Lessons from AlphaZero for Optimal, Model Predictive, and Adaptive Control

A Forthcoming Book

by

Dimitri P. Bertsekas

Arizona State University
and
Massachusetts Institute of Technology

DRAFT OF

November 15, 2021

This monograph represents “work in progress,” and may contain errors, hopefully not serious. It will be periodically updated and enlarged. The latest version can be downloaded from


Your comments and suggestions to the author at dimitrib@mit.edu are welcome. The monograph is scheduled for publication for 2022 as part of a series of related books published by

Athena Scientific, Belmont, Massachusetts

WWW site for book information and orders
http://www.athenasc.com
Just because you are right, does not mean, I am wrong. You just haven't seen life from my side.
PREFACE

The purpose of this monograph is to propose and develop a new conceptual framework for approximate Dynamic Programming (DP) and Reinforcement Learning (RL). This framework centers around two algorithms, which are designed largely independently of each other and operate in synergy through the powerful mechanism of Newton’s method. We call these the off-line training and the on-line play algorithms; the names are borrowed from 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). In these 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.

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 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 in algorithmic mathematics and is the focal point of this work: 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 indirect adaptive control. Here the Bellman equation is perturbed due to the parameter changes, but approximation in value space still operates as a Newton step. An essential requirement here is that a system model is estimated on-line through some identification method, and is used during the one-step or multistep lookahead minimization process.

In this work we will aim to provide insights (often based on visualization), which explain 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, and the control theory views of MPC and adaptive control. While we will deemphasize mathematical proofs, there is considerable related analysis, which supports our conclusions and can be found in the author’s recent RL books [Ber19a], [Ber20a], and the abstract DP monograph [Ber18a].

One of our principal aims is 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.
CONTENTS

1. AlphaZero, Off-Line Training, and On-Line Play ............................................. p. 1
   1.1. Off-Line Training and Policy Iteration ...................................................... p. 2
   1.2. On-Line Play and Approximation in Value Space - Truncated Rollout .......... p. 4
   1.3. The Lessons of AlphaZero ......................................................................... p. 6
   1.4. A New Conceptual Framework for Reinforcement Learning ...................... p. 9
2. Deterministic and Stochastic Dynamic Programming Over an Infinite Horizon ............ p. 11
   2.1. Optimal Control Over an Infinite Horizon - Value and Policy Iteration .......... p. 11
   2.2. Approximation in Value Space - One-Step and Multistep Lookahead - Stability .... p. 17
3. An Abstract View of Reinforcement Learning ....................................................... p. 21
   3.1. Approximation in Value Space and Newton’s Method ................................. p. 28
   3.2. Region of Stability ...................................................................................... p. 30
   3.3. Policy Iteration, Rollout, and Newton’s Method .......................................... p. 33
   3.5. Why Not Just Train a Policy Network and Use it Without On-Line Play? ...... p. 41
   3.6. Multiagent Problems and Multiagent Rollout ............................................. p. 42
   3.7. On-Line Simplified Policy Iteration ............................................................ p. 45
   3.8. Exceptional Cases ....................................................................................... p. 51
4. The Linear Quadratic Case - Illustrations .............................................................. p. 55
   4.1. Optimal Solution ......................................................................................... p. 56
   4.2. Cost Functions of Stable Suboptimal Linear Policies .................................... p. 56
   4.3. Bellman’s Equation and Value Iteration ..................................................... p. 59
   4.4. Approximation in Value Space with One-Step and Multistep Lookahead - Stability Issues p. 60
   4.5. Rollout and Policy Iteration ....................................................................... p. 62
   4.6. Truncated Rollout - Tradeoffs with Lookahead Minimization ..................... p. 64
   4.7. Exceptional Behavior in Linear Quadratic Problems .................................... p. 66
5. Robustness, Adaptive Control, and Model Predictive Control ........................................ p. 68
   5.1. Approximation in Value Space, Rollout, and Adaptive Control .................... p. 71
   5.2. Approximation in Value Space, Rollout, and Model Predictive Control ........ p. 74
6. Concluding Remarks ............................................................................................ p. 81
7. References ........................................................................................................... p. 83
8. APPENDIX: Finite Horizon Deterministic Problems - Discrete Optimization ............... p. 90
1. ALPHAZERO, OFF-LINE TRAINING, AND ON-LINE PLAY

Perhaps the most impressive success story in reinforcement learning (RL) is the development of the AlphaZero program by DeepMind Inc; see [SHS17], [SSS17]. AlphaZero plays Chess, Go, and other games, and is an improvement in terms of performance and generality over the earlier AlphaGo program [SHM16], which plays the game of Go only. AlphaZero plays better than all competitor computer programs available in 2021, 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).

We should note also that the principles of the AlphaZero design have much in common with the TD-Gammon program of Tesauro [Tes94], [Tes95], [TeG96] that plays backgammon (a game of substantial computational and strategical complexity, which involves a number of states estimated to be in excess of $10^{20}$). 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 program for the (one-player) game of Tetris, based on similar principles, is described by Scherrer et al. [SGG15], together with several antecedents. The backgammon and tetris programs, while dealing with less complex games than chess, are of special interest because they involve significant stochastic uncertainty, and are thus unsuitable for the use of long lookahead minimization, which is widely believed to be one of the major contributors to the success of AlphaZero, and chess programs in general.

Still, for all of their brilliant implementations, these impressive game programs are couched on well established methodology, from optimal and suboptimal control, which is portable to far broader domains of engineering, economics, and other fields. This is the methodology of dynamic programming (DP), policy iteration, limited lookahead minimization, rollout, and related approximations in value space. The aim of this work is to propose a conceptual, somewhat abstract framework, which allows insight into the connections of AlphaZero and TD-Gammon with some of the core problems in decision and control, and suggests potentially far-reaching extensions.

To understand the overall structure of AlphaZero and related programs