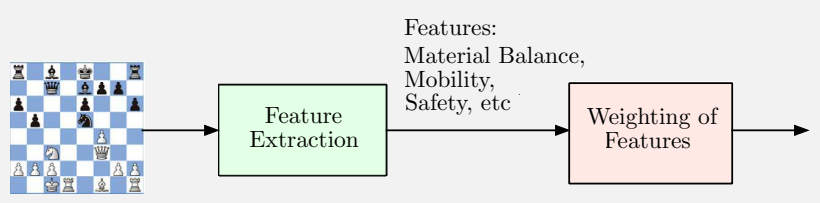


Figure 3.1.6 A feature-based architecture for computer chess.

are a total of 22 features for a "standard" board with 10 columns. Of course the $2^{200} \times 22$ matrix of feature values cannot be stored in a computer, but for any board position, the corresponding row of features can be easily generated, and this is sufficient for implementation of the associated approximate DP algorithms. For recent works involving approximate DP methods and the preceding 22 features, see [Sch13], [GGS13], and [SGG15], which reference several other related papers.

In the works mentioned above the shapes of the falling blocks are stochastically independent. In a more challenging version of the problem, which has not been considered in the literature thus far, successive shapes are correlated. Then the state of the problem would become more complex, since past shapes would be useful in predicting future shapes. As a result, we may need to introduce state estimation and additional features in order to properly deal with the effects of correlations.

Example 3.1.4 (Computer Chess)

Computer chess programs that involve feature-based architectures have been available for many years, and are still used widely (they have been upstaged in the mid-2010s by alternative types of chess programs, which use neural network techniques that will be discussed later). These programs are based on approximate DP for minimax problems, a feature-based parametric architecture, and multistep lookahead.

The fundamental principles on which all computer chess programs (as well as most two-person game programs) are based were laid out by Shannon [Sha50], before Bellman started his work on DP. Shannon proposed multistep lookahead and evaluation of the end positions by means of a "scoring function" (in our terminology this plays the role of a cost function approximation). This function may involve, for example, the calculation of a numerical value for each of a set of major features of a position that chess players easily recognize (such as material balance, mobility, pawn structure, and other positional factors), together with a method to combine these numerical values into a single score. Shannon then went on to describe various strategies of exhaustive and selective search over a multistep lookahead tree of moves.

We may view the scoring function as a feature-based architecture for evaluating a chess position/state (cf. Fig. 3.1.6). In most computer chess programs, the features are weighted linearly, i.e., the architecture $\tilde{J}(x, r)$ that is used for multistep lookahead is linear [cf. Eq. (3.3)]. In many cases, the weights have been determined manually, by trial and error based on experi-

ence. However, in some programs, the weights have been determined with supervised learning techniques that use examples of grandmaster play, i.e., by adjustment to bring the play of the program as close as possible to the play of chess grandmasters. This is a technique that applies more broadly in artificial intelligence; see Tesauro [Tes89b], [Tes01].

In a recent computer chess breakthrough, the entire idea of extracting features of a position through human expertise was abandoned in favor of feature discovery through self-play and the use of neural networks. The first program of this type to attain supremacy over humans, as well as over the best computer programs that use human expertise-based features, was AlphaZero (Silver et al. [SHS17]). This program, described in Section 1.1, is based on DP principles of approximate policy iteration and multistep lookahead based on Monte Carlo tree search.

Our next example relates to a methodology for feature construction, where the number of features may increase as more data is collected. For a simple example, consider the piecewise constant approximation of Example 3.1.1, where more pieces are progressively added based on new data, possibly using some form of exploration-exploitation tradeoff.

Example 3.1.5 (Feature Extraction from Data)

We have viewed so far feature vectors $\phi(x)$ as functions of x, obtained through some unspecified process that is based on prior knowledge about the cost function being approximated. On the other hand, features may also be extracted from data. For example suppose that with some preliminary calculation using data, we have identified some suitable states $x(\ell)$, $\ell = 1, \ldots, m$, that can serve as "anchors" for the construction of Gaussian basis functions of the form

$$\phi_\ell(x) = e^{-\frac{\|x - x(\ell)\|^2}{2\sigma^2}}, \qquad \ell = 1, \ldots, m, \tag{3.4}$$

where σ is a scalar "variance" parameter, and $\|\cdot\|$ denotes the standard Euclidean norm. This type of function is known as a *radial basis function*. It is concentrated around the state $x(\ell)$, and it is weighed with a scalar weight r_ℓ to form a parametric linear feature-based architecture, which can be trained using additional data. Several other types of data-dependent basis functions, such as support vector machines, are used in machine learning, where they are often referred to as *kernels*.

While it is possible to use a preliminary calculation to obtain the anchors $x(\ell)$ in Eq. (3.4), and then use additional data for training, one may also consider enrichment of the set of basis functions simultaneously with training. In this case the number of the basis functions increases as the training data is collected. A motivation here is that the quality of the approximation may increase with additional basis functions. This idea underlies a field of machine learning, known as *kernel methods* or sometimes *nonparametric methods*.

A further discussion is outside our scope. We refer to the literature; see e.g., books such as Cristianini and Shawe-Taylor [ChS00], [ShC04], Scholkopf and Smola [ScS02], Bishop [Bis06], Kung [Kun14], surveys such as Hofmann,

Scholkopf, and Smola [HSS08], Pillonetto et al. [PDC14], RL-related discussions such as Dietterich and Wang [DiW02], Ormoneit and Sen [OrS02], Engel, Mannor, and Meir [EMM05], Jung and Polani [JuP07], Reisinger, Stone, and Miikkulainen [RSM08], Busoniu et al. [BBD10a], Bethke [Bet10], and recent developments such as Tu et al. [TRV16], Rudi, Carratino, and Rosasco [RCR17], Belkin, Ma, and Mandal [BMM18]. In what follows, for the sake of simplicity, we will focus on parametric architectures with a fixed and given feature vector, since the choice of approximation architecture is somewhat peripheral to our main focus.

The next example considers a feature extraction strategy that is particularly relevant to problems of partial state information.

Example 3.1.6 (Feature Extraction from Sufficient Statistics)

The concept of a sufficient statistic, which originated in inference methodologies, plays an important role in DP. As discussed in Section 1.6, it refers to quantities that summarize all the essential content of the state x_k for optimal control selection at time k.

In particular, consider a partial information context where at time k we have accumulated the *information vector* (also called the *past history*)

$$I_k = (z_0, \ldots, z_k, u_0, \ldots, u_{k-1}),$$

which consists of the past controls $u_0, \ldots, u_{k-1}$ and the state-related measurements $z_0, \ldots, z_k$ obtained at the times $0, \ldots, k$. The control u_k is allowed to depend only on I_k, and the optimal policy is a sequence of the form $\big\{\mu_0^*(I_0), \ldots, \mu_{N-1}^*(I_{N-1})\big\}$. We say that a function $S_k(I_k)$ is a *sufficient statistic at time* k if the control function μ_k^* depends on I_k only through $S_k(I_k)$, i.e., for some function $\hat{\mu}_k$, we have

$$\mu_k^*(I_k) = \hat{\mu}_k\big(S_k(I_k)\big),$$

where μ_k^* is optimal.

There are several examples of sufficient statistics, and they are typically problem-dependent. A trivial possibility is to view I_k itself as a sufficient statistic, and a more sophisticated possibility is to view the *belief state* b_k as a sufficient statistic (this is the conditional probability distribution of x_k given I_k; cf. Section 1.6.4). For a proof that b_k is indeed a sufficient statistic and for a more detailed discussion of other possible sufficient statistics, see [Ber17a], Chapter 4. For a mathematically more advanced discussion, see [BeS78], Chapter 10.

Since a sufficient statistic contains all the relevant information for optimal control purposes, an idea that suggests itself is to introduce features of a given sufficient statistic and to train a corresponding approximation architecture accordingly. As examples of potentially good features, one may consider some special characteristic of I_k (such as whether some alarm-like "special" event has been observed), or a partial history (such as the last m measurements and controls in I_k, or more sophisticated versions based on the concept

of a *finite-state controller* proposed by White [Whi91], and White and Scherer [WhS94], and further discussed by Hansen [Han98], Kaelbling, Littman, and Cassandra [KLC98], Meuleau et al. [MPK99], Poupart and Boutilier [PoB04], Yu and Bertsekas [YuB08], Saldi, Yuksel, and Linder [SYL17]). In the case where the belief state b_k is used as a sufficient statistic, examples of good features may be a point estimate based on b_k, the variance of this estimate, and other quantities that can be simply extracted from b_k.

The paper by Bhattacharya et al. [BBW20] considers another type of feature vector that is related to the belief state. This is a sufficient statistic, denoted by y_k, which subsumes the belief state b_k, in the sense that b_k can be computed exactly knowing y_k. One possibility is for y_k to be the union of b_k and some identifiable characteristics of the belief state, or some compact representation of the measurement history up to the current time (such as a number of most recent measurements, or the state of a finite-state controller). Even though the information content of y_k is no different than the information content of b_k for the purposes of exact optimization, a sufficient statistic y_k that is specially designed for the problem at hand may lead to improved performance in the presence of cost and policy approximations.

We finally note a related idea, which is to supplement a sufficient statistic with features of other sufficient statistics, and thus obtain an enlarged/richer sufficient statistic. In problem-specific contexts, and in the presence of approximations, this may yield improved results.

Example 3.1.7 (Feature-Based Dimensionality Reduction by Aggregation)

The use of a feature vector $\phi(x)$ to represent the state x in an approximation architecture of the form $\tilde{J}\big(\phi(x), r\big)$ implicitly involves *state aggregation*, i.e., the grouping of states into subsets. We will discuss aggregation in some detail in Section 3.5. Here we will give a summary of a particular type of aggregation architecture.

In particular, let us assume that the feature vector can take only a finite number of values, and define for each possible value v, the subset of states S_v whose feature vector is equal to v:

$$S_v = \big\{ i \mid \phi(x) = v \big\}.$$

We refer to the sets S_v as the *aggregate states* induced by the feature vector. These sets form a partition of the state space. An approximate cost-to-go function of the form $\tilde{J}\big(\phi(x), r\big)$ is piecewise constant with respect to this partition; that is, it assigns the same cost-to-go value $\tilde{J}(v, r)$ to all states in the set S_v.

An often useful approach to deal with problem complexity in DP is to introduce an "aggregate" DP problem, whose states are some suitably defined feature vectors $\phi(x)$ of the original problem. The precise form of the aggregate problem may depend on intuition and/or heuristic reasoning, based on our understanding of the original problem. Suppose now that the aggregate problem is simple enough to be solved exactly by DP, and let $\hat{J}(v)$

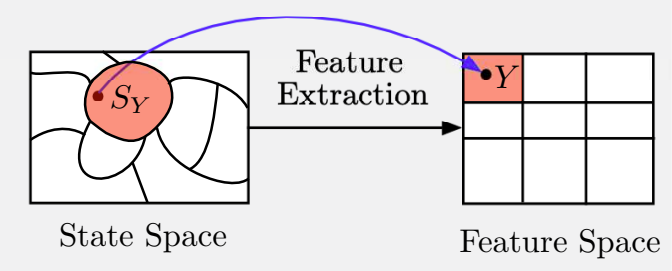


Figure 3.1.7 Feature-based state partitioning using a partition of the space of features. Each set Y of the feature space partition induces a set S_Y of the state space partition that consists of states with "similar" features, i.e., states that map into the same subset of the feature-space partition.

be its optimal cost-to-go when the initial value of the feature vector is v. Then $\hat{J}\big(\phi(x)\big)$ provides an approximation architecture for the original problem, i.e., the architecture that assigns to state x the (exactly) optimal cost-to-go $\hat{J}\big(\phi(x)\big)$ of the feature vector $\phi(x)$ in the aggregate problem. There is considerable freedom on how one formulates and solves aggregate problems. We refer to the DP textbooks [Ber12], [Ber17a], and the RL textbook [Ber19a], Chapter 6, for a detailed treatment; see also the discussion of Section 3.5.

The next example relates to an architecture that is particularly useful when parallel computation is available.

Example 3.1.8 (Feature-Based State Space Partitioning)

A simple method to construct complex and sophisticated approximation architectures, is to partition the state space into several subsets and construct a separate approximation in each subset. For example, by using a separate linear or quadratic polynomial approximation in each subset of the partition, we can construct piecewise linear or piecewise quadratic approximations over the entire state space. Similarly, we may use a separate neural network architecture on each set of the partition. An important issue here is the choice of the method for partitioning the state space. Regular partitions (e.g., grid partitions) may be used, but they often lead to a large number of subsets and very time-consuming computations.

Generally speaking, each subset of the partition should contain "similar" states so that the variation of the optimal cost-to-go over the states of the subset is relatively smooth and can be approximated with smooth functions. An interesting possibility is to use features as the basis for partition. In particular, one may use a more or less regular partition of the space of features, which induces a possibly irregular partition of the original state space. In this way, each subset of the irregular partition contains states with "similar features;" see Fig. 3.1.7.

As an illustration consider the game of chess. The state here consists of the board position, but the nature of the position progresses over time through opening, middlegame, and endgame phases. Moreover each of these

phases may be affected differently by special features of the position. For example there are several different types of endgames (rook endgames, king-and-pawn endgames, minor-piece endgames, etc), which are characterized by identifiable features and call for different playing strategies. It would thus make sense to partition the set of chess positions according to their features, and use a separate strategy on each set of the partition. Indeed this is done to some extent in a number of chess programs.

A potential difficulty with partitioned architectures is that there is discontinuity of the approximation along the boundaries of the partition. For this reason, a variant, called *soft partitioning*, is sometimes employed, whereby the subsets of the partition are allowed to overlap and the discontinuity is smoothed out over their intersection. In particular, once a function approximation is obtained in each subset, the approximate cost-to-go in the overlapping regions is taken to be a smoothly varying linear combination of the function approximations of the corresponding subsets.

Partitioning and local approximations can also be used to enhance the quality of approximation in parts of the space where the target function has some special character. For example, suppose that the state space S is partitioned in subsets $S_1, \ldots, S_M$ and consider approximations of the form

$$\tilde{J}(x,r) = \hat{J}(x,\hat{r}) + \sum_{m=1}^{M} \sum_{k=1}^{K_m} r_m(k)\phi_{k,m}(x), \tag{3.5}$$

where each $\phi_{k,m}(x)$ is a basis function which is local, in the sense that it contributes to the approximation only on the set S_m; that is, it takes the value 0 for $x \notin S_m$. Here $\hat{J}(x,\hat{r})$ is an architecture of the type discussed earlier, and the parameter vector r consists of $\hat{r}$ and the coefficients $r_m(k)$ of the basis functions. Thus the portion $\hat{J}(x,\hat{r})$ of the architecture is used to capture "global" aspects of the target function, while each portion

$$\sum_{k=1}^{K_m} r_m(k)\phi_{k,m}(i)$$

is used to capture aspects of the target function that are "local" to the subset S_m. The book [BeT96] (Section 3.1.3) discusses the training of local-global approximation architectures with methods that are tailored to their special structure.

Architectures with Automatic Feature Construction

Unfortunately, in practice we often do not know an adequate set of features, so it is important to have methods that construct features automatically, to supplement whatever features may already be available. Indeed, there are architectures that do not rely on the knowledge of good features. We have noted the kernel methods of Example 3.1.5 in this connection. Another very popular possibility is *neural networks*, which we will describe in Section 3.2.

Some of these architectures involve training that constructs simultaneously both the feature vectors $\phi(x)$ and the parameter vectors r that weigh them.

Generally, architectures that construct features automatically do not preclude the use of additional features that are based on a priori knowledge or understanding of the problem at hand. In particular these architectures may, in addition to x, use as inputs additional hand-crafted features that are relevant for the problem at hand. Another possibility is to combine automatically constructed features with other a priori known good features into a (mixed) linear architecture that involves both types of features. The weights of the latter linear architecture may be obtained with a separate second stage training process, following the first stage training process that constructs automatically suitable features using a nonlinear architecture such as a neural network.

3.1.3 Training of Linear and Nonlinear Architectures

In this section, we discuss briefly the training process of choosing the parameter vector r of a parametric architecture $\tilde{J}(x, r)$, focusing primarily on incremental gradient methods. The most common type of training is based on a least squares optimization, also known as *least squares regression*. Here a set of state-cost training pairs (x^s, β^s), $s = 1, \ldots, q$, called the *training set*, is collected and r is determined by solving the problem

$$\min_r \sum_{s=1}^{q} \big(\tilde{J}(x^s, r) - \beta^s\big)^2. \tag{3.6}$$

Thus r is chosen to minimize the sum of squared errors between the sample costs β^s and the architecture-predicted costs $\tilde{J}(x^s, r)$. Here there is some target cost function J that we aim to approximate with $\tilde{J}(\cdot, r)$, and the sample cost β^s is the value $J(x^s)$ plus perhaps some error or "noise."

The cost function of the training problem (3.6) is generally nonconvex, and can be quite complicated. This may pose challenges, since there may exist multiple local minima. However, for a linear architecture the cost function is convex quadratic, and the training problem admits a closed-form solution. In particular, for the linear architecture $\tilde{J}(x, r) = r'\phi(x)$, the problem becomes

$$\min_r \sum_{s=1}^{q} \big(r'\phi(x^s) - \beta^s\big)^2.$$

By setting the gradient of the quadratic objective to 0, we obtain

$$\sum_{s=1}^{q} \phi(x^s)\big(r'\phi(x^s) - \beta^s\big) = 0,$$

or

$$\sum_{s=1}^{q} \phi(x^s)\phi(x^s)'r = \sum_{s=1}^{q} \phi(x^s)\beta^s.$$

Thus by matrix inversion we obtain the minimizing parameter vector

$$\hat{r} = \left(\sum_{s=1}^{q} \phi(x^s)\phi(x^s)'\right)^{-1} \sum_{s=1}^{q} \phi(x^s)\beta^s. \tag{3.7}$$

If the inverse above does not exist, an additional quadratic in r, called a *regularization* function, is added to the least squares objective to deal with this, and also to help with other issues to be discussed later. A singular value decomposition approach may also be used to deal with the matrix inversion issue; see [BeT96], Section 3.2.2.

Thus a linear architecture has the important advantage that the training problem can be solved exactly and conveniently with the formula (3.7) (of course it may be solved by any other algorithm that is suitable for linear least squares problems, including iterative algorithms). By contrast, if we use a nonlinear architecture, such as a neural network, the associated least squares problem is nonquadratic and also nonconvex, so it is hard to solve in principle. Despite this fact, through a combination of sophisticated implementation of special gradient algorithms, called *incremental*, and powerful computational resources, neural network methods have been successful in practice.

Incremental Gradient Methods

We will now discuss briefly special methods for solution of the nonlinear least squares training problem (3.6), assuming a parametric architecture that is differentiable in the parameter vector. This methodology can be properly viewed as a subject in nonlinear programming and iterative algorithms, and as such it can be studied independently of the approximate DP methods of this book. Thus the reader who has already some exposure to the subject may skip to the next section. The author's nonlinear programming textbook [Ber16] and the RL book [Ber19a] provide more detailed presentations.

We view the training problem (3.6) as a special case of the minimization of a sum of component functions

$$f(y) = \sum_{i=1}^{m} f_i(y), \tag{3.8}$$

where each f_i is a differentiable scalar function of the n-dimensional column vector y (this is the parameter vector). Thus we use the more common

symbols y and m in place of r and q, respectively, and we replace the squared error terms

$$\big(\tilde{J}(x^s, r) - \beta^s\big)^2$$

in the training problem (3.6) with the generic terms $f_i(y)$.

The (ordinary) gradient method for problem (3.8) generates a sequence $\{y^k\}$ of iterates, starting from some initial guess y^0 for the minimum of the cost function f. It has the form†

$$y^{k+1} = y^k - \gamma^k \nabla f(y^k) = y^k - \gamma^k \sum_{i=1}^{m} \nabla f_i(y^k), \tag{3.9}$$

where γ^k is a positive stepsize parameter. The incremental gradient method is similar to the ordinary gradient method, but uses the gradient of a single component of f at each iteration. It has the general form

$$y^{k+1} = y^k - \gamma^k \nabla f_{i_k}(y^k), \tag{3.10}$$

where i_k is some index from the set $\{1, \ldots, m\}$, chosen by some deterministic or randomized rule. Thus a single component function f_{i_k} is used at iteration k, with great economies in gradient calculation cost over the ordinary gradient method (3.9), particularly when m is large. This is of course a radical simplification, which involves a large approximation error, yet it performs surprisingly well! The idea is to attain faster convergence when far from the solution as we will explain shortly; see the author's books [BeT96], [Ber16], and [Ber19a] for a more detailed discussion.

The method for selecting the index i_k of the component to be iterated on at iteration k is important for the performance of the method. We describe three common rules , the last two of which involve randomization:‡

(1) A *cyclic order*, the simplest rule, whereby the indexes are taken up in the fixed deterministic order $1, \ldots, m$, so that i_k is equal to (k modulo m) plus 1. A contiguous block of iterations involving the components $f_1, \ldots, f_m$ in this order and exactly once is called a *cycle*.

(2) A *uniform random order*, whereby the index i_k chosen randomly by sampling over all indexes with a uniform distribution, independently of the past history of the algorithm. This rule may perform better than the cyclic rule in some circumstances.

---

† We use standard calculus notation for gradients; see, e.g., [Ber16], Appendix A. In particular, $\nabla f(y)$ denotes the n-dimensional column vector whose components are the first partial derivatives $\partial f(y)/\partial y_i$ of f with respect to the components $y_1, \ldots, y_n$ of the column vector y.

‡ With these stepsize rules, the incremental gradient method is often called *stochastic gradient* or *stochastic gradient descent* method.

(3) A *cyclic order with random reshuffling*, whereby the indexes are taken up one by one within each cycle, but their order after each cycle is reshuffled randomly (and independently of the past). This rule is used widely in practice, particularly when the number of components m is modest, for reasons to be discussed later.

Note that in the cyclic cases, it is essential to include all components in a cycle; otherwise some components will be sampled more often than others, leading to a bias in the convergence process. Similarly, it is necessary to sample according to the uniform distribution in the random order case.

Focusing for the moment on the cyclic rule (with or without reshuffling), we note that the motivation for the incremental gradient method is faster convergence: we hope that far from the solution, a single cycle of the method will be as effective as several (as many as m) iterations of the ordinary gradient method (think of the case where the components f_i are similar in structure). Near a solution, however, the incremental method may not be as effective.

To be more specific, we note that there are two complementary performance issues to consider in comparing incremental and nonincremental methods:

(a) *Progress when far from convergence.* Here the incremental method can be much faster. For an extreme case take m large and all components f_i identical to each other. Then an incremental iteration requires m times less computation than a classical gradient iteration, but gives exactly the same result, when the stepsize is scaled to be m times larger. While this is an extreme example, it reflects the essential mechanism by which incremental methods can be much superior: far from the minimum a single component gradient will point to "more or less" the right direction, at least most of the time; see the following example.

(b) *Progress when close to convergence.* Here the incremental method can be inferior. In particular, the ordinary gradient method (3.9) can be shown to converge with a constant stepsize under reasonable assumptions, see e.g., [Ber16], Chapter 1. However, the incremental method requires a diminishing stepsize, and its ultimate rate of convergence can be much slower.

This type of behavior is illustrated in the following example.

Example 3.1.9

Assume that y is a scalar, and that the problem is

$$\begin{aligned} &\text{minimize} \quad f(y) = \tfrac{1}{2}\sum_{i=1}^{m}(c_i y - b_i)^2 \\ &\text{subject to} \quad y \in \Re, \end{aligned}$$

where c_i and b_i are given scalars with $c_i \neq 0$ for all i. The minimum of each of the components $f_i(y) = \frac{1}{2}(c_i y - b_i)^2$ is

$$y_i^* = \frac{b_i}{c_i},$$

while the minimum of the least squares cost function f is

$$y^* = \frac{\sum_{i=1}^m c_i b_i}{\sum_{i=1}^m c_i^2}.$$

It can be seen that y^* lies within the range of the component minima

$$R = \left[\min_i y_i^*,\ \max_i y_i^*\right],$$

and that for all y *outside* the range R, the gradient

$$\nabla f_i(y) = c_i(c_i y - b_i)$$

has the same sign as $\nabla f(y)$ (see Fig. 3.1.8). As a result, when outside the region R, the incremental gradient method

$$y^{k+1} = y^k - \gamma^k c_{i_k}(c_{i_k} y^k - b_{i_k})$$

approaches y^* at each step, provided the stepsize γ^k is small enough. In fact it can be verified that it is sufficient that

$$\gamma^k \leq \min_i \frac{1}{c_i^2}.$$

However, for y *inside* the region R, the ith step of a cycle of the incremental gradient method need not make progress. It will approach y^* (for small enough stepsize γ^k) only if the current point y^k does not lie in the interval connecting y_i^* and y^*. This induces an oscillatory behavior within the region R, and as a result, the incremental gradient method will typically not converge to y^* unless $\gamma^k \to 0$. By contrast, the ordinary gradient method, which takes the form

$$y^{k+1} = y^k - \gamma \sum_{i=1}^m c_i(c_i y^k - b_i),$$

can be verified to converge to y^* for any constant stepsize γ with

$$0 < \gamma \leq \frac{1}{\sum_{i=1}^m c_i^2}.$$

However, for y outside the region R, a full iteration of the ordinary gradient method need not make more progress towards the solution than a single step of the incremental gradient method. In other words, with comparably intelligent stepsize choices, *far from the solution (outside R), a single pass through the entire set of cost components by incremental gradient is roughly as effective as m passes by ordinary gradient.*

The preceding example assumes that each component function f_i has a minimum, so that the range of component minima is defined. In cases where the components f_i have no minima, a similar phenomenon may occur, as illustrated by the following example (the idea here is that we may combine several components into a single component that has a minimum).

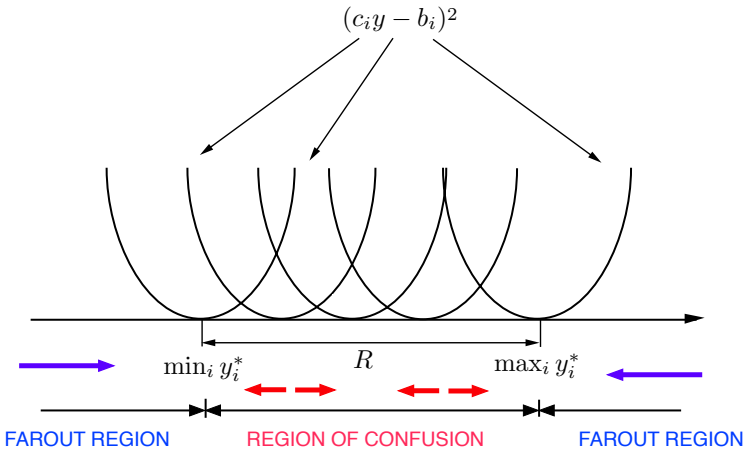


Figure 3.1.8. Illustrating the advantage of incrementalism when far from the optimal solution. The region of component minima

$$R = \left[\min_i y_i^*,\ \max_i y_i^*\right],$$

is labeled as the "region of confusion." It is the region where the method does not have a clear direction towards the optimum. The ith step in an incremental gradient cycle is a gradient step for minimizing $(c_i y - b_i)^2$, so if y lies outside the region of component minima $R = \left[\min_i y_i^*,\ \max_i y_i^*\right]$, (labeled as the "farout region") and the stepsize is small enough, progress towards the solution y^* is made.

Example 3.1.10:

Consider the case where f is the sum of increasing and decreasing convex exponentials, i.e.,

$$f_i(y) = a_i e^{b_i y}, \qquad y \in \Re,$$

where a_i and b_i are scalars with $a_i > 0$ and $b_i \neq 0$. Let

$$I^+ = \{i \mid b_i > 0\}, \qquad I^- = \{i \mid b_i < 0\},$$

and assume that I^+ and I^- have roughly equal numbers of components. Let also y^* be the minimum of $\sum_{i=1}^m f_i$.

Consider the incremental gradient method that given the current point, call it y^k, chooses some component f_{i_k} and iterates according to the incremental gradient iteration

$$y^{k+1} = y^k - \alpha^k \nabla f_{i_k}(y^k).$$

Then it can be seen that if $y^k >> y^*$, y^{k+1} will be substantially closer to y^* if $i \in I^+$, and negligibly further away than y^* if $i \in I^-$. The net effect, averaged

over many incremental iterations, is that if $y^k >> y^*$, an incremental gradient iteration makes roughly one half the progress of a full gradient iteration, with m times less overhead for calculating gradients. The same is true if $y^k << y^*$. On the other hand as y^k gets closer to y^* the advantage of incrementalism is reduced, similar to the preceding example. In fact in order for the incremental method to converge, a diminishing stepsize is necessary, which will ultimately make the convergence slower than the one of the nonincremental gradient method with a constant stepsize.

The discussion of the preceding examples relies on y being one-dimensional, but in many multidimensional problems the same qualitative behavior can be observed. In particular, a pass through the ith component f_i by the incremental gradient method can make progress towards the solution in the region where the component gradient $\nabla f_{i_k}(y^k)$ makes an angle less than 90 degrees with the cost function gradient $\nabla f(y^k)$. If the components f_i are not "too dissimilar," this is likely to happen in a region of points that are not too close to the optimal solution set. This behavior has been verified in many practical contexts, including the training of neural networks (cf. the next section), where incremental gradient methods have been used extensively, frequently under the name *backpropagation methods*.

Stepsize Choice and Diagonal Scaling

The choice of the stepsize γ^k plays an important role in the performance of incremental gradient methods. In practice, it is common to use a constant stepsize for a (possibly prespecified) number of iterations, then decrease the stepsize by a certain factor, and repeat, up to the point where the stepsize reaches a prespecified floor value. An alternative possibility is to use a diminishing stepsize rule of the form

$$\gamma^k = \min\left\{\gamma,\ \frac{\beta_1}{k+\beta_2}\right\},$$

where γ, β_1, and β_2 are some positive scalars. There are also variants of the method that use a constant stepsize throughout, and can be shown to converge to a stationary point of f under reasonable assumptions. In one type of such method the degree of incrementalism gradually diminishes as the method progresses (see [Ber97a]). Another incremental approach with similar aims, is the aggregated gradient method, which is discussed in the author's textbooks [Ber15a], [Ber16], [Ber19a].

Regardless of whether a constant or a diminishing stepsize is ultimately used, the incremental method must use a much larger stepsize than the corresponding nonincremental gradient method (as much as m times larger, so that the size of the incremental gradient step is comparable to the size of the nonincremental gradient step).

One possibility is to use an adaptive stepsize rule, whereby, roughly speaking, the stepsize is reduced (or increased) when the progress of the

method indicates that the algorithm is (or is not) oscillating. There are formal ways to implement such stepsize rules with sound convergence properties (see [Tse98], [MYF03]).

The difficulty with stepsize selection may also be addressed with *diagonal scaling*, i.e., using a stepsize γ_j^k that is different for each of the components y_j of y. Second derivatives can be very useful for this purpose. In generic nonlinear programming problems of unconstrained minimization of a function f, it is common to use diagonal scaling with stepsizes

$$\gamma_j^k = \gamma \left(\frac{\partial^2 f(y^k)}{\partial^2 y_j} \right)^{-1}, \qquad j = 1, \ldots, n,$$

where γ is a constant that is nearly equal 1 (the second derivatives may also be approximated by gradient difference approximations). However, in least squares training problems, this type of scaling is inconvenient because of the additive form of f as a sum of a large number of component functions:

$$f(y) = \sum_{i=1}^{m} f_i(y),$$

cf. Eq. (3.8). The neural network literature includes a number of practical scaling schemes, some of which have been incorporated in publicly and commercially available software.

The RL book [Ber19a] (Section 3.1.3) describes another type method that involves second derivatives and is based on Newton's method. The idea here is to write Newton's method in a format that is well suited to the additive character of the cost function f, and involves low order matrix inversion. One can then implement diagonal scaling by setting to zero the off-diagonal terms of the inverted matrices, so that the algorithm involves no matrix inversion. There is also another related algorithm, which is based on the Gauss-Newton method and the extended Kalman filter; see the author's paper [Ber96], and the books [BeT96] and [Ber16].

3.2 NEURAL NETWORKS

There are several different types of neural networks that can be used for a variety of tasks, such as pattern recognition, classification, image and speech recognition, natural language processing, and others. In this section, we focus on our finite horizon DP context, and the role that neural networks can play in approximating the optimal cost-to-go functions J_k^*. As an example within this context, we may first use a neural network to construct an approximation to J_{N-1}^*. Then we may use this approximation to approximate J_{N-2}^*, and continue this process backwards in time, to obtain approximations to all the optimal cost-to-go functions J_k^*, $k = 1, \ldots, N-1$, as we will discuss in more detail in Section 3.3.

Throughout this section, we will focus on the type of neural network, known as a *multilayer perceptron*, which is the one most used at present in the RL applications discussed in these notes. Naturally, there are variations that are adapted to the problem at hand. For example AlphaZero uses a specialized neural network that can take advantage of the board-like structure of chess and Go to facilitate and expedite the associated computations.

To describe the use of neural networks in finite horizon DP, let us consider the typical stage k, and for convenience drop the index k; the subsequent discussion applies to each value of k separately. We consider parametric architectures $\tilde{J}(x, v, r)$ of the form

$$\tilde{J}(x, v, r) = r' \phi(x, v) \tag{3.11}$$

that depend on two parameter vectors v and r. Our objective is to select v and r so that $\tilde{J}(x, v, r)$ approximates some target cost function that can be sampled (possibly with some error). The process is to collect a training set that consists of a large number of state-cost pairs (x^s, β^s), $s = 1, \ldots, q$, and to find a function $\tilde{J}(x, v, r)$ of the form (3.11) that matches the training set in a least squares sense, i.e., (v, r) minimizes

$$\sum_{s=1}^{q} \big(\tilde{J}(x^s, v, r) - \beta^s\big)^2.$$

We postpone for later the question of how the training pairs (x^s, β^s) are generated.† Notice the different roles of the two parameter vectors here: v *parametrizes* $\phi(x, v)$, which in some interpretation may be viewed as a feature vector, and r *is a vector of linear weighting parameters for the components of* $\phi(x, v)$.

Single Layer Perceptron

A neural network architecture provides a parametric class of functions $\tilde{J}(x, v, r)$ of the form (3.11) that can be used in the optimization framework just described. The simplest type of neural network is the *single layer perceptron*; see Fig. 3.2.1. Here the state x is encoded as a vector of numerical values $y(x)$ with components $y_1(x), \ldots, y_n(x)$, which is then transformed linearly as

$$Ay(x) + b,$$

† The least squares training problem used here is based on *nonlinear regression*. This is a classical method for approximating the expected value of a function with a parametric architecture, and involves a least squares fit of the architecture to simulation-generated samples of the expected value. We refer to machine learning and statistics textbooks for more discussion.

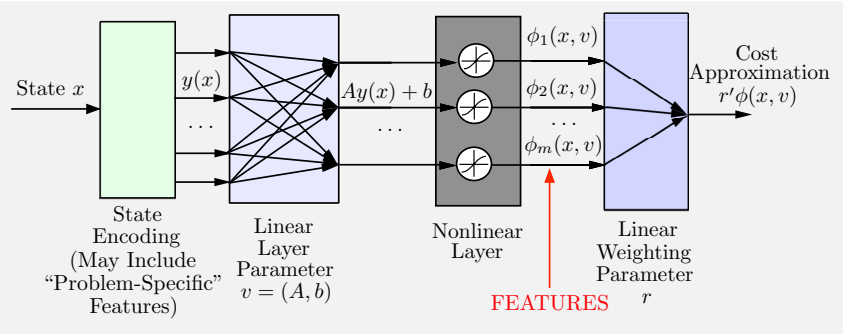


Figure 3.2.1 Schematic illustration of a single layer perceptron, a neural network consisting of a linear layer and a nonlinear layer. It provides a way to compute features of the state, which can be used for cost function approximation. The state x is encoded as a vector of numerical values $y(x)$, which is then transformed linearly as $Ay(x)+b$ in the linear layer. The m scalar output components of the linear layer, become the inputs to nonlinear one-dimensional functions $\sigma : \Re \mapsto \Re$, thus producing the m scalars

$$\phi_\ell(x,v) = \sigma\big((Ay(x)+b)_\ell\big),$$

which can be viewed as features that are in turn linearly weighted with parameters r_ℓ.

where A is an $m \times n$ matrix and b is a vector in $\Re^m$.† This transformation is called the *linear layer* of the neural network. We view the components of A and b as parameters to be determined, and we group them together into the parameter vector $v = (A, b)$.

Each of the m scalar output components of the linear layer,

$$\big(Ay(x)+b\big)_\ell, \qquad \ell = 1, \ldots, m,$$

becomes the input to a nonlinear differentiable and monotonically increasing function σ that maps scalars to scalars. A simple and popular possibility is the *rectified linear unit* (ReLU for short), which is simply the function $\max\{0, \xi\}$, approximated by a differentiable function σ by some form of smoothing operation; for example $\sigma(\xi) = \ln(1+e^\xi)$, which is illustrated in Fig. 3.2.2. Other functions, used since the early days of neural networks, have the property

$$-\infty < \lim_{\xi \to -\infty} \sigma(\xi) < \lim_{\xi \to \infty} \sigma(\xi) < \infty;$$

† The method of encoding x into the numerical vector $y(x)$ is generally problem-dependent, but it can be critical for the success of the training process. We should note also that some of the components of $y(x)$ could be known interesting features of x that can be designed based on problem-specific knowledge.

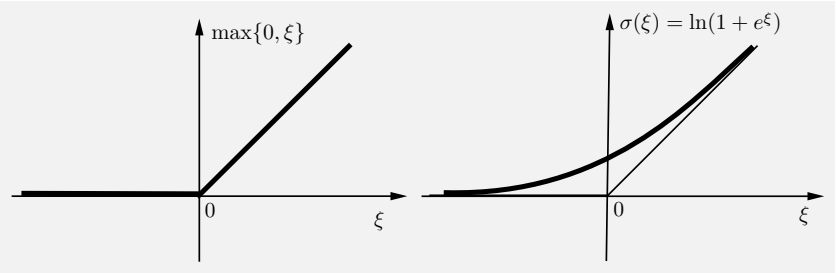


Figure 3.2.2 The rectified linear unit $\sigma(\xi) = \ln(1 + e^{\xi})$. It is the function $\max\{0, \xi\}$ with its corner "smoothed out." Its derivative is $\sigma'(\xi) = e^{\xi}/(1 + e^{\xi})$, and approaches 0 and 1 as $\xi \to -\infty$ and $\xi \to \infty$, respectively.

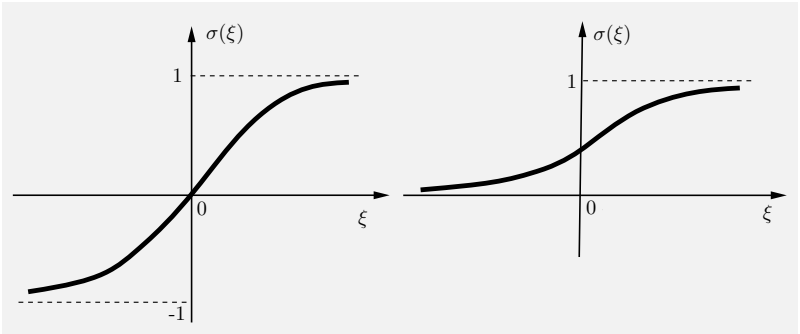


Figure 3.2.3 Some examples of sigmoid functions. The hyperbolic tangent function is on the left, while the logistic function is on the right.

see Fig. 3.2.3. Such functions are called *sigmoids*, and some common choices are the *hyperbolic tangent* function

$$\sigma(\xi) = \tanh(\xi) = \frac{e^{\xi} - e^{-\xi}}{e^{\xi} + e^{-\xi}},$$

and the *logistic* function

$$\sigma(\xi) = \frac{1}{1 + e^{-\xi}}.$$

In what follows, we will ignore the character of the function σ (except for differentiability), and simply refer to it as a "nonlinear unit" and to the corresponding layer as a "nonlinear layer."

At the outputs of the nonlinear units, we obtain the scalars

$$\phi_\ell(x, v) = \sigma\big((Ay(x) + b)_\ell\big), \qquad \ell = 1, \ldots, m.$$

One possible interpretation is to view $\phi_\ell(x, v)$ as features of x, which are linearly combined using weights r_ℓ, $\ell = 1, \ldots, m$, to produce the final

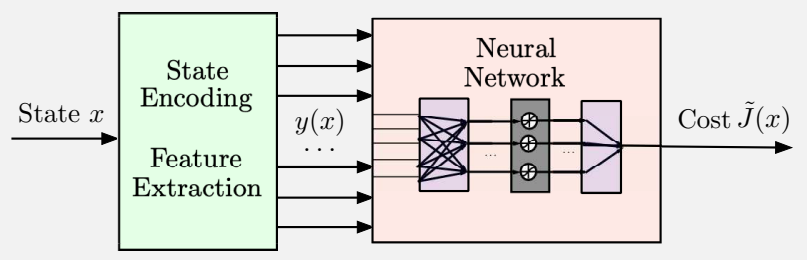


Figure 3.2.4 Nonlinear architecture with a view of the state encoding process as a feature extraction mapping preceding the neural network. The state encoder may also contain tunable parameters.

output

$$\tilde{J}(x, v, r) = \sum_{\ell=1}^{m} r_\ell \phi_\ell(x, v) = \sum_{\ell=1}^{m} r_\ell \sigma\big((Ay(x) + b)_\ell\big).$$

Note that each value $\phi_\ell(x, v)$ depends on just the ℓth row of A and the ℓth component of b, not on the entire vector v. In some cases this motivates placing some constraints on individual components of A and b to achieve special problem-dependent "handcrafted" effects.

State Encoding and Direct Feature Extraction

The state encoding operation that transforms x into the neural network input $y(x)$ can be instrumental in the success of the approximation scheme. Examples of state encodings are components of the state x, numerical representations of qualitative characteristics of x, and more generally features of x, i.e., functions of x that aim to capture "important nonlinearities" of the optimal cost-to-go function. With the latter view of state encoding, we may consider the approximation process as consisting of a feature extraction mapping, followed by a neural network with input the extracted features of x, and output the cost-to-go approximation; see Fig. 3.2.4. In a more general view of the neural network, the state encoder may involve some tunable parameters.

The idea here is that with a good feature extraction mapping, the neural network need not be very complicated (may involve few nonlinear units and corresponding parameters), and may be trained more easily. This intuition is borne out by simple examples and practical experience. However, as is often the case with neural networks, it is hard to support it with a quantitative analysis.

Universal Approximation Property of Neural Networks

An important question is how well we can approximate the target function J_k^* with a neural network architecture, assuming we can choose the number of the nonlinear units m to be as large as we want. The answer to

this question is quite favorable and is provided by the so-called *universal approximation theorem*.

Roughly, the theorem says that assuming that x is an element of a Euclidean space X and $y(x) \equiv x$, a neural network of the form described can approximate arbitrarily closely (in an appropriate mathematical sense), over a compact subset $S \subset X$, any piecewise continuous function $J : S \mapsto \Re$, provided the number m of nonlinear units is sufficiently large. For proofs of the theorem, we refer to Cybenko [Cyb89], Funahashi [Fun89], Hornik, Stinchcombe, and White [HSW89], and Leshno et al. [LLP93]. For additional sources and intuitive explanations we refer to Bishop ([Bis95], pp. 129-130), Jones [Jon90], and the RL textbook [Ber19a], Section 3.2.1.

While the universal approximation theorem provides some assurance about the adequacy of the neural network structure, it does not predict how many nonlinear units we may need for "good" performance in a given problem. Unfortunately, this is a difficult question to even pose precisely, let alone to answer adequately. In practice, one is often reduced to trying increasingly larger values of m until one is convinced that satisfactory performance has been obtained for the task at hand. One may improve on trial-and-error schemes with more systematic hyperparameter search methods, such as Bayesian optimization, and in fact this has been used to tune the parameters of the deep network used by AlphaZero. Experience has shown that in many cases the number of required nonlinear units and corresponding dimension of A can be very large, adding significantly to the difficulty of solving the training problem. This has given rise to many suggestions for modifications of the neural network structure. An important possibility is to concatenate multiple single layer perceptrons so that the output of the nonlinear layer of one perceptron becomes the input to the linear layer of the next, giving rise to deep neural networks, which we will discuss later.

3.2.1 Training of Neural Networks

Given a set of state-cost training pairs (x^s, β^s), $s = 1, \ldots, q$, the parameters of the neural network A, b, and r are obtained by solving the problem

$$\min_{A,b,r} \sum_{s=1}^{q} \left(\sum_{\ell=1}^{m} r_\ell \sigma\left(\big(Ay(x^s) + b\big)_\ell \right) - \beta^s \right)^2. \tag{3.12}$$

Note that the cost function of this problem is generally nonconvex, so there may exist multiple local minima.

In practice it is common to augment the cost function of this problem with a *regularization* function, such as a quadratic in the parameters A, b, and r. This is customary in least squares problems in order to make the problem easier to solve algorithmically. However, in the context of neural network training, regularization is primarily important for a different

reason: it helps to avoid *overfitting*, which occurs when the number of parameters of the neural network is relatively large (comparable to the size of the training set). In this case a neural network model matches the training data very well but may not do as well on new data. This is a known difficulty, which is the subject of much current research, particularly in the context of deep neural networks.

An important issue is to select a method to solve the training problem (3.12). While we can use any unconstrained optimization method that is based on gradients, in practice it is important to take into account the cost function structure of problem (3.12). The salient characteristic of this cost function is that it is the sum of a potentially very large number q of component functions. This makes the computation of the cost function value of the training problem and/or its gradient very costly. For this reason the incremental methods of Section 3.1.3 are universally used for training.† Experience has shown that these methods can be vastly superior to their nonincremental counterparts in the context of neural network training.

The implementation of the training process has benefited from experience that has been accumulated over time, and has provided guidelines for scaling, regularization, initial parameter selection, and other practical issues; we refer to books on neural networks such as Bishop [Bis95], Goodfellow, Bengio, and Courville [GBC16], and Haykin [Hay08], and to the overview paper on deep neural network training [Sun19] for related accounts. Still, incremental methods can be quite slow, and training may be a time-consuming process. Fortunately, the training is ordinarily done off-line, possibly using parallel computation, in which case computation time may not be a serious issue. Moreover, in practice the neural network training problem typically need not be solved with great accuracy. This is also supported by the Newton step view of approximation in value space, which suggests that great accuracy in the terminal cost function approximation is not critically important for good performance of the on-line play controller.

3.2.2 Multilayer and Deep Neural Networks

An important generalization of the single layer perceptron architecture involves a concatenation of multiple layers of linear and nonlinear functions; see Fig. 3.2.5. In particular the outputs of each nonlinear layer become the inputs of the next linear layer. In some cases it may make sense to add

† The incremental methods are valid for an arbitrary order of component selection within the cycle, but it is common to randomize the order at the beginning of each cycle. Also, in a variation of the basic method, we may operate on a batch of several components at each iteration, called a *minibatch*, rather than a single component. This has an averaging effect, which reduces the tendency of the method to oscillate and allows for the use of a larger stepsize; see the end-of-chapter references.

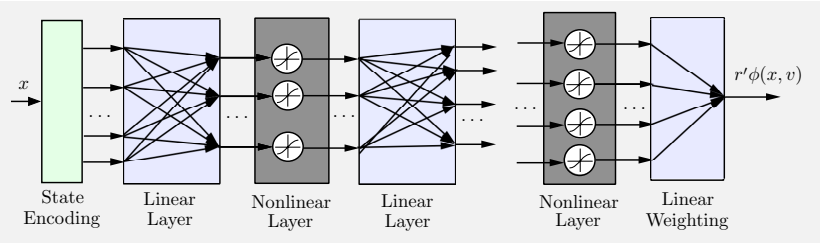


Figure 3.2.5 A deep neural network, with multiple layers. Each nonlinear layer constructs the inputs of the next linear layer.

as additional inputs some of the components of the state x or the state encoding $y(x)$.

In the early days of neural networks practitioners tended to use few nonlinear layers (say one to three). However, more recently a lot of success in certain problem domains (including image and speech processing, as well as approximate DP) has been achieved with *deep neural networks*, which involve a considerably larger number of layers.

There are a few questions to consider here. The first has to do with the reason for having multiple nonlinear layers, when a single one is sufficient to guarantee the universal approximation property. Here are some qualitative (and somewhat speculative) explanations:

(a) If we view the outputs of each nonlinear layer as features, we see that the multilayer network produces a hierarchy of features, where each set of features is a function of the preceding set of features [except for the first set of features, which is a function of the encoding $y(x)$ of the state x]. In the context of specific applications, this hierarchical structure can be exploited to specialize the role of some of the layers and to enhance some characteristics of the state.

(b) Given the presence of multiple linear layers, one may consider the possibility of using matrices A with a particular sparsity pattern, or other structure that embodies special linear operations such as convolution, which may be well-matched to the training problem at hand. Moreover, when such structures are used, the training problem often becomes easier, because the number of parameters in the linear layers is drastically decreased.

(c) Overparametrization (more weights than data, as in a deep neural network) helps to mitigate the detrimental effects of overfitting, and the attendant need for regularization. The explanation for this fascinating phenomenon (observed as early as the late 90s) is the subject of much current research; see [ZBH16], [BMM18], [BRT18], [SJL18], [ADH19], [BLL19], [HMR19], [MVS19], [SuY19], [Sun19], [HaR21],

[VLK21], [ZBH21] for representative works.

We finally note that the use of deep neural networks has been an important factor for the success of the AlphaGo and AlphaZero programs that play Go and chess, respectively; see [SHM16], [SHS17]. By contrast, Tesauro's backgammon program and its descendants have performed well with one or two nonlinear layers [PaR12]. Moreover, as new applications of approximate DP/RL are being considered, it is likely that different and/or specialized neural network architectures will be discovered, which may be better suited to the structure of these applications.

3.3 TRAINING OF COST FUNCTIONS IN APPROXIMATE DP

In the context of approximate DP/RL, architectures are mainly used to approximate either cost functions or policies. When a neural network is involved, the terms *value network* and *policy network* are commonly used, respectively.† In this section we will illustrate the use of value networks in finite horizon DP, while in the next section we will discuss the use of policy networks. We will also illustrate in Section 3.3.3 the combined use of policy and value networks within an approximate policy iteration context, whereby the policies and their cost functions are approximated by a policy and a value network, respectively, to generate a sequence of (approximately) improved policies. Finally, in Sections 3.3.4 and 3.4.5, we will describe how approximating Q-factor or cost differences (rather than Q-factors or costs) can be beneficial within our context of approximation in value space.

3.3.1 Fitted Value Iteration

Let us describe a popular approach for training an approximation architecture $\tilde{J}_k(x_k, r_k)$ for a finite horizon DP problem. The parameter vectors r_k are determined sequentially, starting from the end of the horizon, and proceeding backwards as in the DP algorithm: first r_{N-1} then r_{N-2}, and so on. The algorithm samples the state space for each stage k, and generates a large number of states x_k^s, $s = 1, \ldots, q$. It then determines sequentially the parameter vectors r_k to obtain a good "least squares fit" to the DP algorithm. The method can also be used in the infinite horizon case, in essentially identical form, and it is commonly called *fitted value iteration*.

In particular, each r_k is determined by generating a large number of sample states and solving a least squares problem that aims to minimize the error in satisfying the DP equation for these states at time k. At

† The alternative terms *critic network* and *actor network* are also used often. In these notes, we will use the terms "value network" and "policy network."

the typical stage k, having obtained r_{k+1}, we determine r_k from the least squares problem

$$r_k \in \arg\min_r \sum_{s=1}^{q} \left(\tilde{J}_k(x_k^s, r) - \min_{u \in U_k(x_k^s)} E\Big\{ g_k(x_k^s, u, w_k) + \tilde{J}_{k+1}\big(f_k(x_k^s, u, w_k), r_{k+1}\big) \Big\} \right)^2$$

where x_k^s, $i = 1, \ldots, q$, are the sample states that have been generated for the kth stage. Since r_{k+1} is assumed to be already known, the complicated minimization term in the right side of this equation is the known scalar

$$\beta_k^s = \min_{u \in U_k(x_k^s)} E\Big\{ g_k(x_k^s, u, w_k) + \tilde{J}_{k+1}\big(f_k(x_k^s, u, w_k), r_{k+1}\big) \Big\}, \qquad (3.13)$$

so that r_k is obtained as

$$r_k \in \arg\min_r \sum_{s=1}^{q} \big(\tilde{J}_k(x_k^s, r) - \beta_k^s \big)^2. \qquad (3.14)$$

The algorithm starts at stage $N-1$ with the minimization

$$r_{N-1} \in \arg\min_r \sum_{s=1}^{q} \left(\tilde{J}_{N-1}(x_{N-1}^s, r) - \min_{u \in U_{N-1}(x_{N-1}^s)} E\Big\{ g_{N-1}(x_{N-1}^s, u, w_{N-1}) + g_N\big(f_{N-1}(x_{N-1}^s, u, w_{N-1})\big) \Big\} \right)^2$$

and ends with the calculation of r_0 at $k = 0$.

In the case of a linear architecture, where the approximate cost-to-go functions are

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

the least squares problem (3.14) greatly simplifies, and admits the closed form solution

$$r_k = \left(\sum_{s=1}^{q} \phi_k(x_k^s) \phi_k(x_k^s)' \right)^{-1} \sum_{s=1}^{q} \beta_k^s \phi_k(x_k^s);$$

cf. Eq. (3.7). For a nonlinear architecture such as a neural network, incremental gradient algorithms may be used.

An important implementation issue is how to select the sample states x_k^s, $s = 1, \ldots, q$, $k = 0, \ldots, N-1$. In practice, they are typically obtained

by some form of Monte Carlo simulation, but the distribution by which they are generated is important for the success of the method. In particular, it is important that the sample states are "representative" in the sense that they are visited often under a nearly optimal policy. More precisely, the frequencies with which various states appear in the sample should be roughly proportional to the probabilities of their occurrence under an optimal policy.

Aside from the issue of selection of the sampling distribution that we have just described, a difficulty with fitted value iteration arises when the horizon N is very long, since then the total number of parameters over the N stages may become excessive. In this case, however, the problem is often stationary, in the sense that the system and cost per stage do not change as time progresses. Then it may be possible to treat the problem as one with an infinite horizon and bring to bear additional methods for training approximation architectures; see the relevant discussions in Chapter 5 of the book [Ber19a].

We finally note an important difficulty with the training method of this section: the calculation of each sample β_k^s of Eq. (3.13) requires a minimization of an expected value, which can be very time consuming. In the next section, we describe an alternative type of fitted value iteration, which uses Q-factors, and involves a simpler minimization, whereby the order of the minimization and expectation operations in Eq. (3.13) is reversed.

3.3.2 Q-Factor Parametric Approximation - Model-Free Implementation

We will now consider an alternative form of approximation in value space and fitted value iteration, which involves approximation of the optimal Q-factors of state-control pairs (x_k, u_k) at time k, with no intermediate approximation of cost-to-go functions. An important characteristic of this algorithm is that it *allows for a model-free computation* (i.e., the use of a computer model in place of a mathematical model).

We recall that the optimal Q-factors are defined by

$$Q_k^*(x_k,u_k) = E\Big\{g_k(x_k,u_k,w_k)+J_{k+1}^*\big(f_k(x_k,u_k,w_k)\big)\Big\}, \quad k=0,\ldots,N-1, \tag{3.15}$$

where J_{k+1}^* is the optimal cost-to-go function for stage $k+1$. Thus $Q_k^*(x_k,u_k)$ is the cost attained by using u_k at state x_k, and subsequently using an optimal policy.

As noted in Section 1.3, the DP algorithm can be written as

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

and by using this equation, we can write Eq. (3.15) in the following equivalent form that relates Q_k^* with Q_{k+1}^*:

$$Q_k^*(x_k,u_k) = E\Big\{ g_k(x_k,u_k,w_k) + \min_{u\in U_{k+1}(f_k(x_k,u_k,w_k))} Q_{k+1}^*\big(f_k(x_k,u_k,w_k),u\big)\Big\}. \tag{3.16}$$

This suggests that in place of the Q-factors $Q_k^*(x_k,u_k)$, we may use Q-factor approximations as the basis for suboptimal control.

We can obtain such approximations by using methods that are similar to the ones we have considered so far. Parametric Q-factor approximations $\tilde{Q}_k(x_k,u_k,r_k)$ may involve a neural network, or a feature-based linear architecture. The feature vector may depend on just the state, or on both the state and the control. In the former case, the architecture has the form

$$\tilde{Q}_k(x_k,u_k,r_k) = r_k(u_k)'\phi_k(x_k), \tag{3.17}$$

where $r_k(u_k)$ is a separate weight vector for each control u_k. In the latter case, the architecture has the form

$$\tilde{Q}_k(x_k,u_k,r_k) = r_k'\phi_k(x_k,u_k), \tag{3.18}$$

where r_k is a weight vector that is independent of u_k. The architecture (3.17) is suitable for problems with a relatively small number of control options at each stage. In what follows, we will focus on the architecture (3.18), but the discussion, with few modifications, also applies to the architecture (3.17) and to nonlinear architectures as well.

We may adapt the fitted value iteration approach of the preceding section to compute sequentially the parameter vectors r_k in Q-factor parametric approximations, starting from $k = N-1$. This algorithm is based on Eq. (3.16), with r_k obtained by solving least squares problems similar to the ones of the cost function approximation case [cf. Eq. (3.14)]. As an example, the parameters r_k of the architecture (3.18) are computed sequentially by collecting sample state-control pairs (x_k^s,u_k^s), $s=1,\ldots,q$, and solving the linear least squares problems

$$r_k \in \arg\min_r \sum_{s=1}^{q} \big(r'\phi_k(x_k^s,u_k^s) - \beta_k^s\big)^2, \tag{3.19}$$

where

$$\beta_k^s = E\left\{ g_k(x_k^s,u_k^s,w_k) + \min_{u\in U_{k+1}(f_k(x_k^s,u_k^s,w_k))} r_{k+1}'\phi_{k+1}\big(f_k(x_k^s,u_k^s,w_k),u\big)\right\}. \tag{3.20}$$

Thus, having obtained r_{k+1}, we obtain r_k through a least squares fit that aims to minimize the sum of the squared errors in satisfying Eq. (3.16). Note that the solution of the least squares problem (3.19) can be obtained in closed form as

$$r_k = \left(\sum_{s=1}^{q} \phi_k(x_k^s, u_k^s) \phi_k(x_k^s, u_k^s)' \right)^{-1} \sum_{s=1}^{q} \beta_k^s \phi_k(x_k^s, u_k^s);$$

[cf. Eq. (3.7)]. Once r_k has been computed, the one-step lookahead control $\tilde{\mu}_k(x_k)$ is obtained on-line as

$$\tilde{\mu}_k(x_k) \in \arg \min_{u \in U_k(x_k)} \tilde{Q}_k(x_k, u, r_k), \tag{3.21}$$

without the need to calculate any expected value. This latter property is a primary incentive for using Q-factors in approximate DP, particularly when there are tight constraints on the amount of on-line computation that is possible in the given practical setting.

The samples β_k^s of Eq. (3.20) involve the exact computation of an expected value. In an alternative implementation, we may replace β_k^s with an average of just a few samples (even a single sample) of the random variable

$$g_k(x_k^s, u_k^s, w_k) + \min_{u \in U_{k+1}(f_k(x_k^s, u_k^s, w_k))} r_{k+1}' \phi_{k+1}\big(f_k(x_k^s, u_k^s, w_k), u\big), \tag{3.22}$$

collected according to the probability distribution of w_k. This distribution may either be known explicitly, or in a model-free situation, through a computer simulator. In particular, *to implement this scheme, we only need a simulator that for any pair (x_k, u_k) generates a sample of the stage cost $g_k(x_k, u_k, w_k)$ and the next state $f_k(x_k, u_k, w_k)$ according to the distribution of w_k.*

Note that the samples of the random variable (3.22) do not require the computation of an expected value like the samples (3.13) in the cost approximation method of the preceding chapter. Moreover the samples of (3.22) involve a simpler minimization than the samples (3.13). This is an important advantage of working with Q-factors rather than state costs.

Having obtained the weight vectors $r_0, \ldots, r_{N-1}$, and hence the one-step lookahead policy $\tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$ through Eq. (3.21), a further possibility is to approximate this policy with a parametric architecture. This is *approximation in policy space built on top of approximation in value space*. The idea here is to simplify even further the on-line computation of the suboptimal controls by avoiding the minimization of Eq. (3.21).

3.3.3 Parametric Approximation in Infinite Horizon Problems - Approximate Policy Iteration

In this section we will briefly discuss parametric approximation methods for infinite horizon problems, based on the policy iteration (PI) method. We will focus on the finite-state version of the α-discounted problem of Section 1.4.1, and adopt notation that is more convenient for such problems. In particular, states and successor states will be denoted by i and j, respectively. Moreover the system equation will be represented by control-dependent transition probabilities $p_{ij}(u)$ (the probability that the system will move to state j, given that it starts at state i and control u is applied). For a state-control pair (i, u), the average cost per stage is denoted by $g(i, u, j)$.

We recall that the PI algorithm in its exact form produces a sequence of stationary policies whose cost functions are progressively improving and converge in a finite number of iterations to the optimal. The corresponding convergence proof relies on the generic cost improvement property of PI, and depends on the finiteness of the state and control spaces. This proof, together with other PI-related convergence proofs, can be found in the author's textbooks [Ber17a] or [Ber19a].

Let us state the exact form of the PI algorithm in terms of Q-factors, and in a form that is suitable for the use of approximations and simulation-based implementations. Given any policy μ, it generates the next policy $\tilde{\mu}$ with a two-step process as follows (cf. Section 1.4.1):

(a) *Policy evaluation*: We compute the cost function J_μ of μ and its associated Q-factors, which are given by

$$Q_\mu(i,u) = \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha J_\mu(j)\big), \qquad i = 1, \ldots, n, \ u \in U(i).$$

Thus $Q_\mu(i, u)$ is the cost of starting at i, using u at the first stage, and then using μ for the remaining stages.

(b) *Policy improvement*: We compute the new policy $\tilde{\mu}$ according to

$$\tilde{\mu}(i) \in \arg\min_{u \in U(i)} Q_\mu(i,u), \qquad i = 1, \ldots, n.$$

Let us now describe one way to approximate the two steps of the preceding process.

(a) *Approximate policy evaluation*: Here we introduce a parametric architecture $\tilde{Q}_\mu(i, u, r)$ for the Q-factors of μ. We determine the value of the parameter vector r by generating (using a simulator of the system) a large number of training triplets (i^s, u^s, β^s), $s = 1, \ldots, q$, and

by using a least squares fit:

$$\overline{r} \in \arg\min_{r} \sum_{s=1}^{q} \big(\tilde{Q}_\mu(i^s, u^s, r) - \beta^s\big)^2. \tag{3.23}$$

In particular, for a given pair (i^s, u^s), the scalar β^s is generated by starting at i^s, using u^s at the first stage, and simulating a trajectory of states and controls using μ for some number k of subsequent stages. Thus, β^s is a sample of $Q^k_\mu(i^s, u^s)$, the k-stage Q-factor of μ, which in the limit as $k \to \infty$ yields the infinite horizon Q-factor of μ. The number of stages k may be either large, or fairly small. However, in the latter case some terminal cost function approximation should be added at the end of the k-stage trajectory, to compensate for the difference $\big|Q_\mu(i,u) - Q^k_\mu(i,u)\big|$, which decreases in proportion to α^k, and may be large when k is small. Such a function may be obtained with additional training or from a previous iteration.

(b) *Approximate policy improvement*: Here we compute the new policy $\tilde{\mu}$ according to

$$\tilde{\mu}(i) \in \arg\min_{u \in U(i)} \tilde{Q}_\mu(i, u, \overline{r}), \qquad i = 1, \ldots, n, \tag{3.24}$$

where $\overline{r}$ is the parameter vector obtained from the policy evaluation formula (3.23).

An important alternative for approximate policy improvement, is to compute a set of pairs $\big(i^s, \tilde{\mu}(i^s)\big)$, $s = 1, \ldots, q$, using Eq. (3.24), and fit these pairs with a policy approximation architecture (see the next section on approximation in policy space). The overall scheme then becomes policy iteration that is based on approximation in both value and policy spaces.

At the end of the last policy evaluation step of PI, we have obtained a final Q-factor approximation $\tilde{Q}(i, u, \tilde{r})$. Then, in on-line play mode, we may apply the policy

$$\tilde{\mu}(i) \in \arg\min_{u \in U(i)} \tilde{Q}(i, u, \tilde{r}),$$

i.e., use the (would be) next policy iterate. Alternatively, we may apply the one-step lookahead policy

$$\tilde{\mu}(i) \in \arg\min_{u \in U(i)} \left[\sum_{j=1}^{n} p_{ij}(u) \left(g(i,u,j) + \alpha \min_{u' \in U(j)} \tilde{Q}(j, u', \tilde{r})\right)\right], \tag{3.25}$$

or its multistep lookahead version. The latter alternative implements a Newton step and will likely result in substantially better performance.

However, it is more time consuming, particularly if it is implemented by using a computer model and model-free simulation. Still another possibility, which also implements a Newton step, is to replace the function

$$\min_{u' \in U(j)} \tilde{Q}(j, u', \tilde{r})$$

in the preceding Eq. (3.25) with an off-line trained approximation.

Issues Relating to Approximate Policy Iteration

Approximate PI in its various forms has been the subject of extensive research, both theoretical and applied. A more detailed discussion is beyond our scope, and we refer to the literature, as well as Chapters 6 and 7 of the DP textbook [Ber12] or the RL textbook [Ber19a] for detailed accounts. Let us provide a few comments.

(a) *Architectural issues*: The architecture $\tilde{Q}_\mu(i, u, r)$ may involve the use of features, and it could be linear, or it could be nonlinear such as a neural network. A major advantage of a linear feature-based architecture is that the policy evaluation (3.23) is a linear least squares problem, which admits a closed-form solution. Moreover, when linear architectures are used, there is a broader variety of approximate policy evaluation methods with solid theoretical performance guarantees, such as TD(λ), LSTD(λ), and LSPE(λ), which are not described in these notes, but are discussed extensively in the literature, including the DP textbook [Ber12] and the RL textbook [Ber19a].

(b) *Exploration issues*: Generating an appropriate set of training triplets (i^s, u^s, β^s) at the policy evaluation step poses considerable challenges, and the literature contains several related proposals. A generic difficulty has to do with *inadequate exploration*. In particular, to evaluate a policy μ, we may need to generate Q-factor samples of μ starting from states frequently visited by μ, but this may bias the simulation by underrepresenting states that are unlikely to occur under μ. As a result, the Q-factor estimates of these underrepresented states may be highly inaccurate, causing potentially serious errors in the calculation of the improved control policy $\tilde{\mu}$ via the policy improvement Eq. (3.24).

One possibility to improve the exploration of the state space is to use a large number of initial states to form a rich and representative subset. It may then be necessary to use relatively short trajectories to keep the cost of the simulation low. However, when using short trajectories it may be important to introduce a terminal cost function approximation in the policy evaluation step in order to make the cost sample β^s more accurate. There have been other related approaches to improve exploration, such as using a so-called *off-policy*, i.e., a

policy μ' other than the currently evaluated policy μ, to visit states that are unlikely to be visited using μ. See the discussions in Section 6.4 of the DP textbook [Ber12].

(c) *Oscillation issues*: Contrary to exact PI, which is guaranteed to yield an optimal policy, approximate PI produces a sequence of policies, which are only guaranteed to lie asymptotically within a certain error bound from the optimal; see the books [BeT96], Section 6.2.2, and [Ber12], Section 2.5. Moreover, the generated policies may oscillate. By this we mean that after a few iterations, policies tend to repeat in cycles.

The associated parameter vectors r may also tend to oscillate, although it is possible that there is convergence in parameter space and oscillation in policy space. This phenomenon, known as *chattering*, is explained in the author's survey paper [Ber11b], and book [Ber12] (Section 6.4.3), and can be particularly damaging, because there is no guarantee that the policies involved in the oscillation are "good" policies, and there is often no way to verify how well they perform relative to the optimal. We note, however, that oscillations can be avoided and approximate PI can be shown to converge under special conditions, which arise in particular when an aggregation approach is used; see the approximate PI survey [Ber11b].

We refer to the literature for further discussion of the preceding issues, as well as a variety of other approximate PI methods.

3.3.4 Optimistic Policy Iteration with Parametric Q-Factor Approximation - SARSA and DQN

There are also "optimistic" approximate PI methods with Q-factor approximation, and/or a few samples in between policy updates. Because of the use of Q-factors and the limited number of samples between policy updates, these schemes have the potential of on-line play implementation, but a number of difficulties must be overcome in this case, as we will explain later in this section.

As an example, let us consider an extreme version of Q-factor parametric approximation that uses *a single sample* between policy updates. At the start of iteration k, we have the current parameter vector r^k, we are at some state i^k, and we have chosen a control u^k. Then:

(1) We simulate the next transition (i^k, i^{k+1}) using the transition probabilities $p_{i^k j}(u^k)$.

(2) We generate the control u^{k+1} with the minimization

$$u^{k+1} \in \arg \min_{u \in U(i^{k+1})} \tilde{Q}(i^{k+1}, u, r^k). \tag{3.26}$$

[In some schemes, to enhance exploration, u^{k+1} is chosen with a small probability to be a random element of $U(i^{k+1})$ or one that is "ϵ-greedy," i.e., attains within some ϵ the minimum above. This is commonly referred to as the use of an *off-policy*.]

(3) We update the parameter vector via

$$r^{k+1} = r^k - \gamma^k \nabla \tilde{Q}(i^k, u^k, r^k) \cdot \big(\tilde{Q}(i^k, u^k, r^k) - g(i^k, u^k, i^{k+1}) - \alpha \tilde{Q}(i^{k+1}, u^{k+1}, r^k)\big), \tag{3.27}$$

where γ^k is a positive stepsize, and $\nabla(\cdot)$ denotes gradient with respect to r evaluated at the current parameter vector r^k. To get a sense for the rationale of this iteration, note that if $\tilde{Q}$ is a linear feature-based architecture, $\tilde{Q}(i, u, r) = \phi(i, u)'r$, then $\nabla \tilde{Q}(i^k, u^k, r^k)$ is just the feature vector $\phi(i^k, u^k)$, and iteration (3.27) becomes

$$r^{k+1} = r^k - \gamma^k \phi(i^k, u^k)\big(\phi(i^k, u^k)'r^k - g(i^k, u^k, i^{k+1}) - \alpha \phi(i^{k+1}, u^{k+1})'r^k\big).$$

Thus r^k *is changed in an incremental gradient direction*: the one opposite to the gradient (with respect to r) of the incremental error

$$\big(\phi(i^k, u^k)'r - g(i^k, u^k, i^{k+1}) - \alpha \phi(i^{k+1}, u^{k+1})'r^k\big)^2,$$

evaluated at the current iterate r^k.

The process is now repeated with r^{k+1}, i^{k+1}, and u^{k+1} replacing r^k, i^k, and u^k, respectively.

Extreme optimistic schemes of the type just described have received a lot of attention, in part because *they admit a model-free implementation* [i.e., the use of a computer simulator, which provides for each pair (i^k, u^k), the next state i^{k+1} and corresponding cost $g(i^k, u^k, i^{k+1})$ that are needed in Eq. (3.27)]. They are often referred to as SARSA (State-Action-Reward-State-Action); see e.g., the books [BeT96], [BBD10], [SuB18]. When Q-factor approximation is used, their behavior is very complex, their theoretical convergence properties are unclear, and there are no associated performance bounds in the literature. In practice, SARSA is more commonly used in a less extreme/optimistic form, whereby several (perhaps many) state-control-transition cost-next state samples are batched together and suitably averaged before updating the vector r^k.

Other variants of the method attempt to save in sampling effort by storing the generated samples in a buffer and reusing them in some randomized fashion in subsequent iterations (cf. our earlier discussion of exploration. This is also called sometimes *experience replay*, an idea that has been used since the early days of RL, both to save in sampling effort and to enhance exploration. The DQN (Deep Q Network) scheme, championed by DeepMind (see Mnih et al. [MKS15]), is based on this idea (the term "Deep" is a reference to DeepMind's affinity for deep neural networks, but experience replay does not depend on the use of a deep neural network architecture).

Q-Learning Algorithns and On-Line Play

Algorithms that approximate Q-factors, including SARSA and DQN, are fundamentally off-line training algorithms, primarily because their training process is long and requires the collection of many samples before reaching a stage that resembles parameter convergence. It can therefore be unreliable to use the interim approximate Q-factors for on-line decision making, particularly in an adaptive context that involves changing system parameters, thereby requiring on-line replanning.

On the other hand, compared to the approximate PI method of Section 3.3.3, SARSA and DQN are far better suited for on-line implementation, because the control generation process of Eq. (3.26) can also be used to select controls on-line, thereby facilitating the combination of training and on-line control selection. To this end, it is important, among others, to make sure that the parameters r_k stay at reasonable levels during the on-line control process, which can be a challenge. Still, even if this difficulty can be overcome, there are a number of other difficulties that SARSA and DQN can encounter during on-line play.

(a) *On-line exploration issues*: The need to occasionally select controls using an off-policy in order to enhance exploration. Finding an off-line policy that adequately deals with exploration in a given practical context can be a challenge. Moreover, an additional concern is that the off-policy controls may improve exploration, but may be of poor quality, and in some contexts, may induce instability.

(b) *Robustness and replanning issues*: In an adaptive control context where the problem parameters are changing, the algorithm may be too slow to adapt to the changes.

(c) *Performance degradation issues*: Similar to our earlier discussion [cf. the comparison of Eqs. (3.24) and (3.25)], the minimization of Eq. (3.26) does not implement a Newton step, thereby resulting in performance loss. The alternative implementation

$$u^{k+1} \in \arg\min_{u \in U(i^{k+1})} \left[\sum_{j=1}^{n} p_{i^{k+1}j}(u) \Big(g(i^{k+1}, u, j) + \alpha \min_{u' \in U(j)} \tilde{Q}(j, u', r^k) \Big) \right],$$

which is patterned after Eq. (3.26), is better in this regard, but is computationally more costly, and thus less suitable for on-line implementation.

Generally speaking, the combination of off-line training and on-line play with the use of SARSA and DQN involves serious challenges. However,

in some specific contexts encouraging results have been obtained. Moreover, the methods have received a lot of attention, thanks in part to the availability of publicly available software, which also allow for a model-free implementation.

3.3.5 Approximate Policy Iteration for Infinite Horizon POMDP

In this section, we consider partial observation Markovian decision problems (POMDP) with a finite number of states and controls, and discounted additive cost over an infinite horizon. As discussed in Section 1.6.4, the optimal solution is typically intractable, so approximate DP/RL approaches must be used. In this section we focus on PI methods that are based on rollout, and approximations in policy and value space. They update a policy by using truncated rollout with that policy and a terminal cost function approximation. We focus on cost function approximation schemes, but Q-factor approximation is also possible.

Because of its simulation-based rollout character, the methodology of this section depends critically on the finiteness of the control space. It can be extended to POMDP with infinite state space but finite control space, although we will not consider this possibility in this section. In particular, we assume that there are n states denoted by $i \in \{1, \ldots, n\}$ and a finite set of controls U at each state. We denote by

$$p_{ij}(u) \quad \text{and} \quad g(i, u, j)$$

the transition probabilities and corresponding transition costs, from i to j under $u \in U$. The cost is accumulated over an infinite horizon and is discounted by $\alpha \in (0, 1)$. At each new generated state j, an observation z from a finite set Z is obtained with known probability $p(z \mid j, u)$ that depends on j and the control u that was applied prior to the generation of j. The objective is to select each control optimally as a function of the prior history of observations and controls.

A classical approach to this problem is to convert it to a perfect state information problem whose state is the current belief $b = (b_1, \ldots, b_n)$, where b_i is the conditional distribution of the state i given the prior history. As noted in Section 1.6.4, b is a sufficient statistic, which can serve as a substitute for the set of available observations, in the sense that optimal control can be achieved with knowledge of just b.

In this section, we consider a more general form of sufficient statistic, which we call the *feature state* and we denote by y. *We require that the feature state y subsumes the belief state b.* By this we mean that b can be computed exactly knowing y. One possibility is for y to be the union of b and some identifiable characteristics of the belief state, or some compact representation of the measurement history up to the current time (such as a number of most recent measurements, or the state of a finite-state controller). We also make the additional assumption that y *can be*

sequentially generated using a known feature estimator $F(y, u, z)$. By this we mean that given that the current feature state is y, control u is applied, and observation z is obtained, the next feature can be exactly predicted as $F(y, u, z)$.

Clearly, since b is a sufficient statistic, the same is true for y. Thus the optimal costs achievable by the policies that depend on y and on b are the same. However, specific suboptimal schemes may become more effective with the use of the feature state y instead of just the belief state b.

The optimal cost $J^*(y)$, as a function of the sufficient statistic/feature state y, is the unique solution of the corresponding Bellman equation

$$J^*(y) = \min_{u \in U} \left[\hat{g}(y, u) + \alpha \sum_{z \in Z} \hat{p}(z \mid b_y, u) J^*\big(F(y, u, z)\big) \right].$$

Here we use the following notation:

b_y is the belief state that corresponds to feature state y, with components denoted by $b_{y,i}$, $i = 1, \ldots, n$.

$\hat{g}(y, u)$ is the expected cost per stage

$$\hat{g}(y, u) = \sum_{i=1}^{n} b_{y,i} \sum_{j=1}^{n} p_{ij}(u) g(i, u, j).$$

$\hat{p}(z \mid b_y, u)$ is the conditional probability that the next observation will be z given the current belief state b_y and control u

F is the feature state estimator. In particular, $F(y, u, z)$ is the next feature vector, when the current feature state is y, control u is applied, and observation z is obtained.

The feature space reformulation of the problem can serve as the basis for approximation in value space, whereby J^* is replaced in Bellman's equation by some function $\tilde{J}$ after one-step or multistep lookahead. For example a one-step lookahead scheme yields the suboptimal policy $\tilde{\mu}$ given by

$$\tilde{\mu}(y) \in \arg\min_{u \in U} \left[\hat{g}(y, u) + \alpha \sum_{z \in Z} \hat{p}(z \mid b_y, u) \tilde{J}\big(F(y, u, z)\big) \right].$$

In ℓ-step lookahead schemes, $\tilde{J}$ is used as terminal cost function in an ℓ-step horizon version of the original infinite horizon problem. In the standard form of a rollout algorithm, $\tilde{J}$ is the cost function of some base policy. We will next discuss a rollout scheme with ℓ-step lookahead, which involves rollout truncation and terminal cost approximation.

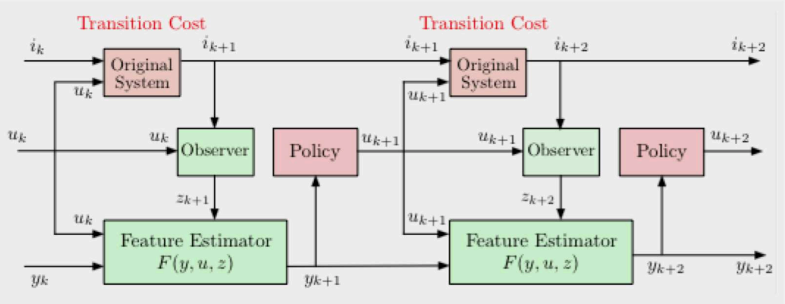


Figure 3.3.1 Composite system simulator for POMDP for a given policy. The starting state i_k at stage k of a trajectory is generated randomly using the belief state b_k, which is in turn computed from the feature state y_k.

Truncated Rollout with Terminal Cost Function Approximation

In the pure form of the rollout algorithm, the cost function approximation $\tilde{J}$ is the cost function J_μ of a known base policy μ, and its value $\tilde{J}(y) = J_\mu(y)$ at any y is obtained by first extracting b from y, and then running a simulator starting from b, and using the system model, the feature generator, and μ. In the truncated form of rollout, $\tilde{J}(y)$ is obtained by running the simulator of μ for a given number of steps m, and then adding a terminal cost approximation $\hat{J}(\bar{y})$ for each terminal feature state $\bar{y}$ that is obtained at the end of the m steps of the simulation with μ (see Fig. 3.3.1).

Thus the rollout policy is defined by the base policy μ, the terminal cost function approximation $\hat{J}$, the number of steps m after which the simulated trajectory with μ is truncated, as well as the lookahead size ℓ. The choices of m and ℓ are typically made by trial and error, based on computational tractability among other considerations, while $\hat{J}$ may be chosen on the basis of problem-dependent insight or through the use of some off-line approximation method. In variants of the method, the multistep lookahead may be implemented approximately using a Monte Carlo tree search or adaptive sampling scheme.

Using m-step rollout between the ℓ-step lookahead and the terminal cost approximation gives the method the character of a single PI. We will use repeated truncated rollout as the basis for constructing a PI algorithm, which we will discuss next.

Supervised Learning of Rollout Policies and Cost Functions - Approximate Policy Iteration

The rollout algorithm uses multistep lookahead and on-line simulation of the base policy to generate the rollout control at any feature state of interest. To avoid the cost of on-line simulation, we can approximate the rollout

policy off-line by using some approximation architecture, which may involve a neural network. This is policy approximation built on top of the rollout scheme.

To this end, we may introduce a parametric family/architecture of policies of the form $\hat{\mu}(y, r)$, where r is a parameter vector. We then construct a training set that consists of a large number of sample feature state-control pairs (y^s, u^s), $s = 1, \ldots, q$, such that for each s, u^s is the rollout control at feature state y^s. We use this data set to obtain a parameter $\bar{r}$ by solving a corresponding classification problem, which associates each feature state y with a control $\hat{\mu}(y, \bar{r})$. The parameter $\bar{r}$ defines a classifier, which given a feature state y, classifies y as requiring control $\hat{\mu}(y, \bar{r})$ (see Section 3.4).

We can also apply the rollout policy approximation to the context of PI. The idea is to view rollout as a single policy improvement, and to view the PI algorithm as a *perpetual rollout process*, which performs multiple policy improvements, using at each iteration the current policy as the base policy, and the next policy as the corresponding rollout policy.

In particular, we consider a PI algorithm where at the typical iteration we have a policy μ, which we use as the base policy to generate many feature state-control sample pairs (y^s, u^s), $s = 1, \ldots, q$, where u^s is the rollout control corresponding to feature state y^s. We then obtain an "improved" policy $\hat{\mu}(y, \overline{r})$ with an approximation architecture and a classification algorithm, as described above. The "improved" policy is then used as a base policy to generate samples of the corresponding rollout policy, which is approximated in policy space, etc.

To use truncated rollout in this PI scheme, we must also provide a terminal cost approximation, which may take a variety of forms. Using zero is a simple possibility, which may work well if either the size ℓ of multistep lookahead or the length m of the rollout is relatively large. Another possibility is to use as terminal cost in the truncated rollout an approximation of the cost function of some base policy, which may be obtained with a neural network-based approximation architecture.

In particular, at any policy iteration with a given base policy, once the rollout data is collected, one or two neural networks are constructed: A policy network that approximates the rollout policy, and (in the case of rollout with truncation) a value network that constructs a cost function approximation for that rollout policy. Thus, we may consider two types of methods:

(a) *Approximate rollout and PI with truncation*, where each generated policy as well as its cost function are approximated by a policy and a value network, respectively. The cost function approximation of the current policy is used to truncate the rollout trajectories that are used to train the next policy.

(b) *Approximate rollout and PI without truncation*, where each gener-

ated policy is approximated using a policy network, but the rollout trajectories are continued up to a large maximum number of stages (enough to make the cost of the remaining stages insignificant due to discounting) or upon reaching a termination state. The advantage of this scheme is that only a policy network is needed; a value network is unnecessary since there is no rollout truncation with cost function approximation at the end.

Note that as in all approximate PI schemes, the sampling of feature states used for training is subject to exploration concerns. In particular, for each policy approximation, it is important to include in the sample set $\{y^s \mid s = 1, \ldots, q\}$, a subset of feature states that are "favored" by the rollout trajectories; e.g., start from some initial subset of feature states y^s and selectively add to this subset feature states that are encountered along the rollout trajectories. This is a challenging issue, which must be approached with care.

An extensive case study of the methodology of this section was given in the paper by Bhattacharya et al. [BBW20], for the case of a pipeline repair problem. The implementation used there also includes the use of a partitioned state space architecture and an asynchronous distributed algorithm for off-line training; see Section 3.4.2.

3.3.6 Advantage Updating - Approximating Q-Factor Differences

Let us now focus on an important alternative to computing Q-factor approximations. It is motivated by the potential benefit of approximating Q-factor differences rather than Q-factors. In this method, called *advantage updating*, instead of computing and comparing $Q_k^*(x_k, u_k)$ for all $u_k \in U_k(x_k)$, we compute

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

The function $A_k(x_k, u_k)$ can serve to compare controls, i.e., at state x_k select

$$\tilde{\mu}_k(x_k) \in \arg \min_{u \in U_k(x_k)} A_k(x_k, u),$$

and this can also be done when $A_k(x_k, u_k)$ is approximated with a value network.

Note that in the absence of approximations, selecting controls by advantage updating is clearly equivalent to selecting controls by comparing their Q-factors. By contrast, when approximation is involved, comparing advantages instead of Q-factors can be important, because the former may have a much smaller range of values than the latter. In particular, Q_k^* may embody sizable quantities that depend on x_k but are independent of u_k, and which may interfere with algorithms such as the fitted value iteration

(3.19)-(3.20). Thus, when training an architecture to approximate Q_k^*, the training algorithm may naturally try to capture the large scale behavior of Q_k^*, which may be irrelevant because it may not be reflected in the Q-factor differences A_k. However, with advantage updating, we may instead focus the training process on finer scale variations of Q_k^*, which may be all that matters. Here is an example (first given in the book [BeT96]) of what can happen when trained approximations of Q-factors are used.

Example 3.3.1

Consider the deterministic scalar linear system

$$x_{k+1} = x_k + \delta u_k,$$

and the quadratic cost per stage

$$g(x_k, u_k) = \delta(x_k^2 + u_k^2),$$

where δ is a very small positive constant [think of δ-discretization of a continuous-time problem involving the differential equation $dx(t)/dt = u(t)$]. Let us focus on the stationary policy π, which applies at state x the control

$$\mu(x) = -2x,$$

and view it as the base policy of a rollout algorithm. The Q-factors of π over an infinite number of stages can be calculated to be

$$Q_\pi(x, u) = \frac{5x^2}{4} + \delta\left(\frac{9x^2}{4} + u^2 + \frac{5}{2}xu\right) + O(\delta^2).$$

(We omit the details of this calculation, which is based on the classical analysis of linear-quadratic optimal control problems; see e.g., Section 1.5, or [Ber17a], Section 3.1.) Thus the important part of $Q_\pi(x, u)$ for the purpose of rollout policy computation is

$$\delta\left(u^2 + \frac{5}{2}xu\right). \tag{3.28}$$

However, when a value network is trained to approximate $Q_\pi(x, u)$, the approximation will be dominated by $\frac{5x^2}{4}$, and the important part (3.28) will be "lost" when δ is very small. By contract, the advantage function can be calculated to be

$$\begin{aligned} A_\mu(x, u) &= Q_\pi(x, u) - \min_v Q_\pi(x, v) + O(\delta^2) \\ &= \delta\left(u^2 + \frac{5}{2}xu - \min_v \left(v^2 + \frac{5}{2}xv\right)\right) + O(\delta^2) \\ &= \delta\left(u^2 + \frac{5}{2}xu + \frac{5^2}{4}x^2\right) + O(\delta^2), \end{aligned}$$

and when approximated with a value network, the approximation will be essentially unaffected by δ.

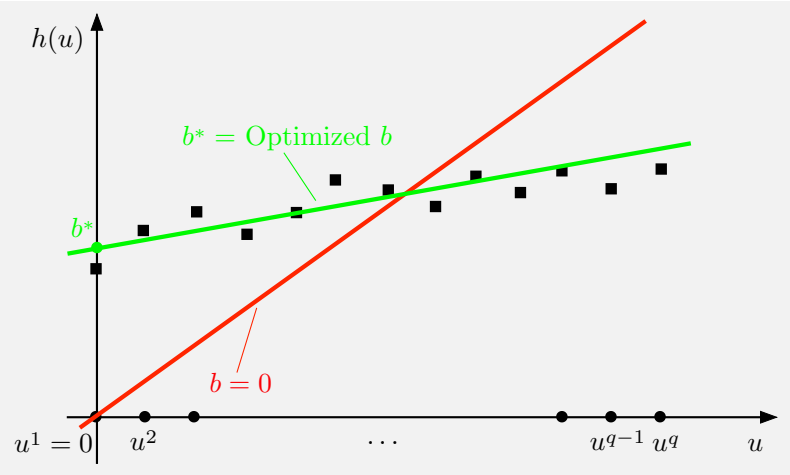


Figure 3.3.2 Illustration of the idea of subtracting a baseline constant from a cost or Q-factor approximation. Here we have samples $h(u^1), \ldots, h(u^q)$ of a scalar function $h(u)$ at sample points $u^1, \ldots, u^q$, and we want to approximate $h(u)$ with a linear function $\tilde{h}(u, r) = ru$, where r is a scalar tunable weight. We subtract a baseline constant b from the samples, and we solve the problem

$$\bar{r} \in \arg\min_r \sum_{s=1}^{q} \Big(\big(h(u^s) - b \big) - ru^s \Big)^2.$$

By properly adjusting b, we can improve the quality of the approximation, which after subtracting b from all the sample values, takes the form $\tilde{h}(u, b, r) = b + ru$. Conceptually, b serves as an additional weight (multiplying the basis function 1), which enriches the approximation architecture.

The Use of a Baseline

The idea of advantage updating is also related to the useful technique of subtracting a suitable constant (often called a *baseline*) from a quantity that is estimated; see Fig. 3.3.2 (in the case of advantage updating, the baselines depend on x_k, but the same general idea applies). This idea can also be used in the context of the fitted value iteration method given earlier, as well as in conjunction with other simulation-based methods in RL.

Example 3.1.1 also points to the connection between the ideas underlying advantage updating and the rollout methods for small stage costs relative to the cost function approximation, which we discussed in Section 2.6. In both cases it is necessary to avoid including terms of disproportionate size in the target function that is being approximated. The remedy in both cases is to subtract from the target function a suitable state-dependent baseline.

3.3.7 Differential Training of Cost Differences for Rollout

Let us now consider ways to approximate Q-factor differences (cf. our advantage updating discussion of the preceding section) by approximating cost function differences first. We recall here that given a base policy $\pi = \{\mu_0, \ldots, \mu_{N-1}\}$, the off-line computation of an approximate rollout policy $\tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$ consists of two steps:

(1) In a preliminary phase, we compute approximations $\tilde{J}_k$ to the cost functions $J_{k,\pi}$ of the base policy π, possibly using simulation and a least squares fit from a parametrized class of functions.

(2) Given $\tilde{J}_k$ and a state x_k at time k, we compute the approximate Q-factor

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

for all $u \in U_k(x_k)$, and we obtain the (approximate) rollout control $\tilde{\mu}_k(x_k)$ from the minimization

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

Unfortunately, this method also suffers from the error magnification inherent in the Q-factor differencing operation. This motivates an alternative approach, called *differential training*, which is based on cost-to-go difference approximations. To this end, we note that to compute the rollout control $\tilde{\mu}_k(x_k)$, it is sufficient to have the differences of costs-to-go

$$\tilde{J}_{k+1}\big(f_k(x_k, u, w_k)\big) - \tilde{J}_{k+1}\big(f_k(x_k, \mu_k(x_k), w_k)\big), \tag{3.29}$$

where $\mu_k(x_k)$ is the control applied by the base policy at x_k.

We thus consider a function approximation approach, whereby given any two states x_{k+1} and $\hat{x}_{k+1}$, we obtain an approximation $\tilde{G}_{k+1}(x_{k+1}, \hat{x}_{k+1})$ of the cost difference (3.29). We then compute the rollout control by

$$\tilde{\mu}_k(x_k) \in \arg\min_{u \in U_k(x_k)} E\Big\{g_k(x_k, u, w_k) - g_k(x_k, \mu_k(x_k), w_k) + \tilde{G}_{k+1}\big(f_k(x_k, u, w_k), f_k(x_k, \mu_k(x_k), w_k)\big)\Big\}, \tag{3.30}$$

where $\mu_k(x_k)$ is the control applied by the base policy at x_k. Note that the minimization (3.30) aims to simply subtract the approximate Q-factor of the base policy control $\mu_k(x_k)$ from the approximate Q-factor of every other control $u \in U_k(x_k)$.

An important point here is that the training of an approximation architecture to obtain $\tilde{G}_{k+1}$ can be done using any of the standard training

methods, and a "differential" system, whose "states" are pairs $(x_k, \hat{x}_k)$ and will be described shortly. To see this, let us denote for all k and pair of states $(x_k, \hat{x}_k)$

$$G_k(x_k, \hat{x}_k) = J_{k,\pi}(x_k) - J_{k,\pi}(\hat{x}_k)$$

the cost function differences corresponding to the base policy π. We consider the DP equations corresponding to π, and to x_k and $\hat{x}_k$:

$$J_{k,\pi}(x_k) = E\Big\{g_k\big(x_k, \mu_k(x), w_k\big) + J_{k+1,\pi}\big(f_k(x_k, \mu_k(x), w_k)\big)\Big\},$$

$$J_{k,\pi}(\hat{x}_k) = E\Big\{g_k\big(\hat{x}_k, \mu_k(\hat{x}_k), w_k\big) + J_{k+1,\pi}\big(f_k(\hat{x}_k, \mu_k(\hat{x}_k), w_k)\big)\Big\},$$

and we subtract these equations to obtain

$$\begin{aligned} G_k(x_k, \hat{x}_k) = E\Big\{&g_k\big(x_k, \mu_k(x_k), w_k\big) - g\big(\hat{x}_k, \mu_k(\hat{x}_k), w_k\big) \\ &+ G_{k+1}\big(f_k(x_k, \mu_k(x_k), w_k), f_k(\hat{x}_k, \mu_k(\hat{x}_k), w_k)\big)\Big\}, \end{aligned}$$

for all $(x_k, \hat{x}_k)$ and k. Therefore, G_k can be viewed as the cost-to-go function for a problem involving a fixed policy (the base policy), the state $(x_k, \hat{x}_k)$, the cost per stage

$$g_k\big(x_k, \mu_k(x_k), w_k\big) - g_k\big(\hat{x}_k, \mu_k(\hat{x}_k), w_k\big), \tag{3.31}$$

and the system equation

$$(x_{k+1}, \hat{x}_{k+1}) = \Big(f_k\big(x_k, \mu_k(x_k), w_k\big), f_k\big(\hat{x}_k, \mu_k(\hat{x}_k), w_k\big)\Big). \tag{3.32}$$

Thus, it can be seen that *any of the standard methods that can be used to train architectures that approximate $J_{k,\pi}$, can also be used for training architectures that approximate G_k*. For example, one may use simulation-based methods that generate pairs of trajectories starting at the pair of initial states $(x_k, \hat{x}_k)$, and generated according to Eq. (3.32) by using the base policy π. Note that a single random sequence $\{w_0, \ldots, w_{N-1}\}$ may be used to simultaneously generate samples of $G_k(x_k, \hat{x}_k)$ for several triples $(x_k, \hat{x}_k, k)$, and in fact this may have a substantial beneficial effect.

A special case of interest arises when a linear, feature-based architecture is used for the approximator $\tilde{G}_k$. In particular, let ϕ_k be a feature extraction mapping that associates a feature vector $\phi_k(x_k)$ with state x_k and time k, and let $\tilde{G}_k$ be of the form

$$\tilde{G}_k(x_k, \hat{x}_k) = r_k'\big(\phi_k(x_k) - \phi_k(\hat{x}_k)\big),$$

where r_k is a tunable weight vector of the same dimension as $\phi_k(x_k)$ and prime denotes transposition. The rollout policy is generated by

$$\tilde{\mu}_k(x_k) \in \arg\min_{u \in U_k(x_k)} E\Big\{g_k(x_k, u, w_k) + r_{k+1}'\phi_{k+1}\big(f_k(x_k, u, w_k)\big)\Big\},$$

which corresponds to using $r'_{k+1}\phi_{k+1}(x_{k+1})$ (plus an unknown inconsequential constant) as an approximation to $J_{k+1,\pi}(x_{k+1})$. Thus, in this approach, *we essentially use a linear feature-based architecture to approximate the cost functions $J_{k,\pi}$ of the base policy, but we train this architecture using the differential system (3.32) and the differential cost per stage of Eq. (3.31).* This is done by selecting pairs of initial states, running in parallel the corresponding trajectories using the base policy, and subtracting the resulting trajectory costs from each other.

3.4 TRAINING OF POLICIES IN APPROXIMATE DP

We have focused so far on approximation in value space using parametric architectures. In this section we will discuss briefly how the cost function approximation methods of this chapter can be suitably adapted for the purpose of approximation in policy space, whereby we select the policy by using optimization over a parametric family of some form.

In particular, suppose that for a given stage k, we have access to a dataset of sample state-control pairs (x_k^s, u_k^s), $s = 1, \ldots, q$, obtained through some unspecified process, such as rollout or problem approximation. We may then wish to "learn" this process by training the parameter vector r_k of a parametric family of policies $\tilde{\mu}_k(x_k, r_k)$, using least squares minimization/regression:

$$\overline{r}_k \in \arg\min_{r_k} \sum_{s=1}^{q} \left\| u_k^s - \tilde{\mu}_k(x_k^s, r_k) \right\|^2; \tag{3.33}$$

cf. our discussion of approximation in policy space in Section 1.3.3.

3.4.1 The Use of Classifiers for Approximation in Policy Space

As we have noted in Section 3.1, in the case of a continuous control space, training of a parametric architecture for policy approximation is similar to training for a cost approximation. In the case where the control space is finite, however, it is useful to make the connection of approximation in policy space with *classification*; cf. Fig. 3.1.2 and the discussion of Section 3.1.

Classification is an important subject in machine learning. The objective is to construct an algorithm, called a *classifier*, which assigns a given "object" to one of a finite number of "categories" based on its "characteristics." Here we use the term "object" generically. In some cases, the classification may relate to persons or situations. In other cases, an object may represent a hypothesis, and the problem is to decide which of the hypotheses is true, based on some data. In the context of approximation in policy space, *objects correspond to states, and categories correspond to*

controls to be applied at the different states. Thus in this case, we view each sample $\big(x_k^s, u_k^s\big)$ as an object-category pair.

Generally, in (multiclass) classification we assume that we have a population of objects, each belonging to one of m categories $c = 1, \ldots, m$. We want to be able to assign a category to any object that is presented to us. Mathematically, we represent an object with a vector x (e.g., some raw description or a vector of features of the object), and we aim to construct a rule that assigns to every possible object x a unique category c.

To illustrate a popular classification method, let us assume that if we draw an object x at random from this population, the conditional probability of the object being of category c is $p(c \,|\, x)$. If we know the probabilities $p(c \,|\, x)$, we can use a classical statistical approach, whereby we assign x to the category $c^*(x)$ that has maximal posterior probability, i.e.,

$$c^*(x) \in \arg \max_{c=1,\ldots,m} p(c \,|\, x). \tag{3.34}$$

This is called the Maximum a Posteriori rule (or MAP rule for short; see for example the book [BeT08], Section 8.2, for a discussion).

When the probabilities $p(c \,|\, x)$ are unknown, we may try to estimate them using a least squares optimization, based on the following property, whose proof is outlined in Exercise 4.1; see also [Ber19a], Section 3.5.

Proposition 3.4.1: (Least Squares Property of Conditional Probabilities) Let $\xi(x)$ be any prior distribution of x, so that the joint distribution of (c, x) is

$$\zeta(c, x) = \sum_x \xi(x) \sum_{c=1}^m p(c \,|\, x).$$

Let $z(c, x)$ be the function of (c, x) defined by

$$z(c, x) = \begin{cases} 1 & \text{if } x \text{ is of category } c, \\ 0 & \text{otherwise.} \end{cases}$$

For any function $h(c, x)$ of (c, x), consider

$$E\Big\{ \big(z(c, x) - h(c, x)\big)^2 \Big\},$$

the expected value with respect to the distribution $\zeta(c, x)$ of the random variable $\big(z(c, x) - h(c, x)\big)^2$. Then $p(c \,|\, x)$ minimizes this expected value over all functions $h(c, x)$, i.e., for all functions h, we have

$$E\Big\{\big(z(c,x)-p(c\,|\,x)\big)^2\Big\} \le E\Big\{\big(z(c,x)-h(c,x)\big)^2\Big\}. \qquad (3.35)$$

The proposition states that $p(c\,|\,x)$ is the function of (c,x) that minimizes

$$E\Big\{\big(z(c,x)-h(c,x)\big)^2\Big\} \qquad (3.36)$$

over all functions h of (c,x), *independently of the prior distribution of* x. This suggests that we can obtain approximations to the probabilities $p(c\,|\,x)$, $c=1,\ldots,m$, by minimizing an empirical/simulation based approximation of the expected value (3.36).

More specifically, let us assume that we have a training set consisting of q object-category pairs (x^s,c^s), $s=1,\ldots,q$, and corresponding vectors

$$z^s(c)=\begin{cases}1 & \text{if } c^s=c,\\ 0 & \text{otherwise,}\end{cases} \qquad c=1,\ldots,m,$$

and adopt a parametric approach. In particular, for each category $c=1,\ldots,m$, we approximate the probability $p(c\,|\,x)$ with a function $\tilde h(c,x,r)$ that is parametrized by a vector r, and optimize over r the empirical approximation to the expected squared error of Eq. (3.36). Thus we can obtain r by the least squares regression:

$$\bar r \in \arg\min_r \sum_{s=1}^{q}\sum_{c=1}^{m}\big(z^s(c)-\tilde h(c,x^s,r)\big)^2, \qquad (3.37)$$

perhaps with some quadratic regularization added. The functions $\tilde h(c,x,r)$ may be provided for example by a feature-based architecture or a neural network.

Note that each training pair (x^s,c^s) is used to generate m examples for use in the regression problem (3.37): $m-1$ "negative" examples of the form $(x^s,0)$, corresponding to the $m-1$ categories $c\neq c^s$, and one "positive" example of the form $(x^s,1)$, corresponding to $c=c^s$. Note also that the incremental gradient method can be applied to the solution of this problem.

The regression problem (3.37) approximates the minimization of the expected value (3.36), so we conclude that its solution $\tilde h(c,x,\bar r)$, $c=1,\ldots,m$, approximates the probabilities $p(c\,|\,x)$. Once this solution is obtained, we may use it to classify a new object x according to the rule

$$\text{Estimated Object Category}=\tilde c(x,\bar r)\in \arg\max_{c=1,\ldots,m}\tilde h(c,x,\bar r), \qquad (3.38)$$

which approximates the MAP rule (3.34); cf. Fig. 3.4.1.

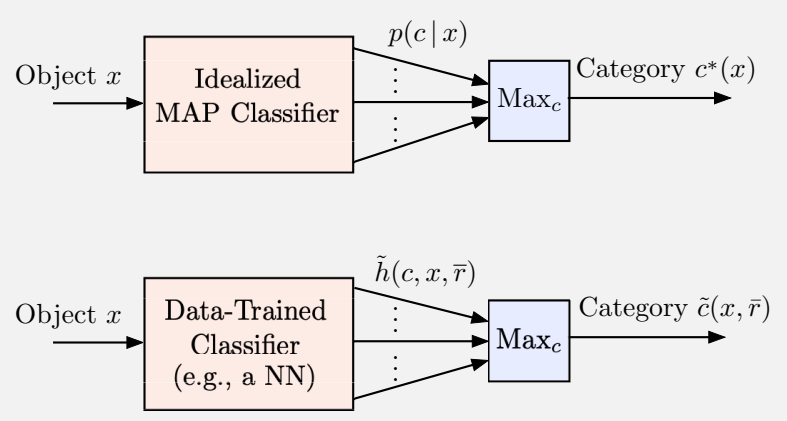


Figure 3.4.1 Illustration of the MAP classifier $c^*(x)$ for the case where the probabilities $p(c \mid x)$ are known [cf. Eq. (3.34)], and its data-trained version $\tilde{c}(x, \bar{r})$ [cf. Eq. (3.38)]. The classifier may be obtained by using the data set (x_k^s, u_k^s), $s = 1, \ldots, q$, and an approximation architecture such as a feature-based architecture or a neural network.

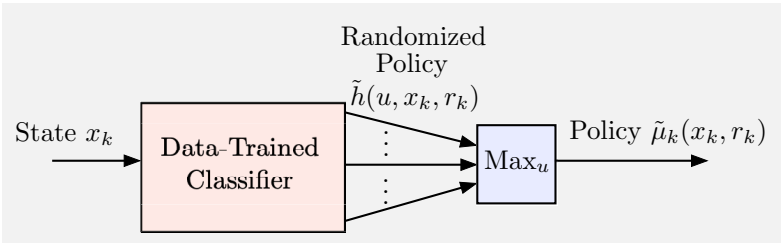


Figure 3.4.2 Illustration of classification-based approximation in policy space. The classifier, defined by the parameter r_k, is constructed by using the training set (x_k^s, u_k^s), $s = 1, \ldots, q$. It yields a randomized policy that consists of the probability $\tilde{h}(u, x_k, r_k)$ of using control $u \in U_k(x_k)$ at state x_k. This policy is approximated by the deterministic policy $\tilde{\mu}_k(x_k, r_k)$ that uses at state x_k the control that maximizes over $u \in U_k(x_k)$ the probability $\tilde{h}(u, x_k, r_k)$ [cf. Eq. (3.38)].

Returning to approximation in policy space, for a given training set (x_k^s, u_k^s), $s = 1, \ldots, q$, the classifier just described provides (approximations to) the "probabilities" of using the controls $u_k \in U_k(x_k)$ at the states x_k, so it yields a "randomized" policy $\tilde{h}(u, x_k, r_k)$ for stage k [once the values $\tilde{h}(u, x_k, r_k)$ are normalized so that, for any given x_k, they add to 1]; cf. Fig. 3.4.2. In practice, this policy is usually approximated by the deterministic policy $\tilde{\mu}_k(x_k, r_k)$ that uses at state x_k the control of maximal probability at that state; cf. Eq. (3.38).

For the simpler case of a classification problem with just two categories, say A and B, a similar formulation is to hypothesize a relation of

the following form between object x and its category:

$$\text{Object Category} = \begin{cases} A & \text{if } \tilde{h}(x,r) = 1, \\ B & \text{if } \tilde{h}(x,r) = -1, \end{cases}$$

where $\tilde{h}$ is a given function and r is the unknown parameter vector. Given a set of q object-category pairs $(x^1, z^1), \ldots, (x^q, z^q)$ where

$$z^s = \begin{cases} 1 & \text{if } x \text{ is of category } A, \\ -1 & \text{if } x \text{ is of category } B, \end{cases}$$

we obtain r by the least squares regression:

$$\bar{r} \in \arg\min_r \sum_{s=1}^{q} \big(z^s - \tilde{h}(x^s, r)\big)^2.$$

The optimal parameter vector $\bar{r}$ is used to classify a new object with data vector x according to the rule

$$\text{Estimated Object Category} = \begin{cases} A & \text{if } \tilde{h}(x,\bar{r}) > 0, \\ B & \text{if } \tilde{h}(x,\bar{r}) < 0. \end{cases}$$

In the context of DP and approximation in policy space, this classifier may be used, among others, in stopping problems where there are just two controls available at each state: stopping (i.e., moving to a termination state) and continuing (i.e., moving to some nontermination state).

There are several variations of the preceding classification schemes, for which we refer to the specialized literature. Moreover, there are several commercially and publicly available software packages for solving the associated regression problems and their variants. They can be brought to bear on the problem of parametric approximation in policy space using any training set of state-control pairs, regardless of how it was obtained.

3.4.2 Policy Iteration with Value and Policy Networks - Multiprocessor Parallelization

As we have already noted, in contrast to rollout, approximate policy iteration (PI) is fundamentally an off-line training algorithm, because for a large scale problem, it is necessary to represent the successively generated policies with an approximation architecture. Thus approximate PI involves the successive use of a value network to implement policy evaluation, and a policy network to represent policy improvement.

In particular, we can start with a base policy and a terminal cost approximation, and generate state-control samples of the corresponding truncated rollout policy. These samples can in turn be used with the

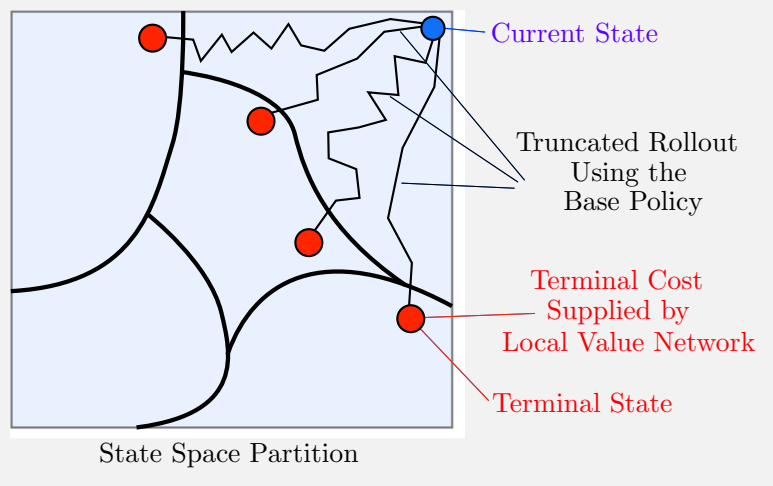


Figure 3.4.3 Illustration of a truncated rollout scheme with a partitioned architecture. A local value network is used for terminal cost function approximation.

approximation in policy space scheme of this section to obtain a policy network that approximates the truncated rollout policy; cf. Fig. 3.4.3.

The cost function of the policy network can in turn be approximated with a value network using the methodology that we have discussed in this chapter. The value network can be used in turn as a terminal cost function approximation in a truncated rollout scheme where the previously obtained policy network can be used as a base policy. In this way a perpetual rollout scheme is obtained, which involves a sequence of value and policy networks.

Parallelization and distributed computation can be used in several different ways in such a scheme, including Q-factor, Monte Carlo, and multiagent parallelization. Moreover, when feature-based partitioning of the state space is used (cf. Example 3.1.8), we may consider a multiprocessor parallelization scheme, which involves multiple local value and policy networks, one per subset of the state space partition; see Fig. 3.4.4.

Let us finally note that multiprocessor parallelization leads to the idea of an approximation architecture that involves a graph. Each node of the graph consists of a neural network and each arc connecting a pair of nodes corresponds to data transfer between the corresponding neural networks. The question of how to train such an architecture is quite complex and one may think of several alternative possibilities. For example the training may be collaborative with the exchange of training results and/or training data communicated periodically or asynchronously; see the book [Ber20a], Section 5.8.

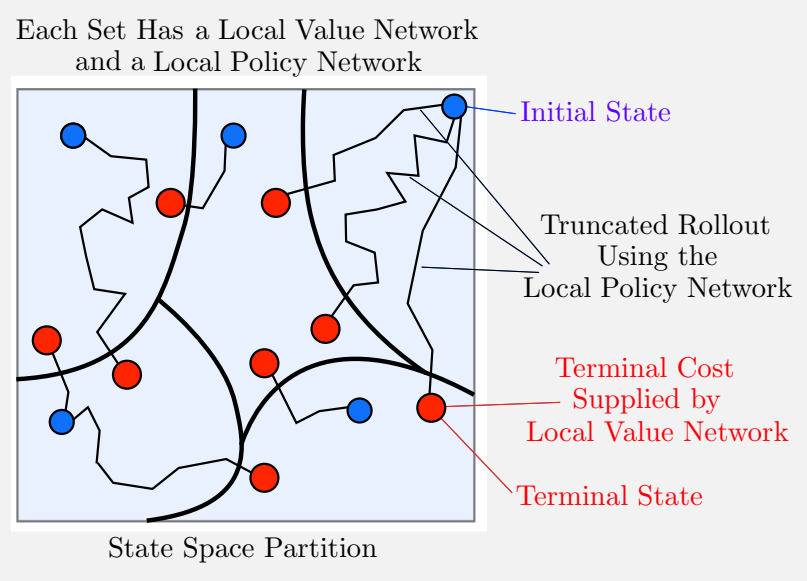


Figure 3.4.4 Illustration of a perpetual truncated rollout scheme with a partitioned architecture. A local value network and a local policy network are used for each subset of the partition. The policy network is used as the base policy and the value network is used to provide a terminal cost function approximation.

State-control training pairs for the corresponding rollout policy are obtained by starting at an initial state within some subset of the partition, generating rollout trajectories using the local policy network, which are truncated once the state enters a different subset of the partition, with the corresponding terminal cost function approximation supplied by the value network of that subset.

When a separate processor is used for each subset of partition, the corresponding value networks are communicated between processors. This can be done asynchronously, with each processor sharing its value network as soon it becomes available. In a variation of this scheme, the local policy networks may also be shared selectively among processors for selective use in the truncated rollout process.

3.4.3 Why Use On-Line Play and not Just Train a Policy Network to Emulate the Lookahead Minimization?

This is a sensible and common question, which stems from the mindset that neural networks have extraordinary function approximation properties. In other words, why go through the arduous on-line process of lookahead minimization, if we can do the same thing off-line and represent the lookahead policy with a trained policy network? In particular, we can select the policy from a suitably restricted class of policies, such as a parametric class of the form $\mu(x, r)$, where r is a parameter vector. We may then estimate r using some type of off-line training process. Then the on-line computation of controls $\mu(x, r)$ can be much faster compared with on-line lookahead minimization.

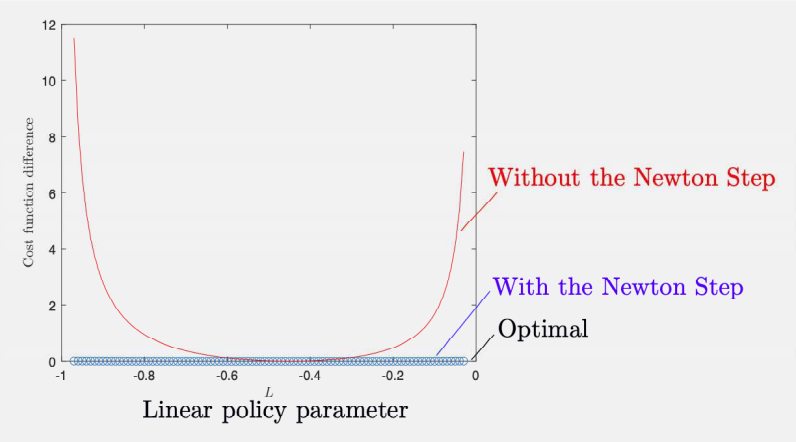


Figure 3.4.5 Illustration of the performance enhancement obtained by rollout with an off-line trained base policy for the linear quadratic problem.

On the negative side, because parametrized approximations often involve substantial calculations, they are not well suited for on-line replanning. From our point of view in these notes, there is another important reason why approximation in value space is needed on top of approximation in policy space: *the off-line trained policy may not perform nearly as well as the corresponding one-step or multistep lookahead/rollout policy*, because it lacks the extra power of the associated exact Newton step (cf. our discussion of AlphaZero and TD-Gammon in Section 1.1, and linear quadratic problems in Section 1.5).

Figure 3.4.5 illustrates this fact with a one-dimensional linear-quadratic example, and compares the performance of a linear policy with its corresponding one-step lookahead policy. In this example the system equation is

$$x_{k+1} = x_k + 2u_k,$$

and the quadratic cost function parameters are $q = 1$, $r = 0.5$. The optimal policy for this system and cost parameter values is

$$\mu^*(x) = L^*x,$$

with $L^* \approx -0.4$, and the optimal cost function is

$$J^*(x) = K^*x^2,$$

where $K^* \approx 1.1$. We want to to explore what happens when we use a policy of the form

$$\mu_L(x) = Lx,$$

where $L \neq L^*$ (which may be optimal for another system equation or cost function parameters). The cost function of μ_L has the form

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

where K_L is obtained by using the formulas given in Section 1.5. The figure shows the quadratic cost coefficient differences $K_L - K^*$ and $K_{\tilde{L}} - K^*$ as a function of L, where K_L and $K_{\tilde{L}}$ are the quadratic cost coefficients of μ (without one-step lookahead/Newton step) and the corresponding one-step lookahead policy $\tilde{\mu}$ (with one-step lookahead/Newton step).

3.5 AGGREGATION

In this section we consider approximation in value space using a problem approximation approach that is based on aggregation. In particular, we construct a simpler and more tractable "aggregate" problem by creating special subsets of states, which we view as "aggregate states." We then solve the aggregate problem exactly by DP. This is the off-line training part of the aggregation approach, and it may be carried out with a variety of DP methods, including simulation-based value and policy iteration; we refer to the RL book [Ber19a] for a detailed account. Finally, we use the optimal cost-to-go function of the aggregate problem to construct a terminal cost approximation in a one-step or multistep lookahead approximation scheme for the original problem. Additionally, we may also use the optimal policy of the aggregate problem to construct a base policy for a truncated rollout scheme.

In addition to problem approximation, aggregation is related to feature-based parametric approximation. In particular, it often produces a piecewise constant cost function approximation, which may be viewed as a linear feature-based parametrization, where the features are 0-1 membership functions; see Example 3.1.1. Aggregation can also be combined with other approximation schemes, to add a local correction to a cost function approximation $\tilde{J}$, which is already available, possibly through the use of a neural network; see the discussion of biased aggregation later in Section 3.5.7.

Aggregation can be applied to both finite horizon and infinite horizon problems. In this section, we will focus primarily on the discounted infinite horizon problem. We will introduce aggregation in a simple intuitive form in Section 3.5.1, and generalize later to a more sophisticated form of feature-based aggregation, which we also discussed briefly in Example 3.1.7.

3.5.1 Aggregation with Representative States

In this section we focus on a relatively simple form of aggregation, which involves a special subset of states, called *representative*. Our approach is to

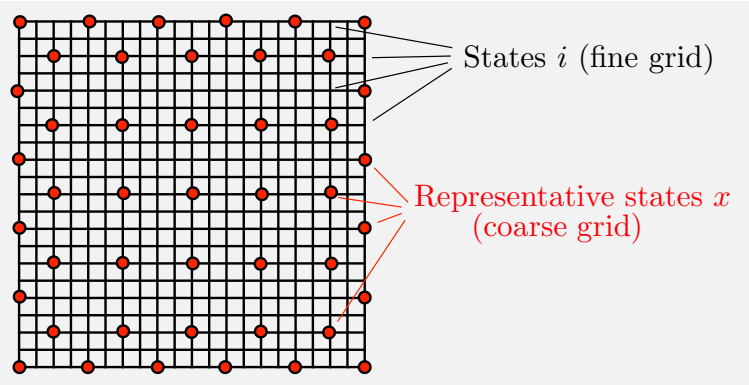


Figure 3.5.1 Illustration of aggregation with representative states; cf. Example 3.5.1. A relatively small number of states are viewed as representative. We define transition probabilities between pairs of aggregate states and we also define the associated expected transition costs. These specify a smaller DP problem, called the aggregate problem, which is solved exactly. The optimal cost function J^* of the original problem is approximated by interpolation from the optimal costs of the representative states r_y^* in the aggregate problem:

$$\tilde{J}(j) = \sum_{y \in \mathcal{A}} \phi_{jy} r_y^*, \qquad j = 1, \ldots, n,$$

and is used in a one-step or multistep lookahead scheme.

view these states as the states of a smaller optimal control problem, the aggregate problem, which we will formulate and solve exactly in place of the original. We will then use the optimal aggregate costs of the representative states to approximate the optimal costs of the original problem states by interpolation. In this chapter, whenever we consider a finite-state problem, we use notation that is more convenient for such a problem. In particular, states and successor states will be denoted by i and j, respectively, and the system equation is represented by control-dependent transition probabilities $p_{ij}(u)$; cf. Section 1.4.1. Let us describe a classical example.

Example 3.5.1 (Coarse Grid Approximation)

Consider a discounted problem where the state space is a grid of points $i = 1, \ldots, n$ on the plane. We introduce a coarser grid that consists of a subset $\mathcal{A}$ of the states/points, which we call representative and denote by x; see Fig. 3.5.1. We now wish to formulate a lower-dimensional DP problem just on the coarse grid of states. The difficulty here is that there may be positive transition probabilities $p_{xj}(u)$ from some representative states x to some non-representative states j. To deal with this difficulty, we introduce artificial transition probabilities ϕ_{jy} from non-representative states j to representative states y, which we call *aggregation probabilities*. In particular, a transition

from representative state x to a nonrepresentative state j, is followed by a transition from j to some other representative state y with probability ϕ_{jy}; see Fig. 3.5.2.

This process involves approximation but constructs a transition mechanism for an *aggregate problem* whose states are just the representative ones. The transition probabilities between representative states x, y under control $u \in U(x)$ and the corresponding expected transition costs are

$$\hat{p}_{xy}(u) = \sum_{j=1}^{n} p_{xj}(u)\phi_{jy}, \qquad \hat{g}(x,u) = \sum_{j=1}^{n} p_{xj}(u)g(x,u,j). \tag{3.39}$$

We can solve the aggregate problem by any suitable exact DP method. Let $\mathcal{A}$ denote the set of representative states and let r_x^* denote the corresponding optimal cost of representative state x. We can then approximate the optimal cost function of the original problem with the interpolation formula

$$\tilde{J}(j) = \sum_{y \in \mathcal{A}} \phi_{jy} r_y^*, \qquad j = 1, \ldots, n. \tag{3.40}$$

This function may in turn be used in a one-step or multistep lookahead scheme for approximation in value space of the original problem.

Note that there is a lot of freedom in selecting the aggregation probabilities ϕ_{jy}. Intuitively, ϕ_{jy} should express a measure of proximity between j and y, e.g., ϕ_{jy} should be relatively large when y is geometrically close to j. For example, we could set $\phi_{jy_j} = 1$ for the representative state y_j that is "closest" to j, and $\phi_{jy_j} = 0$ for all other representative states $y \neq y_j$. In this case, Eq. (3.40) yields a piecewise constant cost function approximation $\tilde{J}$ (the constant values are the scalars r_y^* of the representative states y).

We will now formalize our framework for aggregation with representative states by generalizing the preceding example; see Fig. 3.5.3. We first consider the n-state version of the α-discounted problem of Section 1.4.1. We refer to this problem as the "original problem," to distinguish from the "aggregate problem," which we define next.

Aggregation Framework with Representative States

We introduce a finite subset $\mathcal{A}$ of the original system states, which we call *representative states*, and we denote them by symbols such as x and y. We construct an *aggregate problem*, with state space $\mathcal{A}$, and transition probabilities and transition costs defined as follows:

(a) We relate the original system states j to representative states $y \in \mathcal{A}$ with aggregation probabilities ϕ_{jy}; these are scalar "weights" satisfying $\phi_{jy} \geq 0$ for all $y \in \mathcal{A}$, and $\sum_{y \in \mathcal{A}} \phi_{jy} = 1$.

(b) We define the transition probabilities between representative states x and y under control $u \in U(x)$ by

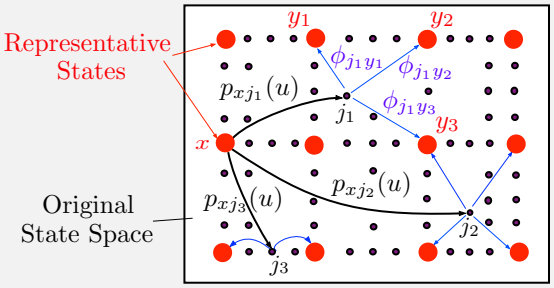


Figure 3.5.2 Illustration of the use of aggregation probabilities ϕ_{jy} from nonrepresentative states j to representative states y in Example 3.5.1. A transition from a state x to a nonrepresentative state j is followed by a transition to aggregate state y with probability ϕ_{jy}. In this figure, from representative state x, there are three possible transitions, to states j_1, j_2, and j_3, according to $p_{xj_1}(u), p_{xj_2}(u), p_{xj_3}(u)$, and each of these states is associated with a convex combination of representative states using the aggregation probabilities. For example, the state j_1 is associated with

$$\phi_{j_1 y_1} y_1 + \phi_{j_1 y_2} y_2 + \phi_{j_1 y_3} y_3.$$

$$\hat{p}_{xy}(u) = \sum_{j=1}^{n} p_{xj}(u)\phi_{jy}. \tag{3.41}$$

(c) We define the expected transition costs at representative states x under control $u \in U(x)$ by

$$\hat{g}(x,u) = \sum_{j=1}^{n} p_{xj}(u) g(x,u,j). \tag{3.42}$$

The optimal costs of the representative states $y \in \mathcal{A}$ in the aggregate problem are denoted by r_y^*, and they define approximate costs for the original problem through the interpolation formula

$$\tilde{J}(j) = \sum_{y \in \mathcal{A}} \phi_{jy} r_y^*, \qquad j = 1, \ldots, n. \tag{3.43}$$

Aside from the selection of representative states, an important consideration is the choice of the aggregation probabilities. These probabilities express "similarity" or "proximity" of original to representative states (as in the case of the coarse grid Example 3.5.1), but in principle they can be

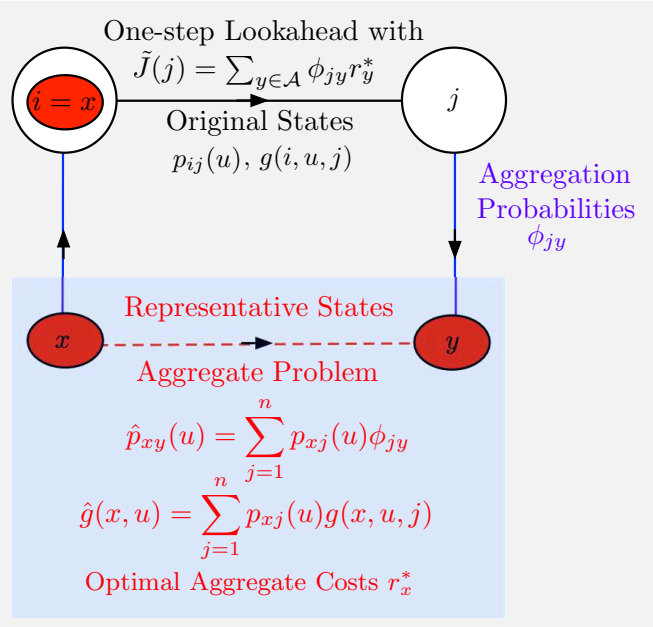


Figure 3.5.3 Illustration of the aggregate problem in the representative states framework. The transition probabilities $\hat{p}_{xy}(u)$ and transition costs $\hat{g}(x,u)$ are shown in the bottom part of the figure. Once the aggregate problem is solved (exactly) for its optimal costs r_y^*, we define approximate costs

$$\tilde{J}(j) = \sum_{y \in \mathcal{A}} \phi_{jy} r_y^*, \qquad j = 1, \ldots, n,$$

which are used for one-step lookahead approximation of the original problem.

arbitrary (as long as they are nonnegative and sum to 1 over y). Intuitively, ϕ_{jy} may be interpreted as some measure of "strength of relation" of j to y. The vectors $\{\phi_{jy} \mid j = 1, \ldots, n\}$ may also be viewed as basis functions for a linear cost function approximation via Eq. (3.43).

Hard Aggregation and Error Bound

A special case of interest, called *hard aggregation*, is when for every state j, we have $\phi_{jy} = 0$ for all representative states y, except a single one, denoted y_j, for which we have $\phi_{jy_j} = 1$. In this case, the one-step lookahead approximation

$$\tilde{J}(j) = \sum_{y \in \mathcal{A}} \phi_{jy} r_y^*, \qquad j = 1, \ldots, n,$$

is *piecewise constant*; it is constant and equal to r_y^* for all j in the set

$$S_y = \{j \mid \phi_{jy} = 1\}, \qquad y \in \mathcal{A},$$

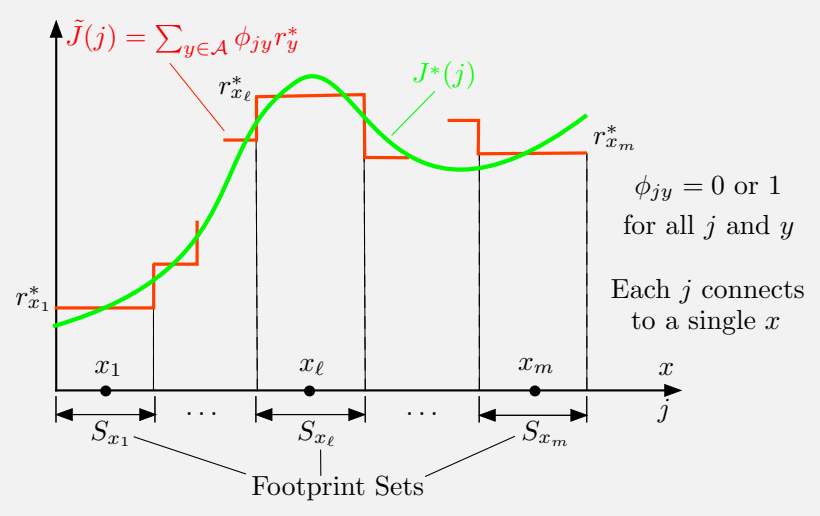


Figure 3.5.4 Illustration of the piecewise constant cost approximation

$$\tilde{J}(j) = \sum_{y \in \mathcal{A}} \phi_{jy} r^*_y, \qquad j = 1, \ldots, n,$$

in the hard aggregation case where we have $\phi_{jy} = 0$ for all representative states y, except a single one. Here $\tilde{J}$ is constant and equal to r^*_y for all j in the footprint set

$$S_y = \{j \mid \phi_{jy} = 1\}, \qquad y \in \mathcal{A}.$$

called the *footprint* of representative state y; see Fig. 3.5.4. Moreover the footprints of all the representative states are disjoint and form a partition of the state space, i.e.,

$$\cup_{x \in \mathcal{A}} S_x = \{1, \ldots, n\}.$$

The footprint sets can be used to define a bound for the error $(J^* - \tilde{J})$. In particular, it can be shown that

$$\left|J^*(j) - \tilde{J}(j)\right| \leq \frac{\epsilon}{1-\alpha}, \qquad j = 1, \ldots, n,$$

where

$$\epsilon = \max_{y \in \mathcal{A}} \max_{i,j \in S_y} \left|J^*(i) - J^*(j)\right|$$

is the *maximum variation of J^* within the footprint sets S_y*. This error bound result can be extended to the more general aggregation framework that will be given in the next section. Note the primary intuition derived

from this bound: *the error due to hard aggregation is small if J^* varies little within each S_y.*

For a special hard aggregation case of interest, consider the geometrical context of Example 3.5.1. There, aggregation probabilities are often based on a nearest neighbor approximation scheme, whereby each nonrepresentative state j takes the cost value of the "closest" representative state y, i.e.,

$$\phi_{jy_j} = 1 \qquad \text{if } y_j \text{ is the closest representative state to } j.$$

Then all states j for which a given representative state y is the closest to j (the footprint of y) are assigned equal approximate cost $\tilde{J}(j) = r^*_y$.

Methods for Solving the Aggregate Problem

The most straightforward way to solve the aggregate problem is to compute the aggregate problem transition probabilities $\hat{p}_{xy}(u)$ [cf. Eq. (3.41)] and transition costs $\hat{g}(x,u)$ [cf. Eq. (3.42)] by either an algebraic calculation or by simulation. The aggregate problem may then be solved by any one of the standard methods, such as VI or PI. This exact calculation is plausible if the number of representative states is relatively small. An alternative possibility is to use a simulation-based VI or PI method. We refer to a discussion of these methods in the author's books [Ber12], Section 6.5, and [Ber19a], Section 6.3. The idea is that a simulator for the original problem can be used to construct a simulator for the aggregate problem; cf. Fig. 3.5.3.

An important observation is that if the original problem is deterministic and hard aggregation is used, the aggregate problem is also deterministic, and can be solved by shortest-path like methods. This is true for both discounted problems and for undiscounted shortest path-type problems. In the latter case, the termination state of the original problem must be included as a representative state in the aggregate problem. However, if hard aggregation is not used, the aggregate problem will be stochastic, because of the introduction of the aggregation probabilities. Of course, once the aggregate problem is solved and the lookahead approximation $\tilde{J}$ is obtained, a deterministic structure in the original problem can be exploited to facilitate the lookahead minimizations.

3.5.2 Continuous Control Space Discretization

Aggregation with representative states extends without difficulty to problems with a continuous state space, as long as the control space is finite. Then once the representative states and the aggregation probabilities have been defined, the corresponding aggregate problem is a discounted problem with finite state and control spaces, which can be solved with the standard

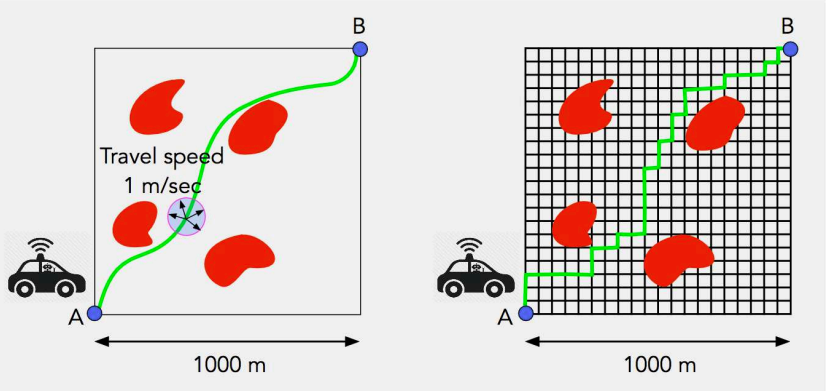


Figure 3.5.5 Illustration of discretization issues for problems with infinite state and control spaces.

methods. The only potential difficulty arises when the disturbance space is also infinite, in which case the calculation of the transition probabilities and expected stage costs of the aggregate problem must be obtained by some form of integration process.

The case where both the state and the control spaces are continuous is somewhat more complicated, because both of these spaces must be discretized using representative state-control pairs, instead of just representative states. The following example illustrates what may happen if we use representative state discretization only.

Example 3.5.2 (Continuous Shortest Path Discretization)

Suppose that we want to find the fastest route for a car to travel between two points A and B located at the opposite ends of a square with side 1000 meters, while avoiding some known obstacles. We assume a constant car speed of 1 meter per second and that the car can drive in any direction; cf. Fig. 3.5.5.

Let us consider discretizing the space with a square grid (a set of representative states), and restrict the directions of motion to horizontal and vertical, so that at each stage the car moves from a grid point to one of the four closest grid points. Thus in the discretized version of the problem the car travels with a sequence of horizontal and vertical moves as indicated in the right side of Fig. 3.5.5. Is it possible to approximate the fastest route arbitrarily closely with the optimal solution of the discretized problem, assuming a sufficiently fine grid?

The answer is no! To see this note that in the discretized problem the optimal travel time is 2000 secs, regardless of how fine the discretization is. On the other hand, in the continuous space/nondiscretized problem the optimal travel time can be as little as $\sqrt{2} \cdot 1000$ secs (this corresponds to the favorable case where the straight line from A to B does not meet an obstacle).

The difficulty in the preceding example is that *the state space is discretized finely but the control space is not*. What is needed is to introduce a fine discretization of the control space as well, through some set of "representative controls." We can deal with this situation with a suitable form of discretized aggregate problem, which when solved provides an appropriate form of cost function approximation for use with one-step lookahead. The discretized problem is a stochastic infinite horizon problem, even if the original problem is deterministic. Further discussion of this approach is outside our scope, and we refer to the sources cited at the end of the chapter. Under reasonable assumptions it is possible to show consistency, i.e., that the optimal cost function of the discretized problem converges to the optimal cost function of the original continuous spaces problem as the discretization of both the state and the control spaces becomes increasingly fine.

The type of difficulty illustrated in Example 3.5.2 does not arise if the state space is continuous but the control space is finite. In particular, this is true in partially observed finite spaces Markov decision problems (POMDP), which are defined over their belief space (the space of probability distributions over their states). We briefly discuss this case next.

3.5.3 Continuous State Space - POMDP Discretization

Let us consider any α-discounted DP problem, where the state space is a bounded convex subset B of a Euclidean space, such as the unit simplex, but the control space U is finite. We use b to denote the states, to emphasize the connection with belief states in POMDP and to distinguish them from x, which we will use to denote representative states. Bellman's equation is $J = TJ$ with the Bellman operator T defined by

$$(TJ)(b) = \min_{u \in U} \mathop{E}_{w} \big\{ g(b,u,w) + \alpha J\big(f(b,u,w)\big) \big\}, \qquad b \in B.$$

We introduce a set of representative states $\{x_1, \ldots, x_m\} \subset B$. We assume that the convex hull of $\{x_1, \ldots, x_m\}$ is equal to B, so each state $b \in B$ can be expressed as

$$b = \sum_{i=1}^{m} \phi_{bx_i} x_i,$$

where $\{\phi_{bx_i} \mid i = 1, \ldots, m\}$ is a probability distribution:

$$\phi_{bx_i} \geq 0, \ i = 1, \ldots, m, \quad \sum_{i=1}^{m} \phi_{bx_i} = 1, \qquad \text{for all } b \in B.$$

We view ϕ_{bx_i} as aggregation probabilities.

Consider the operator $\hat{T}$ that transforms a vector $r = (r_{x_1}, \ldots, r_{x_m})$ into the vector $\hat{T}r$ with components $(\hat{T}r)(x_1), \ldots, (\hat{T}r)(x_m)$ defined by

$$(\hat{T}r)(x_i) = \min_{u \in U} \mathop{E}_{w} \left\{ g(x_i, u, w) + \alpha \sum_{j=1}^{m} \phi_{f(x_i,u,w)\, x_j} r_{x_j} \right\}, \qquad i = 1, \ldots, m,$$

where $\phi_{f(x_i,u,w)\, x_j}$ are the aggregation probabilities of the state $f(x_i, u, w)$. It can then be shown that $\hat{T}$ is a contraction mapping with respect to the maximum norm (we give the proof for a similar result in the next section). Bellman's equation for an aggregate finite-state discounted DP problem whose states are $x_1, \ldots, x_m$ has the form

$$r_{x_i} = (\hat{T}r)(x_i), \qquad i = 1, \ldots, m,$$

and has a unique solution.

The transitions in this problem occur as follows: from state x_i under control u, we first move to $f(x_i, u, w)$ at cost $g(x_i, u, w)$, and then we move to a state x_j, $j = 1, \ldots, m$, according to the probabilities $\phi_{f(x_i,u,w)\, x_j}$. The optimal costs $r^*_{x_i}$, $i = 1, \ldots, m$, of this problem can often be obtained by standard VI and PI methods that may or may not use simulation. We may then approximate the optimal cost function of the original problem by

$$\tilde{J}(b) = \sum_{i=1}^{m} \phi_{bx_i} r^*_{x_i}, \qquad \text{for all } b \in B,$$

and reasonably expect that the optimal discretized solution converges to the optimal as the number of representative states increases.

In the case where B is the belief space of an α-discounted POMDP, the representative states/beliefs and the aggregation probabilities define an aggregate problem, which is a finite-state α-discounted problem with a perfect state information structure. This problem can be solved with exact DP methods if either the aggregate transition probabilities and transition costs can be obtained analytically (in favorable cases) or if the number of representative states is small enough to allow their calculation by simulation. The aggregate problem can also be addressed with approximate DP method that we have discussed earlier, such as problem approximation/certainty equivalence approaches. It can also be addressed with a rollout method, which is suitable for an on-line implementation.

3.5.4 General Aggregation

We will now discuss a more general aggregation framework for the infinite horizon n-state α-discounted problem, by using subsets of states as aggregate states. In particular, we define our general aggregation framework by

essentially replacing the representative states x with *subsets* $I_x \subset \{1, \ldots, n\}$ of the original state space as follows. These subsets are often constructed by using features, however, it is helpful to formulate our aggregation framework in a general form, and introduce features later.

General Aggregation Framework

We introduce a finite subset $\mathcal{A}$ of aggregate states, which we denote by symbols such as x and y. We define:

(a) A collection of disjoint subsets $I_x \subset \{1, \ldots, n\}$, $x \in \mathcal{A}$.

(b) A probability distribution over $\{1, \ldots, n\}$ for each $x \in \mathcal{A}$, denoted by $\{d_{xi} \mid i = 1, \ldots, n\}$, and referred to the *disaggregation probabilities of* x. We require that the distribution corresponding to x is concentrated on the subset I_x:

$$d_{xi} = 0, \qquad \text{for all } i \notin I_x,\ x \in \mathcal{A}. \tag{3.44}$$

(c) For each original system state $j \in \{1, \ldots, n\}$, a probability distribution over $\mathcal{A}$, denoted by $\{\phi_{jy} \mid y \in \mathcal{A}\}$, and referred to as the *aggregation probabilities of* j. We require that

$$\phi_{jy} = 1, \qquad \text{for all } j \in I_y,\ y \in \mathcal{A}. \tag{3.45}$$

The aggregation and disaggregation probabilities specify a dynamic system involving both aggregate and original system states; cf. Fig. 3.5.6. In this system:

(i) From aggregate state x, we generate an original system state $i \in I_x$ according to d_{xi}.

(ii) We generate transitions between original system states i and j according to $p_{ij}(u)$, with cost $g(i, u, j)$.

(iii) From original system state j, we generate aggregate state y according to ϕ_{jy}. [Note that in view of Eq. (3.45), all states j within a set I_y, are aggregated onto y with $\phi_{jy} = 1$.]

The optimal costs of the aggregate states $y \in \mathcal{A}$ in the aggregate problem are denoted by r^*_y, and they define approximate costs for the original problem through the interpolation formula

$$\tilde{J}(j) = \sum_{y \in \mathcal{A}} \phi_{jy} r^*_y, \qquad j = 1, \ldots, n. \tag{3.46}$$

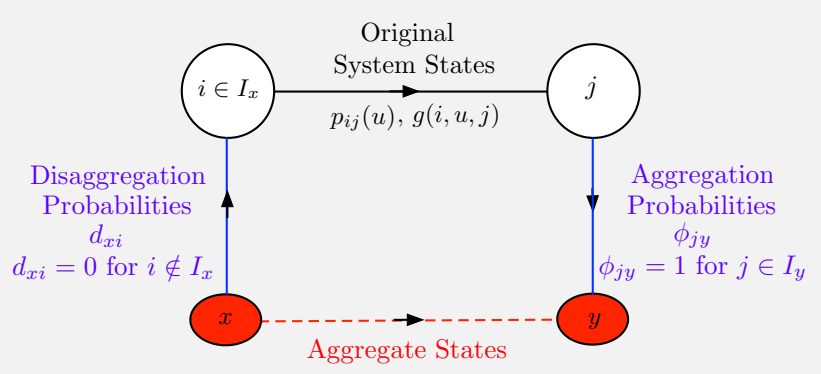


Figure 3.5.6 Illustration of the aggregate system, and the transition mechanism and the costs per stage of the aggregate problem.

Our general aggregation framework is illustrated in Fig. 3.5.6. Note that if each set I_x consists of a single state, we obtain the representative states framework of the preceding section. In this case the disaggregation distribution $\{d_{xi} \mid i \in I_x\}$ is just the atomic distribution that assigns probability 1 to the unique state in I_x. Consistent with the special case of representative states, the disaggregation probability d_{xi} may be interpreted as a "measure of the relation of x and i."

The aggregate problem is a DP problem with an enlarged state space that consists of two copies of the original state space $\{1, \ldots, n\}$ plus the set of aggregate states $\mathcal{A}$. We introduce the corresponding optimal vectors $\tilde{J}_0$, $\tilde{J}_1$, and $r^* = \{r_x^* \mid x \in \mathcal{A}\}$ where:

r_x^* is the optimal cost-to-go from aggregate state x.

$\tilde{J}_0(i)$ is the optimal cost-to-go from original system state i that has just been generated from an aggregate state (left side of Fig. 3.5.6).

$\tilde{J}_1(j)$ is the optimal cost-to-go from original system state j that has just been generated from an original system state (right side of Fig. 3.5.6).

Note that because of the intermediate transitions to aggregate states, $\tilde{J}_0$ and $\tilde{J}_1$ are different.

These three vectors satisfy the following three Bellman equations:

$$r_x^* = \sum_{i \in I_x} d_{xi} \tilde{J}_0(i), \qquad x \in \mathcal{A}, \tag{3.47}$$

$$\tilde{J}_0(i) = \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha \tilde{J}_1(j)\big), \qquad i = 1, \ldots, n, \tag{3.48}$$

$$\tilde{J}_1(j) = \sum_{y \in \mathcal{A}} \phi_{jy} r_y^*, \qquad j = 1, \ldots, n. \tag{3.49}$$

The objective is to solve for the optimal costs r_x^* of the aggregate states in order to obtain approximate costs for the original problem through the interpolation formula

$$\tilde{J}(j) = \sum_{y \in \mathcal{A}} \phi_{jy} r_y^*, \qquad j = 1, \ldots, n;$$

cf. Eq. (3.46).

By combining the three Bellman equations (3.47)-(3.49), we see that r^* satisfies

$$r_x^* = \sum_{i \in I_x} d_{xi} \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u) \left(g(i,u,j) + \alpha \sum_{y \in \mathcal{A}} \phi_{jy}\, r_y^* \right), \qquad x \in \mathcal{A}, \tag{3.50}$$

or equivalently $r^* = Hr^*$, where H is the operator that maps the vector r to the vector Hr with components

$$(Hr)(x) = \sum_{i \in I_x} d_{xi} \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u) \left(g(i,u,j) + \alpha \sum_{y \in \mathcal{A}} \phi_{jy}\, r_y \right), \quad x \in \mathcal{A}. \tag{3.51}$$

It can be shown that H *is a contraction mapping with respect to the maximum norm*, and thus the composite Bellman equation (3.50) has r^* as its unique solution. To see this, we note for any vectors r and r', we have

$$\begin{aligned}
(Hr)(x) &= \sum_{i \in I_x} d_{xi} \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u) \left(g(i,u,j) + \alpha \sum_{y \in \mathcal{A}} \phi_{jy}\, r_y \right) \\
&\leq \sum_{i \in I_x} d_{xi} \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u) \Big(g(i,u,j) + \alpha \sum_{y \in \mathcal{A}} \phi_{jy} \left(r'_y + \|r - r'\| \right) \Big) \\
&= (Hr')(x) + \alpha \|r - r'\|,
\end{aligned}$$

where $\|\cdot\|$ is the maximum norm, and the equality follows from the definition of $(Hr')(x)$, and the fact that d_{xi}, $p_{ij}(u)$, and ϕ_{jy} are probabilities. Thus we have

$$(Hr)(x) - (Hr')(x) \leq \alpha \|r - r'\|, \qquad x \in \mathcal{A}.$$

By reversing the roles of r and r', we also have

$$(Hr')(x) - (Hr)(x) \leq \alpha \|r - r'\|, \qquad x \in \mathcal{A},$$

so that

$$\big|(Hr')(x) - (Hr)(x)\big| \leq \alpha \|r - r'\|, \qquad x \in \mathcal{A}.$$

By taking the maximum over $x \in \mathcal{A}$, it follows that

$$\|Hr - Hr'\| \leq \alpha \|r - r'\|,$$

and that H is a maximum norm contraction.

Note that the composite Bellman equation (3.50) has dimension equal to the number of aggregate states, which is potentially much smaller than n. To apply the aggregation framework of this section, we may solve exactly this equation for the optimal aggregate costs r^*_x, $x \in \mathcal{A}$, by simulation-based analogs of the VI and PI methods, and obtain a cost function approximation for the original problem through the interpolation formula (3.46). We will develop these methods later, but before doing so, we discuss various ways to formulate the aggregation framework, and in particular, how features can be used for this purpose.

3.5.5 Hard Aggregation and Error Bounds

Let us consider the special case of *hard aggregation*, where for every state j, we have $\phi_{jy} = 0$ for all aggregate states y, except a single one, denoted y_j, for which we have $\phi_{jy_j} = 1$. In this case, the one-step lookahead approximation

$$\tilde{J}(j) = \sum_{y \in \mathcal{A}} \phi_{jy} r^*_y, \qquad j = 1, \ldots, n,$$

is piecewise constant; it is constant and equal to r^*_y for all j in the set

$$S_y = \{j \mid \phi_{jy} = 1\}, \qquad y \in \mathcal{A}, \tag{3.52}$$

called the *footprint* of aggregate state y; see Fig. 3.5.4. Note that the footprints of all the aggregate states are disjoint and form a partition of the state space, i.e.,

$$\cup_{x \in \mathcal{A}} S_x = \{1, \ldots, n\}.$$

We can show the following error bound, due to Tsitsiklis and Van Roy [TsV96]; a generalization of this error bound will be given later in this section.

Proposition 3.5.1: (Error Bound for Hard Aggregation) In the case of hard aggregation, we have

$$\big|J^*(j) - \tilde{J}(j)\big| \leq \frac{\epsilon}{1-\alpha}, \qquad \text{for all } j \text{ such that } j \in S_y,\ y \in \mathcal{A},$$

where ϵ is the maximum variation of the optimal cost function J^* over the footprint sets S_y, $y \in \mathcal{A}$:

$$\epsilon = \max_{y \in \mathcal{A}} \max_{i,j \in S_y} \big|J^*(i) - J^*(j)\big|.$$

The meaning of the preceding proposition is that if the optimal cost function J^* varies by at most ϵ within each set S_y, the hard aggregation scheme yields a piecewise constant approximation to the optimal cost function that is within $\epsilon/(1-\alpha)$ of the optimal.

Aside from its intuitive nature and error bound properties, hard aggregation provides a connection with another major approch for approximation in value space, the so called the *projected equation approach*, which we have not discussed here; see the books [Ber12] and [Ber19a]. In particular, it can be shown that for a given policy, the corresponding composite Bellman equation (3.50) for approximate evaluation of μ can be viewed as a projected equation, where a projection seminorm is used that is defined by the disaggregation probabilities; see the paper by Yu and Bertsekas [YuB12] (Section 5.5), or the book [Ber12] (Exercise 6.10).

Selecting the Aggregate States

Generally, the method to select the aggregate states is an important issue, for which there is no mathematical theory at present. However, in practical problems, based on intuition and problem-specific knowledge, there are usually evident choices, which may be fine-tuned by experimentation. For example, suppose that the optimal cost function J^* is piecewise constant over a partition $\{S_y \mid y \in \mathcal{A}\}$ of the state space $\{1, \ldots, n\}$. By this we mean that for some vector

$$r^* = \big\{r_y^* \mid y \in \mathcal{A}\big\},$$

we have

$$J^*(j) = r_y^* \qquad \text{for all } j \in S_y,\ y \in \mathcal{A}.$$

Then from Prop. 3.5.1 it follows that the hard aggregation scheme with $I_x = S_x$ for all $x \in \mathcal{A}$ is exact, so r_x^* are the optimal costs of the aggregate states x in the aggregate problem. This suggests that *in hard aggregation,*

the states in the footprint set S_y corresponding to an aggregate state y should have roughly equal optimal cost, consistently with the error bound of Prop. 3.5.1.

As an extension of the preceding argument, suppose that through some special insight into the problem's structure or some preliminary calculation, we know some features of the system's state that can "predict well" its optimal cost when combined through some approximation architecture, e.g., one that is linear. Then it seems reasonable to form the set aggregate states $\mathcal{A}$ of a hard aggregation scheme so that the sets I_y and S_y consist of states with "similar features" for every $y \in \mathcal{A}$. This is called *feature-based aggregation*, and was suggested in the neuro-dynamic book [BeT96], Section 3.1.2. The next section considers this possibility, and provides a way to introduce features and nonlinearities into the aggregation architecture, without compromising its other favorable aspects.

3.5.6 Aggregation Using Features

Let us consider the guideline for hard aggregation that we just discussed: *states i that belong to the same footprint set S_y should have nearly equal optimal costs*, i.e.,

$$\max_{i,j \in S_y} \left| J^*(i) - J^*(j) \right| \approx 0, \qquad \text{for all } y \in \mathcal{A}.$$

The question now is how to select the sets S_y according to this guideline.

An idea that comes to mind is to use a *feature mapping*, i.e., a function F that maps a state i into an m-dimensional feature vector $F(i)$; cf. Example 3.1.7. In particular, suppose that F has the property that states i with nearly equal feature vector have nearly equal optimal cost $J^*(i)$. Then we can form the sets S_y by grouping together states with nearly equal feature vector. In particular, given F, we introduce a more or less regular partition of the feature space [the subset of $\Re^m$ that consists of all possible feature vectors $F(i)$]. The partition of the feature space induces a possibly irregular collection of subsets of the original state space. Each of these subsets is then used as the footprint of a distinct aggregate state; see Fig. 3.5.7.

Note that in the resulting aggregation scheme the number of aggregate states may become very large. On the other hand, there is a significant advantage over the linear feature-based architectures of Section 3.1, which assign a single weight to each feature: in feature-based hard aggregation we are assigning *a weight to each subset of the feature space partition* (possibly a weight to every possible feature value, in the extreme case where each feature value is viewed by itself as a distinct set of the partition). In effect we use aggregation to construct a *nonlinear* (piecewise constant) feature-based architecture, which may be much more powerful than the corresponding linear architecture.

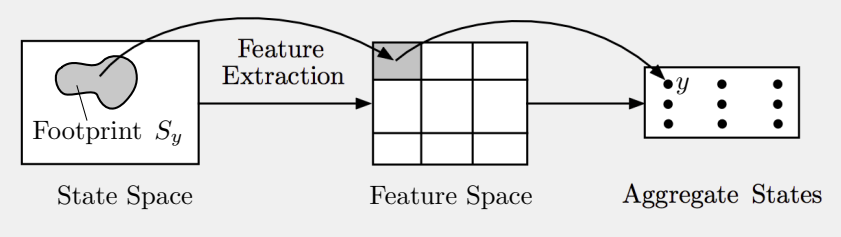


Figure 3.5.7 Feature-based hard aggregation using a partition of the space of features. Each aggregate state y has a footprint S_y that consists of states with "similar" features, i.e., states that map into the same subset of a partition in the space of features.

The question now arises how to obtain a suitable feature vector when there is no obvious choice, based on problem-specific considerations. One possibility, discussed in the book [Ber19a] (Section 6.4), is to obtain "good" features by using a neural network. In fact any method that automatically generates features from data may be used. Here we will discuss a simple possibility.

Using Scoring Functions

Suppose that we have obtained in some way a real-valued *scoring function* $V(i)$ of the state i, which serves as an index of undesirability of state i as a starting state (smaller values of V are assigned to more desirable states, consistent with the view of V as some form of "cost" function). One possibility is to use as V an approximation of the cost function of some "good" (e.g., near-optimal) policy. Another possibility is to obtain V by problem approximation, i.e., as the cost function of some reasonable policy applied to an approximation of the original problem. Still another possibility is to obtain V by training a neural network or other architecture using samples of state-cost pairs obtained by using a software or human expert, and some supervised learning technique.

Given the scoring function V, we will construct a feature mapping that groups together states i with roughly equal scores $V(i)$. In particular, we let R_x, $x = 1, \ldots, q$, be q disjoint intervals that form a partition of the range of possible values of V [i.e., are such that for any state i, there is a unique interval R_x such that $V(i) \in R_x$]. We define a feature vector $F(i)$ of the state i according to

$$F(i) = x, \qquad \text{for all } i \text{ such that } V(i) \in R_x, \quad x = 1, \ldots, q. \tag{3.53}$$

This feature in turn defines a partition of the state space into the sets

$$I_x = \big\{ i \mid F(i) = x \big\} = \big\{ i \mid V(i) \in R_x \big\}, \qquad x = 1, \ldots, q. \tag{3.54}$$

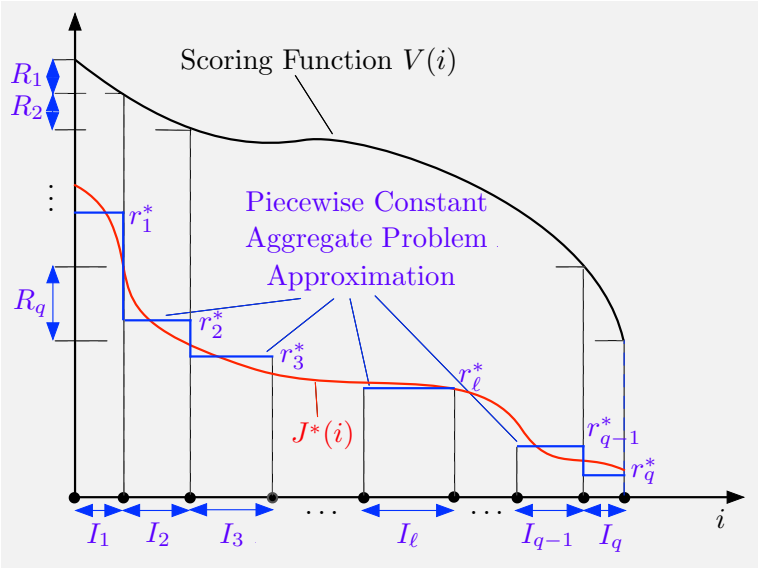


Figure 3.5.8. Hard aggregation scheme based on a single scoring function. We introduce q disjoint intervals $R_1, \ldots, R_q$ that form a partition of the set of possible values of V, and we define a feature vector $F(i)$ of the state i according to

$$F(i) = x, \qquad \text{for all } i \text{ such that } V(i) \in R_x, \ x = 1, \ldots, q.$$

This feature vector in turn defines a partition of the state space into the sets

$$I_x = \big\{ i \mid F(i) = x \big\} = \big\{ i \mid V(i) \in R_x \big\}, \qquad x = 1, \ldots, q.$$

The sets I_x coincide with the footprint sets S_x, and the solution of the aggregate problem yields a piecewise constant approximation of the optimal cost function of the original problem.

Assuming that all the sets I_x are nonempty, we thus obtain a hard aggregation scheme, where the aggregate states are $x = 1, \ldots, q$, and the aggregation probabilities are defined by

$$\phi_{jx} = \begin{cases} 1 & \text{if } j \in I_x, \\ 0 & \text{otherwise,} \end{cases} \qquad j = 1, \ldots, n, \ x = 1, \ldots, q, \tag{3.55}$$

see Fig. 3.5.8. Note that the sets I_x coincide with the footprint sets S_x.

The following proposition (due to Tsitsiklis and VanRoy [TsV96]) illustrates the important role of the *quantization error*, defined as

$$\delta = \max_{x=1,\ldots,q} \max_{i,j \in I_x} \big| V(i) - V(j) \big|. \tag{3.56}$$

It represents the maximum error that can be incurred by approximating V within each set I_x with a single value from its range within the subset. Its

proof with additional discussion can be found in Chapter 6 of the author's RL book [Ber19a].

Proposition 3.5.2: Consider the hard aggregation scheme defined by a scoring function V as described above. Assume that the variations of J^* and V over the sets $I_1, \ldots, I_q$ are within a factor $\beta \geq 0$ of each other, i.e., that

$$\big|J^*(i) - J^*(j)\big| \leq \beta \big|V(i) - V(j)\big|, \qquad \text{for all } i, j \in I_x,\ x = 1, \ldots, q.$$

(a) We have

$$\big|J^*(i) - r_x^*\big| \leq \frac{\beta\delta}{1-\alpha}, \qquad \text{for all } i \in I_x,\ x = 1, \ldots, q,$$

where δ is the quantization error of Eq. (3.56).

(b) Assume that there is no quantization error, i.e., V and J^* are constant within each set I_x. Then the aggregation scheme yields the optimal cost function J^* exactly, i.e.,

$$J^*(i) = r_x^*, \qquad \text{for all } i \in I_x,\ x = 1, \ldots, q.$$

3.5.7 Biased Aggregation

In this section we will introduce an extension of the preceding aggregation framework. This extension involves a vector

$$V = \big(V(1), \ldots, V(n)\big)$$

called the *bias vector* or *bias function*, which affects the cost structure of the aggregate problem, and biases the values of its optimal cost function towards their correct levels. When $V = 0$, we will obtain the aggregation scheme of Section 3.5.4. When $V \neq 0$, we will obtain a different aggregation scheme, which yields an approximation to J^* that is equal to V plus a local correction; see Fig. 3.5.10. In this case the aggregate DP problem aims to provide a correction/improvement to V, which may itself be a reasonably good estimate of J^*.

An obvious context where biased aggregation can be used is to improve on an approximation to J^* obtained using a different method, such as for example by neural network-based approximate PI, by rollout, or by problem approximation. Generally, we may speculate that if V captures

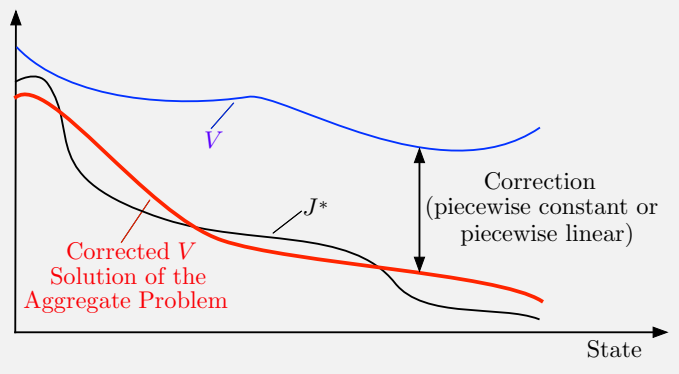


Figure 3.5.10 Schematic illustration of biased aggregation. It provides an approximation to J^* that is equal to the bias function V plus a local correction. When $V = 0$, we obtain the classical aggregation framework.

a fair amount of the nonlinearity of J^*, we may reduce the number of aggregate states needed for adequate performance.

Let us now formulate the aggregate problem in biased aggregation. It is a discounted infinite horizon problem that is similar to the (unbiased) aggregate problem of Section 3.5.4. It involves three sets of states: two copies of the original state space, denoted I_0 and I_1, as well as a finite set $\mathcal{A}$ of aggregate states, as depicted in Fig. 3.5.11. The state transitions in the aggregate problem go from a state in $\mathcal{A}$ to a state in I_0, according to disaggregation probabilities, then to a state in I_1, and then back to a state in $\mathcal{A}$, according to aggregation probabilities, and the process is repeated. At state $i \in I_0$ we must choose a control $u \in U(i)$, and then transition to a state $j \in I_1$ at a cost $g(i, u, j)$ according to the original system transition probabilities $p_{ij}(u)$.

The salient new characteristic of the biased aggregation scheme is a (possibly nonzero) cost $-V(i)$ for transition from any aggregate state to a state $i \in I_0$, and of a cost $V(j)$ from a state $j \in I_1$ to any aggregate state; cf. Fig. 3.5.11. The function V is the bias function, and we will argue that V *should be chosen as close as possible to* J^*. Moreover, for practical purposes its values at various states should be easily computable.

A key insight is that *biased aggregation can be viewed as unbiased aggregation applied to a modified DP problem*, which is equivalent to the original DP problem in the sense that it has the same optimal policies. The modified DP problem is obtained from the original by changing its cost per stage from $g(i, u, j)$ to

$$g(i, u, j) - V(i) + \alpha V(j), \qquad i, j = 1, \ldots, n, \ u \in U(i). \tag{3.57}$$

In particular, by comparing Figs. 3.5.6 and 3.5.11 it can be seen that unbiased aggregation applied to the modified DP problem gives the same

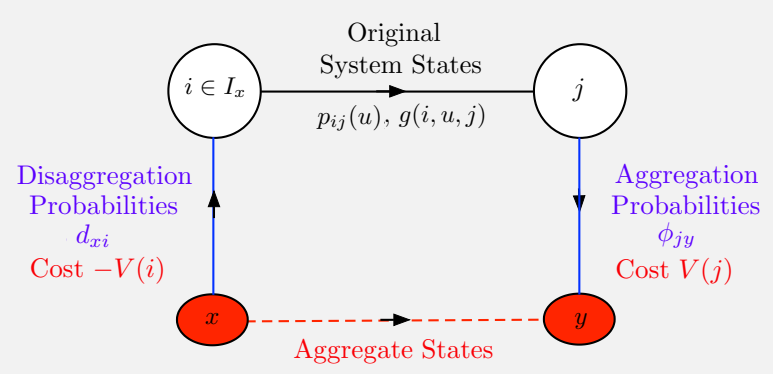


Figure 3.5.11 Illustration of the transition mechanism and the costs per stage of the aggregate problem in the biased aggregation framework. When the bias function V is identically zero, we obtain the aggregation framework of Section 3.5.4.

state-control trajectories as biased aggregation applied to the original DP problem, while the incurred transition costs (from aggregate state to aggregate state) are equal.

Moreover, there is a close connection between the optimal cost functions of the modified DP problem with cost per stage given by Eq. (3.57), and the original DP problem. In particular, the optimal cost function of the modified problem, call it $\tilde{J}$, satisfies the corresponding Bellman equation:

$$\tilde{J}(i) = \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) - V(i) + \alpha V(j) + \alpha \tilde{J}(j)\big), \quad i = 1, \ldots, n,$$

or equivalently

$$\tilde{J}(i) + V(i) = \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)\Big(g(i,u,j) + \alpha\big(\tilde{J}(j) + V(j)\big)\Big), \quad i = 1, \ldots, n.$$

By comparing this equation with the Bellman equation for the original problem, we see that the optimal cost functions of the modified and the original problems are related by

$$J^*(i) = \tilde{J}(i) + V(i), \qquad i = 1, \ldots, n,$$

and that the two problems have the same optimal policies. This of course assumes that the original and modified problems are solved exactly. If instead they are solved approximately using aggregation or another approximation architecture, such as a neural network, the policies obtained may be substantially different. In particular, the choice of V and the approximation architecture may affect substantially the quality of suboptimal policies obtained.

To summarize, any unbiased aggregation scheme and algorithm, when applied to the modified DP problem with cost per stage given by Eq. (3.57), yields a biased aggregation scheme and algorithm for the original DP problem. Thus, we can straightforwardly transfer results, algorithms, and intuition from our earlier unbiased aggregation analysis to the biased aggregation framework, by applying them to the unbiased aggregation framework that corresponds to the modified stage cost (3.57). Moreover, we may use simulation-based algorithms for policy evaluation, policy improvement, and Q-learning for the aggregate problem, with the only requirement that the value $V(i)$ for any state i is available when needed.

3.5.8 Asynchronous Distributed Multiagent Aggregation

Let us now discuss the distributed solution of large-scale discounted DP problems using cost function approximation, multiple agents/processors, and hard aggregation. Here we partition the original system states into aggregate states/subsets $x \in \mathcal{A} = \{x_1, \ldots, x_m\}$, and we envision a network of processors/agents, each updating asynchronously a detailed/exact local cost function, defined on a single aggregate state/subset. Each processor also maintains an aggregate cost for its aggregate state, which is a weighted average of the detailed cost of the (original system) states in the processor's subset, weighted by the corresponding disaggregation probabilities. These aggregate costs are communicated between processors and are used to perform the local updates.

In a synchronous VI method of this type, each processor $\ell = 1, \ldots, m$, maintains/updates a (local) cost $J(i)$ for every original system state $i \in x_\ell$, and an aggregate cost

$$R(\ell) = \sum_{i \in x_\ell} d_{x_\ell i} J(i),$$

where $d_{x_\ell i}$ are the corresponding disaggregation probabilities. We generically denote by J and R the vectors with components $J(i)$, $i = 1, \ldots, n$, and $R(\ell)$, $\ell = 1, \ldots, m$, respectively. These components are updated according to

$$J_{k+1}(i) = \min_{u \in U(i)} H_\ell(i, u, J_k, R_k), \qquad \forall\ i \in x_\ell, \tag{3.58}$$

with

$$R_k(\ell) = \sum_{i \in x_\ell} d_{x_\ell i} J_k(i), \qquad \ell = 1, \ldots, m, \tag{3.59}$$

where the mapping H_ℓ is defined for all $\ell = 1, \ldots, m$, $i \in x_\ell$, $u \in U(i)$, and $J \in \Re^n$, $R \in \Re^m$, by

$$H_\ell(i, u, J, R) = \sum_{j=1}^{n} p_{ij}(u) g(i, u, j) + \alpha \sum_{j \in x_\ell} p_{ij}(u) J(j) + \alpha \sum_{j \notin x_\ell} p_{ij}(u) R\big(x(j)\big), \tag{3.60}$$

and where for each original system state j, we denote by $x(j)$ the subset to which j belongs [i.e., $j \in x(j)$]. Thus the iteration (3.58) is the same as ordinary VI, except that instead of $J(j)$, we use the aggregate costs $R\big(x(j)\big)$ for the states j whose costs are updated by other processors.

It is possible to show that the iteration (3.58)-(3.59) involves a sup-norm contraction mapping of modulus α, so it converges to the unique solution of the system of equations in (J, R)

$$J(i) = \min_{u \in U(i)} H_\ell(i, u, J, R), \qquad R(\ell) = \sum_{i \in x_\ell} d_{x_\ell i} J(i), \qquad \forall\ i \in x_\ell,\ \ell = 1, \ldots, m. \tag{3.61}$$

This follows from the fact that $\{d_{x_\ell i} \mid i = 1, \ldots, n\}$ is a probability distribution. We may view the equations (3.61) as a set of Bellman equations for an "aggregate" DP problem, which similar to our earlier discussion, involves both the original and the aggregate system states. The difference from the Bellman equations (3.47)-(3.49) is that the mapping (3.60) involves $J(j)$ rather than $R\big(x(j)\big)$ for $j \in x_\ell$.

In the algorithm (3.58)-(3.59), all processors ℓ must be updating their local costs $J(i)$ and aggregate costs $R(\ell)$ synchronously, and communicate the aggregate costs to the other processors before a new iteration may begin. This is often impractical and time-wasting. In a more practical asynchronous version of the method, the aggregate costs $R(\ell)$ may be outdated to account for communication "delays" between processors. Moreover, the costs $J(i)$ need not be updated for all i; it is sufficient that they are updated by each processor ℓ only for a (possibly empty) subset of $I_{\ell,k}$ of the aggregate state/set x_ℓ. In this case, the iteration (3.58)-(3.59) is modified to take the form

$$J_{k+1}(i) = \min_{u \in U(i)} H_\ell\big(i, u, J_k, R_{\tau_{1,k}}(1), \ldots, R_{\tau_{m,k}}(m)\big), \qquad \forall\ i \in I_{\ell,k}, \tag{3.62}$$

with $0 \leq \tau_{\ell,k} \leq k$ for $\ell = 1, \ldots, m$, and

$$R_\tau(\ell) = \sum_{i \in x_\ell} d_{x_\ell i} J_\tau(i), \qquad \forall\ \ell = 1, \ldots, m.$$

The differences $k - \tau_{\ell,k}$, $\ell = 1, \ldots, m$, in Eq. (3.62) may be viewed as "delays" between the current time k and the times $\tau_{\ell,k}$ when the corresponding aggregate costs were computed at other processors. For convergence, it is of course essential that every $i \in x_\ell$ belongs to $I_{\ell,k}$ for infinitely many k (so each cost component is updated infinitely often), and $\lim_{k\to\infty} \tau_{\ell,k} = \infty$ for all $\ell = 1, \ldots, m$ (so that processors eventually communicate more recently computed aggregate costs to other processors).

The convergence of this type of method based on the sup-norm contraction property of the mapping underlying Eq. (3.61), can be established

using an asynchronous convergence theory for DP developed by the author in the paper [Ber82] (see also the books [BeT89], [Ber12]). The monotonicity property is also sufficient to establish convergence, and this is useful in the convergence analysis of related aggregation algorithms for nondiscounted DP models (see the paper by Bertsekas and Yu [BeY10]).

3.6 NOTES AND SOURCES

Section 3.1: Our discussion of approximation architectures, neural networks, and training has been limited, and aimed just to provide the connection with approximate DP. The literature on the subject is vast, and some of the textbooks mentioned in the references to Chapter 1 provide detailed accounts and many sources, in addition to the ones given in Sections 3.1 and 3.2.

There are two broad directions of inquiry in parametric architectures:

(1) The design of architectures, either in a general or a problem-specific context.

(2) The training of neural networks, as well as other linear and nonlinear architectures.

Research along both of these directions has been extensive and is continuing.

Methods for selection of basis functions have received much attention, particularly in the context of neural network research and deep reinforcement learning (see e.g., the book by Goodfellow, Bengio, and Courville [GBC16]). For discussions that are focused outside the neural network area, see Bertsekas and Tsitsiklis [BeT96], Keller, Mannor, and Precup [KMP06], Jung and Polani [JuP07], Bertsekas and Yu [BeY09], and Bhatnagar, Borkar, and Prashanth [BBP13]. Moreover, there has been considerable research on optimal feature selection within given parametric classes (see Menache, Mannor, and Shimkin [MMS05], Yu and Bertsekas [YuB09], Busoniu et al. [BBD10a], and Di Castro and Mannor [DiM10]).

Incremental algorithms are the principal methods for training approximation architectures. They are supported by substantial theoretical analysis, which addresses issues of convergence, rate of convergence, stepsize selection, and component order selection. Moreover, incremental algorithms have been extended to constrained optimization settings, where the constraints are also treated incrementally, first by Nedić [Ned11], and then by several other authors: Bertsekas [Ber11a], Wang and Bertsekas [WaB15], [WaB16], Bianchi [Bia16], Iusem, Jofre, and Thompson [IJT18]. It is beyond our scope to cover this analysis. The author's surveys [Ber10a] and [Ber15b], and convex optimization and nonlinear programming textbooks [Ber15a], [Ber16], collectively contain an extensive account of incremental methods, including the Kaczmarz, incremental gradient, subgradient, ag-

gregated gradient, Newton, Gauss-Newton, and extended Kalman filtering methods, and give many references. The book [BeT96] and paper [BeT00] by Bertsekas and Tsitsiklis, and the survey by Bottou, Curtis, and Nocedal [BCN18] provide theoretically oriented treatments.

Section 3.2: The publicly and commercially available neural network training programs incorporate heuristics for scaling and preprocessing data, stepsize selection, initialization, etc, which can be very effective in specialized problem domains. We refer to books on neural networks such as Bishop [Bis95], Goodfellow, Bengio, and Courville [GBC16], and Haykin [Hay08].

Deep neural networks have created a lot of excitement in the machine learning field, in view of some high profile successes in image and speech recognition, and in RL with the AlphaGo and AlphaZero programs. One question is whether and for what classes of target functions we can enhance approximation power by increasing the number of layers while keeping the number of weights constant. For analysis and speculation around this question, see Bengio [Ben09], Liang and Srikant [LiS16], Yarotsky [Yar17], Daubechies et al. [DDF19], and the references quoted there.

Another important research question relates to the role of overparametrization in the success of deep neural networks. With more weights than training data, the training problem has infinitely many solutions, each providing an architecture that fits the training data perfectly. The question then is how to select a solution that works well on test data (i.e., data outside the training set); see Zhang et al. [ZBH16], [ZBH21], Belkin, Ma, and Mandal [BMM18], Belkin, Rakhlin, and Tsybakov [BRT18], Soltanolkotabi, Javanmard, and Lee [SJL18], Bartlett et al. [BLL19], Hastie et al. [HMR19], Muthukumar, Vodrahalli, and Sahai [MVS19], Su and Yang [SuY19], Sun [Sun19], Vaswani et al. [VLK21], Zhang et al. [ZBH21] and the discussion in the book by Hardt and Recht [HaR21].

Section 3.3: Fitted value iteration has a long history; it was mentioned by Bellman among others. It has interesting properties, and at times exhibits pathological/unstable behavior due to accumulation of errors over a long horizon (see [Ber19a], Section 5.2).

The approximate policy iteration method of Section 3.3.3 has been proposed by Fern, Yoon, and Givan [FYG06], and variants have also been discussed and analyzed by several other authors. The method (with some variations) has been used to train a tetris playing computer program that performs impressively better than programs that are based on other variants of approximate policy iteration, and various other methods; see Scherrer [Sch13], Scherrer et al. [SGG15], and Gabillon, Ghavamzadeh, and Scherrer [GGS13], who also provide an analysis of the method. The RL and approximate DP books collectively describe several alternative simulation-based methods for policy evaluation, including the popular temporal difference methods; see e.g., [BeT96], [SuB18], [Ber12], Chapters 6 and 7. The book [Ber20a] describes distributed versions of approximate policy itera-

tion, which are based on partitioning of the state space.

The original proposal of SARSA (Section 3.3.4) is attributed to Rummery and Niranjan [RuN94], with related work presented in the papers by Peng and Williams [PeW96], and Wiering and Schmidhuber [WiS98]. The ideas of the DQN algorithm attracted much attention following the paper by Mnih et al. [MKS15], which reported impressive test results on a suite of 49 classic Atari 2600 games.

The rollout and approximate PI methodology for POMDP of Section 3.3.5 was described in the author's RL book [Ber19a]. It was extended and tested in the paper by Bhattacharya et al. [BBW20] in the context of a challenging pipeline repair problem.

Advantage updating (Section 3.3.6) was proposed by Baird [Bai93], [Bai94], and is discussed further in Section 6.6 of the neuro-dynamic programming book [BeT96]. The differential training methodology (Section 3.3.7) was proposed by the author in the paper [Ber97b], and followup work was presented by Weaver and Baxter [WeB99].

Section 3.4: Classification (sometimes called "pattern classification" or "pattern recognition") is a major subject in machine learning, for which there are many approaches, an extensive literature, and an abundance of public domain and commercial software; see e.g. the textbooks by Bishop [Bis95], [Bis06], Duda, Hart, and Stork [DHS12], and Hardt and Recht [HaR21]. Approximation in policy space was formulated as a classification problem in the context of DP by Lagoudakis and Parr [LaP03], and was followed up by several other authors (see e.g., Dimitrakakis and Lagoudakis [DiL08], Lazaric, Ghavamzadeh, and Munos [LGM10], Gabillon et al. [GLG11], Liu and Wei [LiW14], Farahmand et al. [FPB15], and the references quoted there). While we have focused on a classification approach that makes use of least squares regression and a parametric architecture, other classification methods may also be used. For example the paper [LaP03] discusses the use of nearest neighbor schemes, support vector machines, as well as neural networks.

Section 3.5: The aggregation approach has a long history in scientific computation and operations research (see for example Bean, Birge, and Smith [BBS87], Chatelin and Miranker [ChM82], Douglas and Douglas [DoD93], and Rogers et al. [RPW91]). It was introduced in the simulation-based approximate DP context, mostly in the form of VI; see Singh, Jaakkola, and Jordan [SJJ95], Gordon [Gor95], and Tsitsiklis and Van Roy [TsV96]. It was further discussed in the neuro-dynamic programming book [BeT96], Sections 3.1.2 and 6.7.

The aggregation framework with representative features was introduced in the author's book [Ber12], was discussed in detail in the RL textbook [Ber19a] (Chapter 6), and was further developed in the author's survey paper [Ber18b], which provides an expanded view of the methodology. Biased aggregation (Section 3.5.7) was first proposed in the author's

paper [Ber18c], which contains further discussion, connections with rollout algorithms, and additional methods.

Distributed asynchronous aggregation (Section 3.5.8) was first proposed in the paper by Bertsekas and Yu [BeY10] (Example 2.5); see also the discussions in author's DP books [Ber12] (Section 6.5.4) and [Ber22b] (Example 1.2.11). A recent computational study related to distributed traffic routing is given by Vertovec and Margellos [VeM23].

EXERCISES

3.1 (Proof of Prop. 3.4.1)

Complete the details of the following proof of Prop. 3.4.1. Consider for any pair (c,x) the conditional expected value $E\Big\{\big(z(c,x)-y\big)^2 \;\Big|\; c,x\Big\}$, where y is any scalar. Given (c,x), the random variable $z(c,x)$ takes the value $z(c,x)=1$ with probability $p(c\,|\,x)$ and the value $z(c,x)=0$ with probability $1-p(c\,|\,x)$, so we have

$$E\Big\{\big(z(c,x)-y\big)^2 \;\Big|\; c,x\Big\} = p(c\,|\,x)(y-1)^2 + \big(1-p(c\,|\,x)\big)y^2.$$

We minimize this expression with respect to y, by setting to 0 its derivative, i.e.,

$$0 = 2p(c\,|\,x)(y-1) + 2\big(1-p(c\,|\,x)\big)y = 2\big(-p(c\,|\,x)+y\big).$$

We thus obtain the minimizing value of y, namely $y^* = p(c\,|\,x)$, so that

$$E\Big\{\big(z(c,x)-p(c\,|\,x)\big)^2 \mid c,x\Big\} \leq E\Big\{\big(z(c,x)-y\big)^2 \mid c,x\Big\}, \qquad \text{for all scalars } y.$$

For any function h of (c,x) we set $y=h(c,x)$ in the above expression and obtain

$$E\Big\{\big(z(c,x)-p(c\,|\,x)\big)^2 \mid c,x\Big\} \leq E\Big\{\big(z(c,x)-h(c,x)\big)^2 \mid c,x\Big\}.$$

Since this is true for all (c,x), we also have

$$\sum_{(c,x)} \zeta(c,x) E\Big\{\big(z(c,x)-p(c\,|\,x)\big)^2 \;\Big|\; c,x\Big\} \leq \sum_{(c,x)} \zeta(c,x) E\Big\{\big(z(c,x)-h(c,x)\big)^2 \;\Big|\; c,x\Big\},$$

for all functions h. By using the theorem of total probability (see e.g., [BeT08], Chapter 1), this is equivalent to

$$E\Big\{\big(z(c,x)-p(c\,|\,x)\big)^2\Big\} \leq E\Big\{\big(z(c,x)-h(c,x)\big)^2\Big\},$$

i.e., Eq. (3.35) holds.

References

[ABB19] Agrawal, A., Barratt, S., Boyd, S., and Stellato, B., 2019. "Learning Convex Optimization Control Policies," arXiv:1912.09529.

[ACF02] Auer, P., Cesa-Bianchi, N., and Fischer, P., 2002. "Finite Time Analysis of the Multiarmed Bandit Problem," Machine Learning, Vol. 47, pp. 235-256.

[ADH19] Arora, S., Du, S. S., Hu, W., Li, Z., and Wang, R., 2019. "Fine-Grained Analysis of Optimization and Generalization for Overparameterized Two-Layer Neural Networks," arXiv:1901.08584.

[AHZ19] Arcari, E., Hewing, L., and Zeilinger, M. N., 2019. "An Approximate Dynamic Programming Approach for Dual Stochastic Model Predictive Control," arXiv:1911.03728.

[ALZ08] Asmuth, J., Littman, M. L., and Zinkov, R., 2008. "Potential-Based Shaping in Model-Based Reinforcement Learning," Proc. of 23rd AAAI Conference, pp. 604-609.

[AMS09] Audibert, J.Y., Munos, R., and Szepesvari, C., 2009. "Exploration-Exploitation Tradeoff Using Variance Estimates in Multi-Armed Bandits," Theoretical Computer Science, Vol. 410, pp. 1876-1902.

[ASR20] Andersen, A. R., Stidsen, T. J. R., and Reinhardt, L. B., 2020. "Simulation-Based Rolling Horizon Scheduling for Operating Theatres," in SN Operations Research Forum, Vol. 1, pp. 1-26.

[AXG16] Ames, A. D., Xu, X., Grizzle, J. W., and Tabuada, P., 2016. "Control Barrier Function Based Quadratic Programs for Safety Critical Systems," IEEE Transactions on Automatic Control, Vol. 62, pp. 3861-3876.

[Abr90] Abramson, B., 1990. "Expected-Outcome: A General Model of Static Evaluation," IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol. 12, pp. 182-193.

[Agr95] Agrawal, R., 1995. "Sample Mean Based Index Policies with $O(\log n)$ Regret for the Multiarmed Bandit Problem," Advances in Applied Probability, Vol. 27, pp. 1054-1078.

[Ala22] Alamir, M., 2022. "Learning Against Uncertainty in Control Engineering," Annual Reviews in Control.

[AnH14] Antunes, D., and Heemels, W.P.M.H., 2014. "Rollout Event-Triggered Control: Beyond Periodic Control Performance," IEEE Transactions on Automatic Control, Vol. 59, pp. 3296-3311.

[AnM79] Anderson, B. D. O., and Moore, J. B., 1979. Optimal Filtering, Prentice-Hall, Englewood Cliffs, N. J.

[AsH06] Aström, K. J., and Hagglund, T., 2006. Advanced PID Control, Instrument Society of America, Research Triangle Park, N. C.

[AsW94] Aström, K. J., and Wittenmark, B., 1994. Adaptive Control, 2nd Edition, Prentice-Hall, Englewood Cliffs, N. J.

[Ast83] Aström, K. J., 1983. "Theory and Applications of Adaptive Control - A Survey," Automatica, Vol. 19, pp. 471-486.

[AtF66] Athans, M., and Falb, P., 1966. Optimal Control, McGraw-Hill, N. Y.

[AvB20] Avrachenkov, K., and Borkar, V. S., 2020. "Whittle Index Based Q-Learning for Restless Bandits with Average Reward," arXiv:2004.14427.

[BBB22] Bhambri, S., Bhattacharjee, A., and Bertsekas, D. P., 2022. "Reinforcement Learning Methods for Wordle: A POMDP/Adaptive Control Approach," arXiv:2211.10298.

[BBD08] Busoniu, L., Babuska, R., and De Schutter, B., 2008. "A Comprehensive Survey of Multiagent Reinforcement Learning," IEEE Transactions on Systems, Man, and Cybernetics, Part C, Vol. 38, pp. 156-172.

[BBD10a] Busoniu, L., Babuska, R., De Schutter, B., and Ernst, D., 2010. Reinforcement Learning and Dynamic Programming Using Function Approximators, CRC Press, N. Y.

[BBD10b] Busoniu, L., Babuska, R., and De Schutter, B., 2010. "Multi-Agent Reinforcement Learning: An Overview," in Innovations in Multi-Agent Systems and Applications, Springer, pp. 183-221.

[BBG13] Bertazzi, L., Bosco, A., Guerriero, F., and Lagana, D., 2013. "A Stochastic Inventory Routing Problem with Stock-Out," Transportation Research, Part C, Vol. 27, pp. 89-107.

[BBM17] Borrelli, F., Bemporad, A., and Morari, M., 2017. Predictive Control for Linear and Hybrid Systems, Cambridge Univ. Press, Cambridge, UK.

[BBP13] Bhatnagar, S., Borkar, V. S., and Prashanth, L. A., 2013. "Adaptive Feature Pursuit: Online Adaptation of Features in Reinforcement Learning," in *Reinforcement Learning and Approximate Dynamic Programming for Feedback Control*, by F. Lewis and D. Liu (eds.), IEEE Press, Piscataway, N. J., pp. 517-534.

[BBW20] Bhattacharya, S., Badyal, S., Wheeler, T., Gil, S., Bertsekas, D. P., 2020. "Reinforcement Learning for POMDP: Partitioned Rollout and Policy Iteration with Application to Autonomous Sequential Repair Problems," to appear in IEEE Robotics and Automation Letters, 2020; arXiv:2002.04175.

[BCD10] Brochu, E., Cora, V. M., and De Freitas, N., 2010. "A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning," arXiv:1012.2599.

[BCN18] Bottou, L., Curtis, F. E., and Nocedal, J., 2018. "Optimization Methods for Large-Scale Machine Learning," SIAM Review, Vol. 60, pp. 223-311.

[BFH86] Breton, M., Filar, J. A., Haurie, A., and Schultz, T. A., 1986. "On the Computation of Equilibria in Discounted Stochastic Dynamic Games," in Dynamic Games and Applications in Economics, Springer, pp. 64-87.

[BLJ23] Bai, T., Li, Y., Johansson, K. H., and Martensson, J., 2023. "Rollout-Based Charging Strategy for Electric Trucks with Hours-of-Service Regulations," arXiv Preprint, arXiv:2303.08895.

[BKB20] Bhattacharya, S., Kailas, S., Badyal, S., Gil, S., and Bertsekas, D. P., 2020. "Multiagent Rollout and Policy Iteration for POMDP with Application to Multi-Robot

Repair Problems," in Proc. of Conference on Robot Learning (CoRL); also arXiv preprint, arXiv:2011.04222.

[BLL19] Bartlett, P. L., Long, P. M., Lugosi, G., and Tsigler, A., 2019. "Benign Overfitting in Linear Regression," arXiv:1906.11300.

[BMM18] Belkin, M., Ma, S., and Mandal, S., 2018. "To Understand Deep Learning we Need to Understand Kernel Learning," arXiv:1802.01396.

[BPW12] Browne, C., Powley, E., Whitehouse, D., Lucas, L., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S., 2012. "A Survey of Monte Carlo Tree Search Methods," IEEE Trans. on Computational Intelligence and AI in Games, Vol. 4, pp. 1-43.

[BRT18] Belkin, M., Rakhlin, A., and Tsybakov, A. B., 2018. "Does Data Interpolation Contradict Statistical Optimality?" arXiv:1806.09471.

[BTW97] Bertsekas, D. P., Tsitsiklis, J. N., and Wu, C., 1997. "Rollout Algorithms for Combinatorial Optimization," Heuristics, Vol. 3, pp. 245-262.

[BWL19] Beuchat, P. N., Warrington, J., and Lygeros, J., 2019. "Accelerated Point-Wise Maximum Approach to Approximate Dynamic Programming," arXiv:1901.03619.

[BYB94] Bradtke, S. J., Ydstie, B. E., and Barto, A. G., 1994. "Adaptive Linear Quadratic Control Using Policy Iteration," Proc. IEEE American Control Conference, Vol. 3, pp. 3475-3479.

[BaF88] Bar-Shalom, Y., and Fortman, T. E., 1988. Tracking and Data Association, Academic Press, N. Y.

[BaL19] Banjac, G., and Lygeros, J., 2019. "A Data-Driven Policy Iteration Scheme Based on Linear Programming," Proc. 2019 IEEE CDC, pp. 816-821.

[BaP12] Bauso, D., and Pesenti, R., 2012. "Team Theory and Person-by-Person Optimization with Binary Decisions," SIAM Journal on Control and Optimization, Vol. 50, pp. 3011-3028.

[Bai93] Baird, L. C., 1993. "Advantage Updating," Report WL-TR-93-1146, Wright Patterson AFB, OH.

[Bai94] Baird, L. C., 1994. "Reinforcement Learning in Continuous Time: Advantage Updating," International Conf. on Neural Networks, Orlando, Fla.

[Bar90] Bar-Shalom, Y., 1990. Multitarget-Multisensor Tracking: Advanced Applications, Artech House, Norwood, MA.

[BeC89] Bertsekas, D. P., and Castañon, D. A., 1989. "The Auction Algorithm for Transportation Problems," Annals of Operations Research, Vol. 20, pp. 67-96.

[BeC99] Bertsekas, D. P., and Castañon, D. A., 1999. "Rollout Algorithms for Stochastic Scheduling Problems," Heuristics, Vol. 5, pp. 89-108.

[BeC02] Ben-Gal, I., and Caramanis, M., 2002. "Sequential DOE via Dynamic Programming," IIE Transactions, Vol. 34, pp. 1087-1100.

[BeC08] Besse, C., and Chaib-draa, B., 2008. "Parallel Rollout for Online Solution of DEC-POMDPs," Proc. of 21st International FLAIRS Conference, pp. 619-624.

[BeL14] Beyme, S., and Leung, C., 2014. "Rollout Algorithm for Target Search in a Wireless Sensor Network," 80th Vehicular Technology Conference (VTC2014), IEEE, pp. 1-5.

[BeI96] Bertsekas, D. P., and Ioffe, S., 1996. "Temporal Differences-Based Policy Iter-

ation and Applications in Neuro-Dynamic Programming," Lab. for Info. and Decision Systems Report LIDS-P-2349, Massachusetts Institute of Technology.

[BeP03] Bertsimas, D., and Popescu, I., 2003. "Revenue Management in a Dynamic Network Environment," Transportation Science, Vol. 37, pp. 257-277.

[BeR71a] Bertsekas, D. P., and Rhodes, I. B., 1971. "On the Minimax Reachability of Target Sets and Target Tubes," Automatica, Vol. 7, pp. 233-247.

[BeR71b] Bertsekas, D. P., and Rhodes, I. B., 1971. "Recursive State Estimation for a Set-Membership Description of the Uncertainty," IEEE Trans. Automatic Control, Vol. AC-16, pp. 117-128.

[BeR73] Bertsekas, D. P., and Rhodes, I. B., 1973. "Sufficiently Informative Functions and the Minimax Feedback Control of Uncertain Dynamic Systems," IEEE Trans. Automatic Control, Vol. AC-18, pp. 117-124.

[BeS78] Bertsekas, D. P., and Shreve, S. E., 1978. Stochastic Optimal Control: The Discrete Time Case, Academic Press, N. Y.; republished by Athena Scientific, Belmont, MA, 1996 (can be downloaded in from the author's website).

[BeS18] Bertazzi, L., and Secomandi, N., 2018. "Faster Rollout Search for the Vehicle Routing Problem with Stochastic Demands and Restocking," European J. of Operational Research, Vol. 270, pp.487-497.

[BeT89] Bertsekas, D. P., and Tsitsiklis, J. N., 1989. Parallel and Distributed Computation: Numerical Methods, Prentice-Hall, Englewood Cliffs, N. J.; republished by Athena Scientific, Belmont, MA, 1997 (can be downloaded from the author's website).

[BeT91] Bertsekas, D. P., and Tsitsiklis, J. N., 1991. "An Analysis of Stochastic Shortest Path Problems," Math. Operations Res., Vol. 16, pp. 580-595.

[BeT96] Bertsekas, D. P., and Tsitsiklis, J. N., 1996. Neuro-Dynamic Programming, Athena Scientific, Belmont, MA.

[BeT97] Bertsimas, D., and Tsitsiklis, J. N., 1997. Introduction to Linear Optimization, Athena Scientific, Belmont, MA.

[BeT00] Bertsekas, D. P., and Tsitsiklis, J. N., 2000. "Gradient Convergence of Gradient Methods with Errors," SIAM J. on Optimization, Vol. 36, pp. 627-642.

[BeT08] Bertsekas, D. P., and Tsitsiklis, J. N., 2008. Introduction to Probability, 2nd Edition, Athena Scientific, Belmont, MA.

[BeY09] Bertsekas, D. P., and Yu, H., 2009. "Projected Equation Methods for Approximate Solution of Large Linear Systems," J. of Computational and Applied Math., Vol. 227, pp. 27-50.

[BeY10] Bertsekas, D. P., and Yu, H., 2010. "Asynchronous Distributed Policy Iteration in Dynamic Programming," Proc. of Allerton Conf. on Communication, Control and Computing, Allerton Park, Ill, pp. 1368-1374.

[BeY12] Bertsekas, D. P., and Yu, H., 2012. "Q-Learning and Enhanced Policy Iteration in Discounted Dynamic Programming," Math. of Operations Research, Vol. 37, pp. 66-94.

[BeY16] Bertsekas, D. P., and Yu, H., 2016. "Stochastic Shortest Path Problems Under Weak Conditions," Lab. for Information and Decision Systems Report LIDS-2909, MIT.

[Bel56] Bellman, R., 1956. "A Problem in the Sequential Design of Experiments," Sankhya: The Indian Journal of Statistics, Vol. 16, pp. 221-229.

[Bel57] Bellman, R., 1957. Dynamic Programming, Princeton University Press, Princeton, N. J.

[Bel67] Bellman, R., 1967. Introduction to the Mathematical Theory of Control Processes, Academic Press, Vols. I and II, New York, N. Y.

[Bel84] Bellman, R., 1984. Eye of the Hurricane, World Scientific Publishing, Singapore.

[Ben09] Bengio, Y., 2009. "Learning Deep Architectures for AI," Foundations and Trends in Machine Learning, Vol. 2, pp. 1-127.

[Ber71] Bertsekas, D. P., 1971. "Control of Uncertain Systems With a Set-Membership Description of the Uncertainty," Ph.D. Dissertation, Massachusetts Institute of Technology, Cambridge, MA (can be downloaded from the author's website).

[Ber72a] Bertsekas, D. P., 1972. "Infinite Time Reachability of State Space Regions by Using Feedback Control," IEEE Trans. Automatic Control, Vol. AC-17, pp. 604-613.

[Ber72b] Bertsekas, D. P., 1972. "On the Solution of Some Minimax Control Problems," Proc. 1972 IEEE Decision and Control Conf., New Orleans, LA.

[Ber73] Bertsekas, D. P., 1973. "Linear Convex Stochastic Control Problems over an Infinite Horizon," IEEE Trans. Automatic Control, Vol. AC-18, pp. 314-315.

[Ber77] Bertsekas, D. P., 1977. "Monotone Mappings with Application in Dynamic Programming," SIAM J. on Control and Opt., Vol. 15, pp. 438-464.

[Ber79] Bertsekas, D. P., 1979. "A Distributed Algorithm for the Assignment Problem," Lab. for Information and Decision Systems Report, MIT, May 1979.

[Ber82] Bertsekas, D. P., 1982. "Distributed Dynamic Programming," IEEE Trans. Automatic Control, Vol. AC-27, pp. 610-616.

[Ber83] Bertsekas, D. P., 1983. "Asynchronous Distributed Computation of Fixed Points," Math. Programming, Vol. 27, pp. 107-120.

[Ber91] Bertsekas, D. P., 1991. Linear Network Optimization: Algorithms and Codes, MIT Press, Cambridge, MA (can be downloaded from the author's website).

[Ber96] Bertsekas, D. P., 1996. "Incremental Least Squares Methods and the Extended Kalman Filter," SIAM J. on Optimization, Vol. 6, pp. 807-822.

[Ber97a] Bertsekas, D. P., 1997. "A New Class of Incremental Gradient Methods for Least Squares Problems," SIAM J. on Optimization, Vol. 7, pp. 913-926.

[Ber97b] Bertsekas, D. P., 1997. "Differential Training of Rollout Policies," Proc. of the 35th Allerton Conference on Communication, Control, and Computing, Allerton Park, Ill.

[Ber98] Bertsekas, D. P., 1998. Network Optimization: Continuous and Discrete Models, Athena Scientific, Belmont, MA (can be downloaded from the author's website).

[Ber05a] Bertsekas, D. P., 2005. "Dynamic Programming and Suboptimal Control: A Survey from ADP to MPC," European J. of Control, Vol. 11, pp. 310-334.

[Ber05b] Bertsekas, D. P., 2005. "Rollout Algorithms for Constrained Dynamic Programming," Lab. for Information and Decision Systems Report LIDS-P-2646, MIT.

[Ber07] Bertsekas, D. P., 2007. "Separable Dynamic Programming and Approximate Decomposition Methods," IEEE Trans. on Aut. Control, Vol. 52, pp. 911-916.

[Ber10a] Bertsekas, D. P., 2010. "Incremental Gradient, Subgradient, and Proximal Methods for Convex Optimization: A Survey," Lab. for Information and Decision Systems Report LIDS-P-2848, MIT; a condensed version with the same title appears in

Optimization for Machine Learning, by S. Sra, S. Nowozin, and S. J. Wright, (eds.), MIT Press, Cambridge, MA, 2012, pp. 85-119.

[Ber10b] Bertsekas, D. P., 2010. "Williams-Baird Counterexample for Q-Factor Asynchronous Policy Iteration,"
http://web.mit.edu/dimitrib/www/Williams-Baird Counterexample.pdf.

[Ber11a] Bertsekas, D. P., 2011. "Incremental Proximal Methods for Large Scale Convex Optimization," Math. Programming, Vol. 129, pp. 163-195.

[Ber11b] Bertsekas, D. P., 2011. "Approximate Policy Iteration: A Survey and Some New Methods," J. of Control Theory and Applications, Vol. 9, pp. 310-335; a somewhat expanded version appears as Lab. for Info. and Decision Systems Report LIDS-2833, MIT, 2011.

[Ber12] Bertsekas, D. P., 2012. Dynamic Programming and Optimal Control, Vol. II, 4th Edition, Athena Scientific, Belmont, MA.

[Ber13a] Bertsekas, D. P., 2013. "Rollout Algorithms for Discrete Optimization: A Survey," Handbook of Combinatorial Optimization, Springer.

[Ber13b] Bertsekas, D. P., 2013. "λ-Policy Iteration: A Review and a New Implementation," in Reinforcement Learning and Approximate Dynamic Programming for Feedback Control, by F. Lewis and D. Liu (eds.), IEEE Press, Piscataway, N. J., pp. 381-409.

[Ber15a] Bertsekas, D. P., 2015. Convex Optimization Algorithms, Athena Scientific, Belmont, MA.

[Ber15b] Bertsekas, D. P., 2015. "Incremental Aggregated Proximal and Augmented Lagrangian Algorithms," Lab. for Information and Decision Systems Report LIDS-P-3176, MIT; arXiv:1507.1365936.

[Ber16] Bertsekas, D. P., 2016. Nonlinear Programming, 3rd Edition, Athena Scientific, Belmont, MA.

[Ber17a] Bertsekas, D. P., 2017. Dynamic Programming and Optimal Control, Vol. I, 4th Edition, Athena Scientific, Belmont, MA.

[Ber17b] Bertsekas, D. P., 2017. "Value and Policy Iteration in Deterministic Optimal Control and Adaptive Dynamic Programming," IEEE Transactions on Neural Networks and Learning Systems, Vol. 28, pp. 500-509.

[Ber18a] Bertsekas, D. P., 2018. "Feature-Based Aggregation and Deep Reinforcement Learning: A Survey and Some New Implementations," Lab. for Information and Decision Systems Report, MIT; arXiv:1804.04577; IEEE/CAA Journal of Automatica Sinica, Vol. 6, 2019, pp. 1-31.

[Ber18b] Bertsekas, D. P., 2018. "Biased Aggregation, Rollout, and Enhanced Policy Improvement for Reinforcement Learning," Lab. for Information and Decision Systems Report, MIT; arXiv:1910.02426.

[Ber19a] Bertsekas, D. P., 2019. Reinforcement Learning and Optimal Control, Athena Scientific, Belmont, MA.

[Ber19b] Bertsekas, D. P., 2019. "Robust Shortest Path Planning and Semicontractive Dynamic Programming," Naval Research Logistics, Vol. 66, pp. 15-37.

[Ber19c] Bertsekas, D. P., 2019. "Multiagent Rollout Algorithms and Reinforcement Learning," arXiv:1910.00120.

[Ber19d] Bertsekas, D. P., 2019. "Constrained Multiagent Rollout and Multidimensional Assignment with the Auction Algorithm," arXiv preprint, arxiv:2002.07407.

[Ber20a] Bertsekas, D. P., 2020. Rollout, Policy Iteration, and Distributed Reinforcement Learning, Athena Scientific, Belmont, MA.

[Ber20b] Bertsekas, D. P., 2020. "Multiagent Value Iteration Algorithms in Dynamic Programming and Reinforcement Learning," arXiv preprint, arxiv.org/abs/2005.01627; appears in Results in Control and Optimization Journal, Vol. 1, 2020.

[Ber21a] Bertsekas, D. P., 2021. "Multiagent Reinforcement Learning: Rollout and Policy Iteration," IEEE/CAA Journal of Automatica Sinica, Vol. 8, pp. 249-271.

[Ber21b] Bertsekas, D. P., 2021. "Distributed Asynchronous Policy Iteration for Sequential Zero-Sum Games and Minimax Control," arXiv:2107.10406

[Ber22a] Bertsekas, D. P., 2022. Lessons from AlphaZero for Optimal, Model Predictive, and Adaptive Control, Athena Scientific, Belmont, MA.

[Ber22b] Bertsekas, D. P., 2022. Abstract Dynamic Programming, 3rd Edition, Athena Scientific, Belmont, MA (can be downloaded from the author's website).

[Ber22c] Bertsekas, D. P., 2022. "Newton's Method for Reinforcement Learning and Model Predictive Control," Results in Control and Optimization, Vol. 7, 2022, pp. 100-121.

[Ber22d] Bertsekas, D. P., 2022. "Rollout Algorithms and Approximate Dynamic Programming for Bayesian Optimization and Sequential Estimation," arXiv:2212.07998.

[Bet10] Bethke, B. M., 2010. Kernel-Based Approximate Dynamic Programming Using Bellman Residual Elimination, Ph.D. Thesis, MIT.

[BiL97] Birge, J. R., and Louveaux, 1997. Introduction to Stochastic Programming, Springer, New York, N. Y.

[Bia16] Bianchi, P., 2016. "Ergodic Convergence of a Stochastic Proximal Point Algorithm," SIAM J. on Optimization, Vol. 26, pp. 2235-2260.

[Bis95] Bishop, C. M, 1995. Neural Networks for Pattern Recognition, Oxford University Press, N. Y.

[Bis06] Bishop, C. M, 2006. Pattern Recognition and Machine Learning, Springer, N. Y.

[BlG54] Blackwell, D., and Girshick, M. A., 1954. Theory of Games and Statistical Decisions, Wiley, N. Y.

[BlM08] Blanchini, F., and Miani, S., 2008. Set-Theoretic Methods in Control, Birkhauser, Boston.

[Bla86] Blackman, S. S., 1986. Multi-Target Tracking with Radar Applications, Artech House, Dehdam, MA.

[Bla99] Blanchini, F., 1999. "Set Invariance in Control – A Survey," Automatica, Vol. 35, pp. 1747-1768.

[Bod20] Bodson, M., 2020. Adaptive Estimation and Control, Independently Published.

[Bor08] Borkar, V. S., 2008. Stochastic Approximation: A Dynamical Systems Viewpoint, Cambridge Univ. Press.

[BrH75] Bryson, A., and Ho, Y. C., 1975. Applied Optimal Control: Optimization, Estimation, and Control, (revised edition), Taylor and Francis, Levittown, Penn.

[Bra21] Brandimarte, P., 2021. From Shortest Paths to Reinforcement Learning: A MATLAB-Based Tutorial on Dynamic Programming, Springer.

[BuK97] Burnetas, A. N., and Katehakis, M. N., 1997. "Optimal Adaptive Policies for Markov Decision Processes," Math. of Operations Research, Vol. 22, pp. 222-255.

[CBH09] Choi, H. L., Brunet, L., and How, J. P., 2009. "Consensus-Based Decentralized Auctions for Robust Task Allocation," IEEE Transactions on Robotics, Vol. 25, pp. 912-926.

[CFH05] Chang, H. S., Hu, J., Fu, M. C., and Marcus, S. I., 2005. "An Adaptive Sampling Algorithm for Solving Markov Decision Processes," Operations Research, Vol. 53, pp. 126-139.

[CFH13] Chang, H. S., Hu, J., Fu, M. C., and Marcus, S. I., 2013. Simulation-Based Algorithms for Markov Decision Processes, 2nd Edition, Springer, N. Y.

[CLT19] Chapman, M. P., Lacotte, J., Tamar, A., Lee, D., Smith, K. M., Cheng, V., Fisac, J. F., Jha, S., Pavone, M., and Tomlin, C. J., 2019. "A Risk-Sensitive Finite-Time Reachability Approach for Safety of Stochastic Dynamic Systems," arXiv:1902.11277.

[CMT87a] Clarke, D. W., Mohtadi, C., and Tuffs, P. S., 1987. "Generalized Predictive Control - Part I. The Basic Algorithm," Automatica Vol. 23, pp. 137-148.

[CMT87b] Clarke, D. W., Mohtadi, C., and Tuffs, P. S., 1987. "Generalized Predictive Control - Part II," Automatica Vol. 23, pp. 149-160.

[CRV06] Cogill, R., Rotkowitz, M., Van Roy, B., and Lall, S., 2006. "An Approximate Dynamic Programming Approach to Decentralized Control of Stochastic Systems," in Control of Uncertain Systems: Modelling, Approximation, and Design, Springer, Berlin, pp. 243-256.

[CXL19] Chu, Z., Xu, Z., and Li, H., 2019. "New Heuristics for the RCPSP with Multiple Overlapping Modes," Computers and Industrial Engineering, Vol. 131, pp. 146-156.

[CaB07] Camacho, E. F., and Bordons, C., 2007. Model Predictive Control, 2nd Edition, Springer, New York, N. Y.

[Can16] Candy, J. V., 2016. Bayesian Signal Processing: Classical, Modern, and Particle Filtering Methods, Wiley-IEEE Press.

[Cao07] Cao, X. R., 2007. Stochastic Learning and Optimization: A Sensitivity-Based Approach, Springer, N. Y.

[ChC17] Chui, C. K., and Chen, G., 2017. Kalman Filtering, Springer International Publishing.

[Che72] Chernoff, H., 1972. "Sequential Analysis and Optimal Design," Regional Conference Series in Applied Mathematics, SIAM, Philadelphia, PA.

[Chr97] Christodouleas, J. D., 1997. "Solution Methods for Multiprocessor Network Scheduling Problems with Application to Railroad Operations," Ph.D. Thesis, Operations Research Center, Massachusetts Institute of Technology.

[Cou06] Coulom, R., 2006. "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," International Conference on Computers and Games, Springer, pp. 72-83.

[CrS00] Cristianini, N., and Shawe-Taylor, J., 2000. An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods, Cambridge Univ. Press.

[Cyb89] Cybenko, 1989. "Approximation by Superpositions of a Sigmoidal Function," Math. of Control, Signals, and Systems, Vol. 2, pp. 303-314.

[DDF19] Daubechies, I., DeVore, R., Foucart, S., Hanin, B., and Petrova, G., 2019. "Nonlinear Approximation and (Deep) ReLU Networks," arXiv:1905.02199.

[DFM12] Desai, V. V., Farias, V. F., and Moallemi, C. C., 2012. "Aproximate Dynamic Programming via a Smoothed Approximate Linear Program," Operations Research, Vol. 60, pp. 655-674.

[DFM13] Desai, V. V., Farias, V. F., and Moallemi, C. C., 2013. "Bounds for Markov Decision Processes," in Reinforcement Learning and Approximate Dynamic Programming for Feedback Control, by F. Lewis and D. Liu (eds.), IEEE Press, Piscataway, N. J., pp. 452-473.

[DFV03] de Farias, D. P., and Van Roy, B., 2003. "The Linear Programming Approach to Approximate Dynamic Programming," Operations Research, Vol. 51, pp. 850-865.

[DFV04] de Farias, D. P., and Van Roy, B., 2004. "On Constraint Sampling in the Linear Programming Approach to Approximate Dynamic Programming," Mathematics of Operations Research, Vol. 29, pp. 462-478.

[DHS12] Duda, R. O., Hart, P. E., and Stork, D. G., 2012. Pattern Classification, J. Wiley, N. Y.

[DNW16] David, O. E., Netanyahu, N. S., and Wolf, L., 2016. "Deepchess: End-to-End Deep Neural Network for Automatic Learning in Chess," in International Conference on Artificial Neural Networks, pp. 88-96.

[DeF04] De Farias, D. P., 2004. "The Linear Programming Approach to Approximate Dynamic Programming," in Learning and Approximate Dynamic Programming, by J. Si, A. Barto, W. Powell, and D. Wunsch, (Eds.), IEEE Press, N. Y.

[DeG70] DeGroot, M. H., 1970. Optimal Statistical Decisions, McGraw-Hill, N. Y.

[DeK11] Devlin, S., and Kudenko, D., 2011. "Theoretical Considerations of Potential-Based Reward Shaping for Multi-Agent Systems," in Proceedings of AAMAS.

[Den67] Denardo, E. V., 1967. "Contraction Mappings in the Theory Underlying Dynamic Programming," SIAM Review, Vol. 9, pp. 165-177.

[DiL08] Dimitrakakis, C., and Lagoudakis, M. G., 2008. "Rollout Sampling Approximate Policy Iteration," Machine Learning, Vol. 72, pp. 157-171.

[DiM10] Di Castro, D., and Mannor, S., 2010. "Adaptive Bases for Reinforcement Learning," Machine Learning and Knowledge Discovery in Databases, Vol. 6321, pp. 312-327.

[DiW02] Dietterich, T. G., and Wang, X., 2002. "Batch Value Function Approximation via Support Vectors," in Advances in Neural Information Processing Systems, pp. 1491-1498.

[DoJ09] Doucet, A., and Johansen, A. M., 2009. "A Tutorial on Particle Filtering and Smoothing: Fifteen Years Later," Handbook of Nonlinear Filtering, Oxford University Press, Vol. 12, p. 3.

[DrH01] Drezner, Z., and Hamacher, H. W. eds., 2001. Facility Location: Applications and Theory, Springer Science and Business Media.

[Dre65] Dreyfus, S. D., 1965. Dynamic Programming and the Calculus of Variations, Academic Press, N. Y.

[DuV99] Duin, C., and Voss, S., 1999. "The Pilot Method: A Strategy for Heuristic Repetition with Application to the Steiner Problem in Graphs," Networks: An International Journal, Vol. 34, pp. 181-191.

[EDS18] Efroni, Y., Dalal, G., Scherrer, B., and Mannor, S., 2018. "Beyond the One-Step Greedy Approach in Reinforcement Learning," in Proc. International Conf. on Machine Learning, pp. 1387-1396.

[EMM05] Engel, Y., Mannor, S., and Meir, R., 2005. "Reinforcement Learning with Gaussian Processes," in Proc. of the 22nd ICML, pp. 201-208.

[FHS09] Feitzinger, F., Hylla, T., and Sachs, E. W., 2009. "Inexact Kleinman?Newton Method for Riccati Equations," SIAM Journal on Matrix Analysis and Applications, Vol. 3, pp. 272-288.

[FIA03] Findeisen, R., Imsland, L., Allgower, F., and Foss, B.A., 2003. "State and Output Feedback Nonlinear Model Predictive Control: An Overview," European Journal of Control, Vol. 9, pp. 190-206.

[FPB15] Farahmand, A. M., Precup, D., Barreto, A. M., and Ghavamzadeh, M., 2015. "Classification-Based Approximate Policy Iteration," IEEE Trans. on Automatic Control, Vol. 60, pp. 2989-2993.

[FeV02] Ferris, M. C., and Voelker, M. M., 2002. "Neuro-Dynamic Programming for Radiation Treatment Planning," Numerical Analysis Group Research Report NA-02/06, Oxford University Computing Laboratory, Oxford University.

[FeV04] Ferris, M. C., and Voelker, M. M., 2004. "Fractionation in Radiation Treatment Planning," Mathematical Programming B, Vol. 102, pp. 387-413.

[Fel60] Feldbaum, A. A., 1960. "Dual Control Theory," Automation and Remote Control, Vol. 21, pp. 874-1039.

[FiT91] Filar, J. A., and Tolwinski, B., 1991. "On the Algorithm of Pollatschek and Avi-ltzhak," in Stochastic Games and Related Topics, Theory and Decision Library, Springer, Vol. 7, pp. 59-70.

[FiV96] Filar, J., and Vrieze, K., 1996. Competitive Markov Decision Processes, Springer.

[FoK09] Forrester, A. I., and Keane, A. J., 2009. "Recent Advances in Surrogate-Based Optimization. Progress in Aerospace Sciences," Vol. 45, pp. 50-79.

[Fra18] Frazier, P. I., 2018. "A Tutorial on Bayesian Optimization," arXiv:1807.02811.

[Fu17] Fu, M. C., 2017. "Markov Decision Processes, AlphaGo, and Monte Carlo Tree Search: Back to the Future," Leading Developments from INFORMS Communities, INFORMS, pp. 68-88.

[Fun89] Funahashi, K., 1989. "On the Approximate Realization of Continuous Mappings by Neural Networks," Neural Networks, Vol. 2, pp. 183-192.

[GBC16] Goodfellow, I., Bengio, J., and Courville, A., Deep Learning, MIT Press, Cambridge, MA.

[GBL19] Goodson, J. C., Bertazzi, L., and Levary, R. R., 2019. "Robust Dynamic Media Selection with Yield Uncertainty: Max-Min Policies and Dual Bounds," Report.

[GDM19] Guerriero, F., Di Puglia Pugliese, L., and Macrina, G., 2019. "A Rollout Algorithm for the Resource Constrained Elementary Shortest Path Problem," Optimization Methods and Software, Vol. 34, pp. 1056-1074.

[GGS13] Gabillon, V., Ghavamzadeh, M., and Scherrer, B., 2013. "Approximate Dynamic Programming Finally Performs Well in the Game of Tetris," in NIPS, pp. 1754-1762.

[GGW11] Gittins, J., Glazebrook, K., and Weber, R., 2011. Multi-Armed Bandit Allocation Indices, J. Wiley, N. Y.

[GHC21] Gerlach, T., Hoffmann, F., and Charlish, A., 2021. "Policy Rollout Action Selection with Knowledge Gradient for Sensor Path Planning," 2021 IEEE 24th International Conference on Information Fusion, pp. 1-8.

[GLG11] Gabillon, V., Lazaric, A., Ghavamzadeh, M., and Scherrer, B., 2011. "Classification-Based Policy Iteration with a Critic," in Proc. of ICML.

[GMP15] Ghavamzadeh, M., Mannor, S., Pineau, J., and Tamar, A., 2015. "Bayesian Reinforcement Learning: A Survey," Foundations and Trends in Machine Learning, Vol. 8, pp. 359-483.

[GSD06] Goodwin, G., Seron, M. M., and De Dona, J. A., 2006. Constrained Control and Estimation: An Optimisation Approach, Springer, N. Y.

[GSS93] Gordon, N. J., Salmond, D. J., and Smith, A. F., 1993. "Novel Approach to Nonlinear/Non-Gaussian Bayesian State Estimation," in IEE Proceedings, Vol. 140, pp. 107-113.

[GTA17] Gommans, T. M. P., Theunisse, T. A. F., Antunes, D. J., and Heemels, W. P. M. H., 2017. "Resource-Aware MPC for Constrained Linear Systems: Two Rollout Approaches," Journal of Process Control, Vol. 51, pp. 68-83.

[GTO15] Goodson, J. C., Thomas, B. W., and Ohlmann, J. W., 2015. "Restocking-Based Rollout Policies for the Vehicle Routing Problem with Stochastic Demand and Duration Limits," Transportation Science, Vol. 50, pp. 591-607.

[GTO17] Goodson, J. C., Thomas, B. W., and Ohlmann, J. W., 2017. "A Rollout Algorithm Framework for Heuristic Solutions to Finite-Horizon Stochastic Dynamic Programs," European Journal of Operational Research, Vol. 258, pp. 216-229.

[GeB13] Geffner, H., and Bonet, B., 2013. A Concise Introduction to Models and Methods for Automated Planning, Morgan and Claypool Publishers.

[GoS84] Goodwin, G. C., and Sin, K. S. S., 1984. Adaptive Filtering, Prediction, and Control, Prentice-Hall, Englewood Cliffs, N. J.

[Gos15] Gosavi, A., 2015. Simulation-Based Optimization: Parametric Optimization Techniques and Reinforcement Learning, 2nd Edition, Springer, N. Y.

[Grz17] Grzes, M., 2017. "Reward Shaping in Episodic Reinforcement Learning," in Proc. of the 16th Conference on Autonomous Agents and MultiAgent Systems, pp. 565-573.

[GuM01] Guerriero, F., and Musmanno, R., 2001. "Label Correcting Methods to Solve Multicriteria Shortest Path Problems," J. Optimization Theory Appl., Vol. 111, pp. 589-613.

[GuM03] Guerriero, F., and Mancini, M., 2003. "A Cooperative Parallel Rollout Algorithm for the Sequential Ordering Problem," Parallel Computing, Vol. 29, pp. 663-677.

[Gup20] Gupta, A., 2020. "Existence of Team-Optimal Solutions in Static Teams with Common Information: A Topology of Information Approach," SIAM J. on Control and Optimization, Vol. 58, pp.998-1021.

[HCR21] Hoffmann, F., Charlish, A., Ritchie, M., and Griffiths, H., 2021. "Policy Rollout Action Selection in Continuous Domains for Sensor Path Planning," IEEE Trans. on Aerospace and Electronic Systems.

[HJG16] Huang, Q., Jia, Q. S., and Guan, X., 2016. "Robust Scheduling of EV Charging Load with Uncertain Wind Power Integration," IEEE Trans. on Smart Grid, Vol. 9, pp. 1043-1054.

[HLS06] Han, J., Lai, T. L. and Spivakovsky, V., 2006. "Approximate Policy Optimization and Adaptive Control in Regression Models," Computational Economics, Vol. 27, pp. 433-452.

[HMR19] Hastie, T., Montanari, A., Rosset, S., and Tibshirani, R. J., 2019. "Surprises in High-Dimensional Ridgeless Least Squares Interpolation," arXiv:1903.08560.

[HLZ19] Ho, T. Y., Liu, S., and Zabinsky, Z. B., 2019. "A Multi-Fidelity Rollout Algorithm for Dynamic Resource Allocation in Population Disease Management," Health Care Management Science, Vol. 22, pp. 727-755.

[HSS08] Hofmann, T., Scholkopf, B., and Smola, A. J., 2008. "Kernel Methods in Machine Learning," The Annals of Statistics, Vol. 36, pp. 1171-1220.

[HSW89] Hornick, K., Stinchcombe, M., and White, H., 1989. "Multilayer Feedforward Networks are Universal Approximators," Neural Networks, Vol. 2, pp. 359-159.

[HWM19] Hewing, L., Wabersich, K. P., Menner, M., and Zeilinger, M. N., 2019. "Learning-Based Model Predictive Control: Toward Safe Learning in Control," Annual Review of Control, Robotics, and Autonomous Systems.

[HaR21] Hardt, M., and Recht, B., 2021. Patterns, Predictions, and Actions: A Story About Machine Learning, arXiv:2102.05242; published by Priceton Univ. Press, 2022.

[Han98] Hansen, E. A., 1998. "Solving POMDPs by Searching in Policy Space," in Proc. of the 14th Conf. on Uncertainty in Artificial Intelligence, pp. 211-219.

[Hay08] Haykin, S., 2008. Neural Networks and Learning Machines, 3rd Edition, Prentice-Hall, Englewood-Cliffs, N. J.

[HeZ19] Hewing, L., and Zeilinger, M. N., 2019. "Scenario-Based Probabilistic Reachable Sets for Recursively Feasible Stochastic Model Predictive Control," IEEE Control Systems Letters, Vol. 4, pp. 450-455.

[Hew71] Hewer, G., 1971. "An Iterative Technique for the Computation of the Steady State Gains for the Discrete Optimal Regulator," IEEE Trans. on Automatic Control, Vol. 16, pp. 382-384.

[Ho80] Ho, Y. C., 1980. "Team Decision Theory and Information Structures," Proceedings of the IEEE, Vol. 68, pp. 644-654.

[HuM16] Huan, X., and Marzouk, Y. M., 2016. "Sequential Bayesian Optimal Experimental Design via Approximate Dynamic Programming," arXiv:1604.08320.

[Hua15] Huan, X., 2015. Numerical Approaches for Sequential Bayesian Optimal Experimental Design, Ph.D. Thesis, MIT.

[Hyl11] Hylla, T., 2011. Extension of Inexact Kleinman-Newton Methods to a General Monotonicity Preserving Convergence Theory, PhD Thesis, Univ. of Trier.

[IFT19] Issakkimuthu, M., Fern, A., and Tadepalli, P., 2019. "The Choice Function Framework for Online Policy Improvement," arXiv:1910.00614.

[IJT18] Iusem, Jofre, A., and Thompson, P., 2018. "Incremental Constraint Projection Methods for Monotone Stochastic Variational Inequalities," Math. of Operations Research, Vol. 44, pp. 236-263.

[IoS96] Ioannou, P. A., and Sun, J., 1996. Robust Adaptive Control, Prentice-Hall, Englewood Cliffs, N. J.

[JCG20] Jiang, S., Chai, H., Gonzalez, J., and Garnett, R., 2020. "BINOCULARS for Efficient, Nonmyopic Sequential Experimental Design," in Proc. Intern. Conference on Machine Learning, pp. 4794-4803.

[JGJ18] Jones, M., Goldstein, M., Jonathan, P., and Randell, D., 2018. "Bayes Linear Analysis of Risks in Sequential Optimal Design Problems," Electronic Journal of Statistics, Vol. 12, pp. 4002-4031.

[JJB20] Jiang, S., Jiang, D. R., Balandat, M., Karrer, B., Gardner, J. R., and Garnett, R., 2020. "Efficient Nonmyopic Bayesian Optimization via One-Shot Multi-Step Trees," arXiv preprint arXiv:2006.15779.

[JSW98] Jones, D. R., Schonlau, M., and Welch, W. J., 1998. "Efficient Global Optimization of Expensive Black-Box Functions," J. of Global Optimization, Vol. 13, pp. 455?492.

[JiJ17] Jiang, Y., and Jiang, Z. P., 2017. Robust Adaptive Dynamic Programming, J. Wiley, N. Y.

[Jon90] Jones, L. K., 1990. "Constructive Approximations for Neural Networks by Sigmoidal Functions," Proceedings of the IEEE, Vol. 78, pp. 1586-1589.

[JuP07] Jung, T., and Polani, D., 2007. "Kernelizing LSPE(λ)," Proc. 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning, Honolulu, Ha., pp. 338-345.

[KAC15] Kochenderfer, M. J., with Amato, C., Chowdhary, G., How, J. P., Davison Reynolds, H. J., Thornton, J. R., Torres-Carrasquillo, P. A., Ore, N. K., Vian, J., 2015. Decision Making under Uncertainty: Theory and Application, MIT Press, Cambridge, MA.

[KAH15] Khashooei, B. A., Antunes, D. J. and Heemels, W.P.M.H., 2015. "Rollout Strategies for Output-Based Event-Triggered Control," in Proc. 2015 European Control Conference, pp. 2168-2173.

[KKK95] Krstic, M., Kanellakopoulos, I., Kokotovic, P., 1995. Nonlinear and Adaptive Control Design, J. Wiley, N. Y.

[KLC98] Kaelbling, L. P., Littman, M. L., and Cassandra, A. R., 1998. "Planning and Acting in Partially Observable Stochastic Domains," Artificial Intelligence, Vol. 101, pp. 99-134.

[KLM82a] Krainak, J. L. S. J. C., Speyer, J., and Marcus, S., 1982. "Static Team Problems - Part I: Sufficient Conditions and the Exponential Cost Criterion," IEEE Transactions on Automatic Control, Vol. 27, pp. 839-848.

[KLM82b] Krainak, J. L. S. J. C., Speyer, J., and Marcus, S., 1982. "Static Team Problems - Part II: Affine Control Laws, Projections, Algorithms, and the LEGT Problem," IEEE Transactions on Automatic Control, Vol. 27, pp. 848-859.

[KLM96] Kaelbling, L. P., Littman, M. L., and Moore, A. W., 1996. "Reinforcement Learning: A Survey," J. of Artificial Intelligence Res., Vol. 4, pp. 237-285.

[KMP06] Keller, P. W., Mannor, S., and Precup, D., 2006. "Automatic Basis Function Construction for Approximate Dynamic Programming and Reinforcement Learning," Proc. of the 23rd ICML, Pittsburgh, Penn.

[KaW94] Kall, P., and Wallace, S. W., 1994. Stochastic Programming, Wiley, Chichester, UK.

[KeG88] Keerthi, S. S., and Gilbert, E. G., 1988. "Optimal, Infinite Horizon Feedback Laws for a General Class of Constrained Discrete Time Systems: Stability and Moving-Horizon Approximations," J. Optimization Theory Appl., Vo. 57, pp. 265-293.

[Kir04] Kirk, D. E., 2004. Optimal Control Theory: An Introduction, Courier Corporation.

[Kle09] Kleijnen, J. P., 2009. "Kriging Metamodeling in Simulation: A Review," European Journal of Operational Research, Vol. 192, pp. 707-716.

[Kle68] Kleinman, D. L., 1968. "On an Iterative Technique for Riccati Equation Computations," IEEE Trans. Aut. Control, Vol. AC-13, pp. 114-115.

[KoC16] Kouvaritakis, B., and Cannon, M., 2016. Model Predictive Control: Classical, Robust and Stochastic, Springer, N. Y.

[KoG98] Kolmanovsky, I.. and Gilbert, E. G., 1998. "Theory and Computation of Disturbance Invariant Sets for Discrete-Time Linear Systems," Math. Problems in Engineering, Vol. 4, pp. 317-367.

[KoS06] Kocsis, L., and Szepesvari, C., 2006. "Bandit Based Monte-Carlo Planning," Proc. of 17th European Conference on Machine Learning, Berlin, pp. 282-293.

[Kre19] Krener, A. J., 2019. "Adaptive Horizon Model Predictive Control and Al'brekht's Method," arXiv:1904.00053.

[Kri16] Krishnamurthy, V., 2016. Partially Observed Markov Decision Processes, Cambridge Univ. Press.

[KuV86] Kumar, P. R., and Varaiya, P. P., 1986. Stochastic Systems: Estimation, Identification, and Adaptive Control, Prentice-Hall, Englewood Cliffs, N. J.

[Kun14] Kung, S. Y., 2014. Kernel Methods and Machine Learning, Cambridge Univ. Press.

[LEC20] Lee, E. H., Eriksson, D., Cheng, B., McCourt, M., and Bindel, D., 2020. "Efficient Rollout Strategies for Bayesian Optimization," arXiv:2002.10539.

[LEP21] Lee, E. H., Eriksson, D., Perrone, V., and Seeger, M., 2021. "A Nonmyopic Approach to Cost-Constrained Bayesian Optimization," In Uncertainty in Artificial Intelligence Proceedings, pp. 568-577.

[LGM10] Lazaric, A., Ghavamzadeh, M., and Munos, R., 2010. "Analysis of a Classification-Based Policy Iteration Algorithm," INRIA Report.

[LGW16] Lan, Y., Guan, X., and Wu, J., 2016. "Rollout Strategies for Real-Time Multi-Energy Scheduling in Microgrid with Storage System," IET Generation, Transmission and Distribution, Vol. 10, pp. 688-696.

[LJM19] Li, Y., Johansson, K. H., and Martensson, J., 2019. "Lambda-Policy Iteration with Randomization for Contractive Models with Infinite Policies: Well Posedness and Convergence," arXiv:1912.08504.

[LJM21] Li, Y., Johansson, K. H., Martensson, J., and Bertsekas, D. P., 2021. "Data-Driven Rollout for Deterministic Optimal Control," arXiv preprint arXiv:2105.03116.

[LKG21] Li, T., Krakow, L. W., and Gopalswamy, S., 2021. "Optimizing Consensus-Based Multi-Target Tracking with Multiagent Rollout Control Policies," In 2021 IEEE Conference on Control Technology and Applications, pp. 131-137.

[LLL19] Liu, Z., Lu, J., Liu, Z., Liao, G., Zhang, H. H., and Dong, J., 2019. "Patient Scheduling in Hemodialysis Service," J. of Combinatorial Optimization, Vol. 37, pp. 337-362.

[LLP93] Leshno, M., Lin, V. Y., Pinkus, A., and Schocken, S., 1993. "Multilayer Feedforward Networks with a Nonpolynomial Activation Function can Approximate any Function," Neural Networks, Vol. 6, pp. 861-867.

[LPS22] Liu, M., Pedrielli, G., Sulc, P., Poppleton, E., Bertsekas, D. P., 2022. "ExpertRNA: A New Framework for RNA Structure Prediction," INFORMS Journal on Computing.

[LTZ19] Li, Y., Tang, Y., Zhang, R., and Li, N., 2019. "Distributed Reinforcement Learn-

ing for Decentralized Linear Quadratic Control: A Derivative-Free Policy Optimization Approach," arXiv:1912.09135.

[LWT17] Lowe, L., Wu, Y., Tamar, A., Harb, J., Abbeel, P., Mordatch, I., 2017. "Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments," in Advances in Neural Information Processing Systems, pp. 6379-6390.

[LWW16] Lam, R., Willcox, K., and Wolpert, D. H., 2016. "Bayesian Optimization with a Finite Budget: An Approximate Dynamic Programming Approach," In Advances in Neural Information Processing Systems, pp. 883-891.

[LWW17] Liu, D., Wei, Q., Wang, D., Yang, X., and Li, H., 2017. Adaptive Dynamic Programming with Applications in Optimal Control, Springer, Berlin.

[LZS20] Li, H., Zhang, X., Sun, J., and Dong, X., 2020. "Dynamic Resource Levelling in Projects under Uncertainty," International J. of Production Research.

[LaP03] Lagoudakis, M. G., and Parr, R., 2003. "Reinforcement Learning as Classification: Leveraging Modern Classifiers," in Proc. of ICML, pp. 424-431.

[LaR85] Lai, T., and Robbins, H., 1985. "Asymptotically Efficient Adaptive Allocation Rules," Advances in Applied Math., Vol. 6, pp. 4-22.

[LaS20] Lattimore, T., and Szepesvri, C., 2020. Bandit Algorithms, Cambridge University Press.

[LaW13] Lavretsky, E., and Wise, K., 2013. Robust and Adaptive Control with Aerospace Applications, Springer.

[LaW17] Lam, R., and Willcox, K., 2017. "Lookahead Bayesian Optimization with Inequality Constraints," in Advances in Neural Information Processing Systems, pp. 1890-1900.

[Lee20] Lee, E. H., 2020. "Budget-Constrained Bayesian Optimization, Doctoral dissertation, Cornell University.

[LiS16] Liang, S., and Srikant, R., 2016. "Why Deep Neural Networks for Function Approximation?" arXiv:1610.04161.

[LiW14] Liu, D., and Wei, Q., 2014. "Policy Iteration Adaptive Dynamic Programming Algorithm for Discrete-Time Nonlinear Systems," IEEE Trans. on Neural Networks and Learning Systems, Vol. 25, pp. 621-634.

[LiW15] Li, H., and Womer, N. K., 2015. "Solving Stochastic Resource-Constrained Project Scheduling Problems by Closed-Loop Approximate Dynamic Programming," European J. of Operational Research, Vol. 246, pp. 20-33.

[Lib11] Liberzon, D., 2011. Calculus of Variations and Optimal Control Theory: A Concise Introduction, Princeton Univ. Press.

[LoC23] Loxley, P. N., and Cheung, K. W., 2023. "A Dynamic Programming Algorithm for Finding an Optimal Sequence of Informative Measurements," Entropy, Vol. 25, p. 251.

[MCT10] Mishra, N., Choudhary, A. K., Tiwari, M. K., and Shankar, R., 2010. "Rollout Strategy-Based Probabilistic Causal Model Approach for the Multiple Fault Diagnosis," Robotics and Computer-Integrated Manufacturing, Vol. 26, pp. 325-332.

[MKS15] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., and Petersen, S., 2015. "Human-Level Control Through Deep Reinforcement Learning," Nature, Vol. 518, p. 529.

[MLM20] Montenegro, M., Lopez, R., Menchaca-Mendez, R., Becerra, E., and Menchaca-Mendez, R., 2020. "A Parallel Rollout Algorithm for Wildfire Suppression," in Proc. Intern. Congress of Telematics and Computing, pp. 244-255.

[MMB02] McGovern, A., Moss, E., and Barto, A., 2002. "Building a Basic Building Block Scheduler Using Reinforcement Learning and Rollouts," Machine Learning, Vol. 49, pp. 141-160.

[MMS05] Menache, I., Mannor, S., and Shimkin, N., 2005. "Basis Function Adaptation in Temporal Difference Reinforcement Learning," Ann. Oper. Res., Vol. 134, pp. 215-238.

[MPK99] Meuleau, N., Peshkin, L., Kim, K. E., and Kaelbling, L. P., 1999. "Learning Finite-State Controllers for Partially Observable Environments," in Proc. of the 15th Conference on Uncertainty in Artificial Intelligence, pp. 427-436.

[MPP04] Meloni, C., Pacciarelli, D., and Pranzo, M., 2004. "A Rollout Metaheuristic for Job Shop Scheduling Problems," Annals of Operations Research, Vol. 131, pp. 215-235.

[MRR00] Mayne, D., Rawlings, J. B., Rao, C. V., and Scokaert, P. O. M., 2000. "Constrained Model Predictive Control: Stability and Optimality," Automatica, Vol. 36, pp. 789-814.

[MVS19] Muthukumar, V., Vodrahalli, K., and Sahai, A., 2019. "Harmless Interpolation of Noisy Data in Regression," arXiv:1903.09139.

[MYF03] Moriyama, H., Yamashita, N., and Fukushima, M., 2003. "The Incremental Gauss-Newton Algorithm with Adaptive Stepsize Rule," Computational Optimization and Applications, Vol. 26, pp. 107-141.

[MaJ15] Mastin, A., and Jaillet, P., 2015. "Average-Case Performance of Rollout Algorithms for Knapsack Problems," J. of Optimization Theory and Applications, Vol. 165, pp. 964-984.

[MaS02] Martinez, L., and Soares, S., 2002. Comparison Between Closed-Loop and Partial Open-Loop Feedback Control Policies in Long Term Hydrothermal Scheduling," IEEE Transactions on Power Systems, Vol. 17, pp. 330-336.

[Mac02] Maciejowski, J. M., 2002. Predictive Control with Constraints, Addison-Wesley, Reading, MA.

[Mar55] Marschak, J., 1975. "Elements for a Theory of Teams," Management Science, Vol. 1, pp. 127-137.

[Mar84] Martins, E. Q. V., 1984. "On a Multicriteria Shortest Path Problem," European J. of Operational Research, Vol. 16, pp. 236-245.

[May14] Mayne, D. Q., 2014. "Model Predictive Control: Recent Developments and Future Promise," Automatica, Vol. 50, pp. 2967-2986.

[MeB99] Meuleau, N., and Bourgine, P., 1999. "Exploration of Multi-State Environments: Local Measures and Back-Propagation of Uncertainty," Machine Learning, Vol. 35, pp. 117-154.

[MeK20] Meshram, R.. and Kaza, K., 2020. "Simulation Based Algorithms for Markov Decision Processes and Multi-Action Restless Bandits," arXiv:2007.12933.

[Mey07] Meyn, S., 2007. Control Techniques for Complex Networks, Cambridge Univ. Press, N. Y.

[Mey22] Meyn, S., 2022. Control Systems and Reinforcement Learning, Cambridge Univ. Press, N. Y.

[Min22] Minorsky, N., 1922. "Directional Stability of Automatically Steered Bodies," J. Amer. Soc. Naval Eng.,Vol. 34, pp. 280-309.

[MoL99] Morari, M., and Lee, J. H., 1999. "Model Predictive Control: Past, Present, and Future," Computers and Chemical Engineering, Vol. 23, pp. 667-682.

[Mon17] Montgomery, D. C., 2017. Design and Analysis of Experiments, J. Wiley.

[Mun14] Munos, R., 2014. "From Bandits to Monte-Carlo Tree Search: The Optimistic Principle Applied to Optimization and Planning," Foundations and Trends in Machine Learning, Vol. 7, pp. 1-129.

[NHR99] Ng, A. Y., Harada, D., and Russell, S. J., 1999. "Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping," in Proc. of the 16th International Conference on Machine Learning, pp. 278-287.

[NMT13] Nayyar, A., Mahajan, A. and Teneketzis, D., 2013. "Decentralized Stochastic Control with Partial History Sharing: A Common Information Approach," IEEE Transactions on Automatic Control, Vol. 58, pp. 1644-1658.

[NSE19] Nozhati, S., Sarkale, Y., Ellingwood, B., Chong, E. K., and Mahmoud, H., 2019. "Near-Optimal Planning Using Approximate Dynamic Programming to Enhance Post-Hazard Community Resilience Management," Reliability Engineering and System Safety, Vol. 181, pp. 116-126.

[NaA12] Narendra, K. S., and Annaswamy, A. M., 2012. Stable Adaptive Systems, Courier Corporation.

[NaT19] Nayyar, A., and Teneketzis, D., 2019. "Common Knowledge and Sequential Team Problems," IEEE Trans. on Automatic Control, Vol. 64, pp. 5108-5115.

[Ned11] Nedić, A., 2011. "Random Algorithms for Convex Minimization Problems," Math. Programming, Ser. B, Vol. 129, pp. 225-253.

[OrS02] Ormoneit, D., and Sen, S., 2002. "Kernel-Based Reinforcement Learning," Machine Learning, Vol. 49, pp. 161-178.

[PDB92] Pattipati, K. R., Deb, S., Bar-Shalom, Y., and Washburn, R. B., 1992. "A New Relaxation Algorithm and Passive Sensor Data Association," IEEE Trans. Automatic Control, Vol. 37, pp. 198-213.

[PDC14] Pillonetto, G., Dinuzzo, F., Chen, T., De Nicolao, G., and Ljung, L., 2014. "Kernel Methods in System Identification, Machine Learning and Function Estimation: A Survey," Automatica, Vol. 50, pp. 657-682.

[PPB01] Popp, R. L., Pattipati, K. R., and Bar-Shalom, Y., 2001. "m-Best SD Assignment Algorithm with Application to Multitarget Tracking," IEEE Transactions on Aerospace and Electronic Systems, Vol. 37, pp. 22-39.

[PSC22] Paulson, J. A., Sonouifar, F., and Chakrabarty, A., 2022. "Efficient Multi-Step Lookahead Bayesian Optimization with Local Search Constraints," IEEE Conf. on Decision and Control, pp. 123-129.

[PPG16] Perolat, J., Piot, B., Geist, M., Scherrer, B., and Pietquin, O., 2016. "Softened Approximate Policy Iteration for Markov Games," in Proc. International Conference on Machine Learning, pp. 1860-1868.

[PSP15] Perolat, J., Scherrer, B., Piot, B., and Pietquin, O., 2015. "Approximate Dynamic Programming for Two-Player Zero-Sum Markov Games," in Proc. International Conference on Machine Learning, pp. 1321-1329.

[PaB99] Patek, S. D., and Bertsekas, D. P., 1999. "Stochastic Shortest Path Games," SIAM J. on Control and Optimization, Vol. 37, pp. 804-824.

[PaR12] Papahristou, N., and Refanidis, I., 2012. "On the Design and Training of Bots to Play Backgammon Variants," in IFIP International Conference on Artificial Intelligence Applications and Innovations, pp. 78-87.

[PaT00] Paschalidis, I. C., and Tsitsiklis, J. N., 2000. "Congestion-Dependent Pricing of Network Services," IEEE/ACM Trans. on Networking, Vol. 8, pp. 171-184.

[PeG04] Peret, L., and Garcia, F., 2004. "On-Line Search for Solving Markov Decision Processes via Heuristic Sampling," in Proc. of the 16th European Conference on Artificial Intelligence, pp. 530-534.

[PeW96] Peng, J., and Williams, R., 1996. "Incremental Multi-Step Q-Learning," Machine Learning, Vol. 22, pp. 283-290.

[PoA69] Pollatschek, M. A. and Avi-Itzhak, B., 1969. "Algorithms for Stochastic Games with Geometrical Interpretation," Management Science, Vol. 15, pp. 399-415.

[PoB04] Poupart, P., and Boutilier, C., 2004. "Bounded Finite State Controllers," in Advances in Neural Information Processing Systems, pp. 823-830.

[PoF08] Powell, W. B. and Frazier, P., 2008. "Optimal Learning," in State-of-the-Art Decision-Making Tools in the Information-Intensive Age, INFORMS, pp. 213-246.

[PoR97] Poore, A. B., and Robertson, A. J. A., 1997. "New Lagrangian Relaxation Based Algorithm for a Class of Multidimensional Assignment Problems," Computational Optimization and Applications, Vol. 8, pp. 129-150.

[PoR12] Powell, W. B., and Ryzhov, I. O., 2012. Optimal Learning, J. Wiley, N. Y.

[Poo94] Poore, A. B., 1994. "Multidimensional Assignment Formulation of Data Association Problems Arising from Multitarget Tracking and Multisensor Data Fusion," Computational Optimization and Applications, Vol. 3, pp. 27-57.

[Pow11] Powell, W. B., 2011. Approximate Dynamic Programming: Solving the Curses of Dimensionality, 2nd Edition, J. Wiley and Sons, Hoboken, N. J.

[Pre95] Prekopa, A., 1995. Stochastic Programming, Kluwer, Boston.

[PuB78] Puterman, M. L., and Brumelle, S. L., 1978. "The Analytic Theory of Policy Iteration," in Dynamic Programming and Its Applications, M. L. Puterman (ed.), Academic Press, N. Y.

[PuB79] Puterman, M. L., and Brumelle, S. L., 1979. "On the Convergence of Policy Iteration in Stationary Dynamic Programming," Mathematics of Operations Research, Vol. 4, pp. 60-69.

[PuS78] Puterman, M. L., and Shin, M. C., 1978. "Modified Policy Iteration Algorithms for Discounted Markov Decision Problems," Management Sci., Vol. 24, pp. 1127-1137.

[PuS82] Puterman, M. L., and Shin, M. C., 1982. "Action Elimination Procedures for Modified Policy Iteration Algorithms," Operations Research, Vol. 30, pp. 301-318.

[Put94] Puterman, M. L., 1994. Markovian Decision Problems, J. Wiley, N. Y.

[QHS05] Queipo, N. V., Haftka, R. T., Shyy, W., Goel, T., Vaidyanathan, R., and Tucker, P. K., 2005. "Surrogate-Based Analysis and Optimization," Progress in Aerospace Sciences, Vol. 41, pp. 1-28.

[QuL19] Qu, G., and Li, N., "Exploiting Fast Decaying and Locality in Multi-Agent MDP with Tree Dependence Structure," Proc. of 2019 CDC, Nice, France.

[RCR17] Rudi, A., Carratino, L., and Rosasco, L., 2017. "Falkon: An Optimal Large Scale Kernel Method," in Advances in Neural Information Processing Systems, pp. 3888-3898.

[RGG21] Rimélé, A., Grangier, P., Gamache, M., Gendreau, M., and Rousseau, L. M., 2021. "E-Commerce Warehousing: Learning a Storage Policy, arXiv:2101.08828.

[RMD17] Rawlings, J. B., Mayne, D. Q., and Diehl, M. M., 2017. Model Predictive Control: Theory, Computation, and Design, 2nd Ed., Nob Hill Publishing.

[RPF12] Ryzhov, I. O., Powell, W. B., and Frazier, P. I., 2012. "The Knowledge Gradient Algorithm for a General Class of Online Learning Problems," Operations Research, Vol. 60, pp. 180-195.

[RSM08] Reisinger, J., Stone, P., and Miikkulainen, R., 2008. "Online Kernel Selection for Bayesian Reinforcement Learning," in Proc. of the 25th International Conference on Machine Learning, pp. 816-823.

[RST23] Rusmevichientong, P., Sumida, M., Topaloglu, H., and Bai, Y., 2023. "Revenue Management with Heterogeneous Resources: Unit Resource Capacities, Advance Bookings, and Itineraries over Time Intervals," Operations Research.

[RaF91] Raghavan, T. E. S., and Filar, J. A., 1991. "Algorithms for Stochastic Games - A Survey," Zeitschrift fur Operations Research, Vol. 35, pp. 437-472.

[RaR17] Rawlings, J. B., and Risbeck, M. J., 2017. "Model Predictive Control with Discrete Actuators: Theory and Application," Automatica, Vol. 78, pp. 258-265.

[RaW06] Rasmussen, C. E., and Williams, C. K., 2006. Gaussian Processes for Machine Learning, MIT Press, Cambridge, MA.

[Rad62] Radner, R., 1962. "Team Decision Problems," Ann. Math. Statist., Vol. 33, pp. 857-881.

[RoB17] Rosolia, U., and Borrelli, F., 2017. "Learning Model Predictive Control for Iterative Tasks. A Data-Driven Control Framework," IEEE Trans. on Automatic Control, Vol. 63, pp. 1883-1896.

[RoB19] Rosolia, U., and Borrelli, F., 2019. "Sample-Based Learning Model Predictive Control for Linear Uncertain Systems," 58th Conference on Decision and Control (CDC), pp. 2702-2707.

[Rob52] Robbins, H., 1952. "Some Aspects of the Sequential Design of Experiments," Bulletin of the American Mathematical Society, Vol. 58, pp. 527-535.

[Ros70] Ross, S. M., 1970. Applied Probability Models with Optimization Applications, Holden-Day, San Francisco, CA.

[Ros12] Ross, S. M., 2012. Simulation, 5th Edition, Academic Press, Orlando, Fla.

[Rot79] Rothblum, U. G., 1979. "Iterated Successive Approximation for Sequential Decision Processes," in Stochastic Control and Optimization, by J. W. B. van Overhagen and H. C. Tijms (eds), Vrije University, Amsterdam.

[RuK16] Rubinstein, R. Y., and Kroese, D. P., 2016. Simulation and the Monte Carlo Method, 3rd Edition, J. Wiley, N. Y.

[RuN94] Rummery, G. A., and Niranjan, M., 1994. "On-Line Q-Learning Using Connectionist Systems," University of Cambridge, England, Department of Engineering, TR-166.

[RuN16] Russell, S. J., and Norvig, P., 2016. Artificial Intelligence: A Modern Approach, Pearson Education Limited, Malaysia.

[RuS03] Ruszczynski, A., and Shapiro, A., 2003. "Stochastic Programming Models," in Handbooks in Operations Research and Management Science, Vol. 10, pp. 1-64.

[SGC02] Savagaonkar, U., Givan, R., and Chong, E. K. P., 2002. "Sampling Techniques for Zero-Sum, Discounted Markov Games," in Proc. 40th Allerton Conference on Communication, Control and Computing, Monticello, Ill.

[SGG15] Scherrer, B., Ghavamzadeh, M., Gabillon, V., Lesner, B., and Geist, M., 2015. "Approximate Modified Policy Iteration and its Application to the Game of Tetris," J. of Machine Learning Research, Vol. 16, pp. 1629-1676.

[SHB15] Simroth, A., Holfeld, D., and Brunsch, R., 2015. "Job Shop Production Planning under Uncertainty: A Monte Carlo Rollout Approach," Proc. of the International Scientific and Practical Conference, Vol. 3, pp. 175-179.

[SHM16] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., and Dieleman, S., 2016. "Mastering the Game of Go with Deep Neural Networks and Tree Search," Nature, Vol. 529, pp. 484-489.

[SHS17] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., and Lillicrap, T., 2017. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm," arXiv:1712.01815.

[SJL18] Soltanolkotabi, M., Javanmard, A., and Lee, J. D., 2018. "Theoretical Insights into the Optimization Landscape of Over-Parameterized Shallow Neural Networks," IEEE Trans. on Information Theory, Vol. 65, pp. 742-769.

[SLA12] Snoek, J., Larochelle, H., and Adams, R. P., 2012. "Practical Bayesian Optimization of Machine Learning Algorithms," in Advances in Neural Information Processing Systems, pp. 2951-2959.

[SLJ13] Sun, B., Luh, P. B., Jia, Q. S., Jiang, Z., Wang, F., and Song, C., 2013. "Building Energy Management: Integrated Control of Active and Passive Heating, Cooling, Lighting, Shading, and Ventilation Systems," IEEE Trans. on Automation Science and Engineering, Vol. 10, pp. 588-602.

[SNC18] Sarkale, Y., Nozhati, S., Chong, E. K., Ellingwood, B. R., and Mahmoud, H., 2018. "Solving Markov Decision Processes for Network-Level Post-Hazard Recovery via Simulation Optimization and Rollout," in 2018 IEEE 14th International Conference on Automation Science and Engineering, pp. 906-912.

[SSS17] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A. and Chen, Y., 2017. "Mastering the Game of Go Without Human Knowledge," Nature, Vol. 550, pp. 354-359.

[SSW16] Shahriari, B., Swersky, K., Wang, Z., Adams, R. P., and De Freitas, N., 2015. "Taking the Human Out of the Loop: A Review of Bayesian Optimization," Proc. of IEEE, Vol. 104, pp. 148-175.

[SWM89] Sacks, J., Welch, W. J., Mitchell, T. J., and Wynn, H. P., 1989. "Design and Analysis of Computer Experiments," Statistical Science, pp. 409-423.

[SYL17] Saldi, N., Yuksel, S., and Linder, T., 2017. "Finite Model Approximations for Partially Observed Markov Decision Processes with Discounted Cost," arXiv:1710.07009.

[SZL08] Sun, T., Zhao, Q., Lun, P., and Tomastik, R., 2008. "Optimization of Joint Replacement Policies for Multipart Systems by a Rollout Framework," IEEE Trans. on Automation Science and Engineering, Vol. 5, pp. 609-619.

[SaB11] Sastry, S., and Bodson, M., 2011. Adaptive Control: Stability, Convergence and Robustness, Courier Corporation.

[Sal21] Saldi, N., 2021. "Regularized Stochastic Team Problems," Systems and Control Letters, Vol. 149.

[Sas02] Sasena, M. J., 2002. Flexibility and Efficiency Enhancements for Constrained Global Design Optimization with Kriging Approximations, PhD Thesis, Univ. of Michigan.

[ScS02] Scholkopf, B., and Smola, A. J., 2002. Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond, MIT Press, Cambridge, MA.

[Sch13] Scherrer, B., 2013. "Performance Bounds for Lambda Policy Iteration and Application to the Game of Tetris," J. of Machine Learning Research, Vol. 14, pp. 1181-1227.

[Sco10] Scott, S. L., 2010. "A Modern Bayesian Look at the Multi-Armed Bandit," Applied Stochastic Models in Business and Industry, Vol. 26, pp. 639-658.

[Sec00] Secomandi, N., 2000. "Comparing Neuro-Dynamic Programming Algorithms for the Vehicle Routing Problem with Stochastic Demands," Computers and Operations Research, Vol. 27, pp. 1201-1225.

[Sec01] Secomandi, N., 2001. "A Rollout Policy for the Vehicle Routing Problem with Stochastic Demands," Operations Research, Vol. 49, pp. 796-802.

[Sec03] Secomandi, N., 2003. "Analysis of a Rollout Approach to Sequencing Problems with Stochastic Routing Applications," J. of Heuristics, Vol. 9, pp. 321-352.

[ShC04] Shawe-Taylor, J., and Cristianini, N., 2004. Kernel Methods for Pattern Analysis, Cambridge Univ. Press.

[Sha50] Shannon, C., 1950. "Programming a Digital Computer for Playing Chess," Phil. Mag., Vol. 41, pp. 356-375.

[Sha53] Shapley, L. S., 1953. "Stochastic Games," Proc. of the National Academy of Sciences, Vol. 39, pp. 1095-1100.

[SiK19] Singh, R., and Kumar, P. R., 2019. "Optimal Decentralized Dynamic Policies for Video Streaming over Wireless Channels," arXiv:1902.07418.

[SlL91] Slotine, J.-J. E., and Li, W., Applied Nonlinear Control, Prentice-Hall, Englewood Cliffs, N. J.

[StW91] Stewart, B. S., and White, C. C., 1991. "Multiobjective A^*," J. ACM, Vol. 38, pp. 775-814.

[Ste94] Stengel, R. F., 1994. Optimal Control and Estimation, Courier Corporation.

[SuB18] Sutton, R., and Barto, A. G., 2018. Reinforcement Learning, 2nd Edition, MIT Press, Cambridge, MA.

[SuY19] Su, L., and Yang, P., 2019. 'On Learning Over-Parameterized Neural Networks: A Functional Approximation Perspective," in Advances in Neural Information Processing Systems, pp. 2637-2646.

[Sun19] Sun, R., 2019. "Optimization for Deep Learning: Theory and Algorithms," arXiv:1912.08957.

[Sze10] Szepesvari, C., 2010. Algorithms for Reinforcement Learning, Morgan and Claypool Publishers, San Franscisco, CA.

[TBP21] Tuncel, Y., Bhat, G., Park, J., and Ogras, U., 2021. "ECO: Enabling Energy-Neutral IoT Devices through Runtime Allocation of Harvested Energy," arXiv:2102.13605.

[TCW19] Tseng, W. J., Chen, J. C., Wu, I. C., and Wei, T. H., 2019. "Comparison Training for Computer Chinese Chess," IEEE Trans. on Games, Vol. 12, pp. 169-176.

[TGL13] Tesauro, G., Gondek, D. C., Lenchner, J., Fan, J., and Prager, J. M., 2013. "Analysis of Watson's Strategies for Playing Jeopardy!," J. of Artificial Intelligence Research, Vol. 47, pp. 205-251.

[TRV16] Tu, S., Roelofs, R., Venkataraman, S., and Recht, B., 2016. "Large Scale Kernel Learning Using Block Coordinate Descent," arXiv:1602.05310.

[TaL20] Tanzanakis, A., and Lygeros, J., 2020. "Data-Driven Control of Unknown Systems: A Linear Programming Approach," arXiv:2003.00779.

[TeG96] Tesauro, G., and Galperin, G. R., 1996. "On-Line Policy Improvement Using Monte Carlo Search," NIPS, Denver, CO.

[Tes89a] Tesauro, G. J., 1989. "Neurogammon Wins Computer Olympiad," Neural Computation, Vol. 1, pp. 321-323.

[Tes89b] Tesauro, G. J., 1989. "Connectionist Learning of Expert Preferences by Comparison Training," in Advances in Neural Information Processing Systems, pp. 99-106.

[Tes92] Tesauro, G. J., 1992. "Practical Issues in Temporal Difference Learning," Machine Learning, Vol. 8, pp. 257-277.

[Tes94] Tesauro, G. J., 1994. "TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play," Neural Computation, Vol. 6, pp. 215-219.

[Tes95] Tesauro, G. J., 1995. "Temporal Difference Learning and TD-Gammon," Communications of the ACM, Vol. 38, pp. 58-68.

[Tes01] Tesauro, G. J., 2001. "Comparison Training of Chess Evaluation Functions," in Machines that Learn to Play Games, Nova Science Publishers, pp. 117-130.

[Tes02] Tesauro, G. J., 2002. "Programming Backgammon Using Self-Teaching Neural Nets," Artificial Intelligence, Vol. 134, pp. 181-199.

[ThS09] Thiery, C., and Scherrer, B., 2009. "Improvements on Learning Tetris with Cross-Entropy," International Computer Games Association J., Vol. 32, pp. 23-33.

[Tol89] Tolwinski, B., 1989. "Newton-Type Methods for Stochastic Games," in Basar T. S., and Bernhard P. (eds), Differential Games and Applications, Lecture Notes in Control and Information Sciences, vol. 119, Springer, pp. 128-144.

[TsV96] Tsitsiklis, J. N., and Van Roy, B., 1996. "Feature-Based Methods for Large-Scale Dynamic Programming," Machine Learning, Vol. 22, pp. 59-94.

[Tse98] Tseng, P., 1998. "Incremental Gradient(-Projection) Method with Momentum Term and Adaptive Stepsize Rule," SIAM J. on Optimization, Vol. 8, pp. 506-531.

[TuP03] Tu, F., and Pattipati, K. R., 2003. "Rollout Strategies for Sequential Fault Diagnosis," IEEE Trans. on Systems, Man and Cybernetics, Part A, pp. 86-99.

[UGM18] Ulmer, M. W., Goodson, J. C., Mattfeld, D. C., and Hennig, M., 2018. "Offline-Online Approximate Dynamic Programming for Dynamic Vehicle Routing with Stochastic Requests," Transportation Science, Vol. 53, pp. 185-202.

[Ulm17] Ulmer, M. W., 2017. Approximate Dynamic Programming for Dynamic Vehicle Routing, Springer, Berlin.

[VBC19] Vinyals, O., Babuschkin, I., Czarnecki, W. M., and thirty nine more authors, 2019. "Grandmaster Level in StarCraft II Using Multi-Agent Reinforcement Learning," Nature, Vol. 575, p. 350.

[VLK21] Vaswani, S., Laradji, I. H., Kunstner, F., Meng, S. Y., Schmidt, M., and Lacoste-Julien, S., 2021. "Adaptive Gradient Methods Converge Faster with Over-Parameterization (but you should do a line search)," arXiv:2006.06835.

[VPA09] Vrabie, D., Pastravanu, O., Abu-Khalaf, M., and Lewis, F. L., 2009. "Adaptive Optimal Control for Continuous-Time Linear Systems Based on Policy Iteration," Automatica, Vol. 45, pp. 477-484.

[VVL13] Vrabie, D., Vamvoudakis, K. G., and Lewis, F. L., 2013. Optimal Adaptive Control and Differential Games by Reinforcement Learning Principles, The Institution of Engineering and Technology, London.

[Van76] Van Nunen, J. A., 1976. Contracting Markov Decision Processes, Mathematical Centre Report, Amsterdam.

[Van78] van der Wal, J., 1978. "Discounted Markov Games: Generalized Policy Iteration Method," J. of Optimization Theory and Applications, Vol. 25, pp. 125-138.

[VeM23] Vertovec, N., and Margellos, K., 2023. "State Aggregation for Distributed Value Iteration in Dynamic Programming," arXiv preprint arXiv:2303.10675.

[WGB22] Wang, T. T., Gleave, A., Belrose, N., Tseng, T., Miller, J., Dennis, M. D., Duan, Y., Pogrebniak, V., Levine, S., and Russell, S., 2022. "Adversarial Policies Beat Professional-Level Go AIs," arXiv preprint arXiv:2211.00241.

[WCG02] Wu, G., Chong, E. K. P., and Givan, R. L., 2002. "Burst-Level Congestion Control Using Hindsight Optimization," IEEE Transactions on Aut. Control, Vol. 47, pp. 979-991.

[WCG03] Wu, G., Chong, E. K. P., and Givan, R. L., 2003. "Congestion Control Using Policy Rollout," Proc. 2nd IEEE CDC, Maui, Hawaii, pp. 4825-4830.

[WOB15] Wang, Y., O'Donoghue, B., and Boyd, S., 2015. "Approximate Dynamic Programming via Iterated Bellman Inequalities," International J. of Robust and Nonlinear Control, Vol. 25, pp. 1472-1496.

[WaB14] Wang, M., and Bertsekas, D. P., 2014. "Incremental Constraint Projection Methods for Variational Inequalities," Mathematical Programming, pp. 1-43.

[WaB16] Wang, M., and Bertsekas, D. P., 2016. "Stochastic First-Order Methods with Random Constraint Projection," SIAM Journal on Optimization, Vol. 26, pp. 681-717.

[WaS00] de Waal, P. R., and van Schuppen, J. H., 2000. "A Class of Team Problems with Discrete Action Spaces: Optimality Conditions Based on Multimodularity," SIAM J. on Control and Optimization, Vol. 38, pp. 875-892.

[Wat89] Watkins, C. J. C. H., Learning from Delayed Rewards, Ph.D. Thesis, Cambridge Univ., England.

[WeB99] Weaver, L., and Baxter, J., 1999. "Learning from State Differences: STD(λ)," Tech. Report, Dept. of Computer Science, Australian National University.

[WhS94] White, C. C., and Scherer, W. T., 1994. "Finite-Memory Suboptimal Design for Partially Observed Markov Decision Processes," Operations Research, Vol. 42, pp. 439-455.

[Whi82] Whittle, P., 1982. Optimization Over Time, Wiley, N. Y., Vol. 1, 1982, Vol. 2, 1983.

[Whi88] Whittle, P., 1988. "Restless Bandits: Activity Allocation in a Changing World," J. of Applied Probability, pp. 287-298.

[Whi91] White, C. C., 1991. "A Survey of Solution Techniques for the Partially Observed Markov Decision Process," Annals of Operations Research, Vol. 32, pp. 215-230.

[WiB93] Williams, R. J., and Baird, L. C., 1993. "Analysis of Some Incremental Variants of Policy Iteration: First Steps Toward Understanding Actor-Critic Learning Systems," Report NU-CCS-93-11, College of Computer Science, Northeastern University, Boston, MA.

[WiS98] Wiering, M., and Schmidhuber, J., 1998. "Fast Online Q(λ)," Machine Learning, Vol. 33, pp. 105-115.

[Wie03] Wiewiora, E., 2003. "Potential-Based Shaping and Q-Value Initialization are Equivalent," J. of Artificial Intelligence Research, Vol. 19, pp. 205-208.

[Wit66] Witsenhausen, H. S., 1966. Minimax Control of Uncertain Systems, Ph.D. thesis, MIT.

[Wit68] Witsenhausen, H., 1968. "A Counterexample in Stochastic Optimum Control," SIAM Journal on Control, Vol. 6, pp. 131-147.

[Wit71a] Witsenhausen, H. S., 1971. "On Information Structures, Feedback and Causality," SIAM J. Control, Vol. 9, pp. 149-160.

[Wit71b] Witsenhausen, H., 1971. "Separation of Estimation and Control for Discrete Time Systems," Proceedings of the IEEE, Vol. 59, pp. 1557-1566.

[YDR04] Yan, X., Diaconis, P., Rusmevichientong, P., and Van Roy, B., 2004. "Solitaire: Man Versus Machine," Advances in Neural Information Processing Systems, Vol. 17, pp. 1553-1560.

[YYM20] Yu, L., Yang, H., Miao, L., and Zhang, C., 2019. "Rollout Algorithms for Resource Allocation in Humanitarian Logistics," IISE Transactions, Vol. 51, pp. 887-909.

[Yar17] Yarotsky, D., 2017. "Error Bounds for Approximations with Deep ReLU Networks," Neural Networks, Vol. 94, pp. 103-114.

[YuB08] Yu, H., and Bertsekas, D. P., 2008. "On Near-Optimality of the Set of Finite-State Controllers for Average Cost POMDP," Math. of OR, Vol. 33, pp. 1-11.

[YuB09] Yu, H., and Bertsekas, D. P., 2009. "Basis Function Adaptation Methods for Cost Approximation in MDP," Proceedings of 2009 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL 2009), Nashville, Tenn.

[YuB13] Yu, H., and Bertsekas, D. P., 2013. "Q-Learning and Policy Iteration Algorithms for Stochastic Shortest Path Problems," Annals of Operations Research, Vol. 208, pp. 95-132.

[YuB15] Yu, H., and Bertsekas, D. P., 2015. "A Mixed Value and Policy Iteration Method for Stochastic Control with Universally Measurable Policies," Math. of OR, Vol. 40, pp. 926-968.

[YuK20] Yue, X., and Kontar, R. A., 2020. "Lookahead Bayesian Optimization via Rollout: Guarantees and Sequential Rolling Horizons," arXiv:1911.01004.

[Yu14] Yu, H., 2014. "Stochastic Shortest Path Games and Q-Learning," arXiv:1412.8570.

[Yua19] Yuanhong, L. I. U., 2019. "Optimal Selection of Tests for Fault Detection and Isolation in Multi-Operating Mode System," Journal of Systems Engineering and Electronics, Vol. 30, pp. 425-434.

[ZBH16] Zhang, C., Bengio, S., Hardt, M., Recht, B., and Vinyals, O., 2016. "Understanding Deep Learning Requires Rethinking Generalization," arXiv:

1611.03530.

[ZBH21] Zhang, C., Bengio, S., Hardt, M., Recht, B., and Vinyals, O., 2021. "Understanding Deep Learning (Still) Requires Rethinking Generalization," Communications of the ACM, VOL. 64, pp. 107-115.

[ZOT18] Zhang, S., Ohlmann, J. W., and Thomas, B. W., 2018. "Dynamic Orienteering on a Network of Queues," Transportation Science, Vol. 52, pp. 691-706.

[ZSG20] Zoppoli, R., Sanguineti, M., Gnecco, G., and Parisini, T., 2020. Neural Approximations for Optimal Control and Decision, Springer.

[ZYB21] Zhang, K., Yang, Z. and Basar, T., 2021. "Multi-Agent Reinforcement Learning: A Selective Overview of Theories and Algorithms.," Handbook of Reinforcement Learning and Control, pp. 321-384.

[ZuS81] Zuker, M., and Stiegler, P., 1981. "Optimal Computer Folding of Larger RNA Sequences Using Thermodynamics and Auxiliary Information," Nucleic Acids Res., Vol. 9, pp. 133-148.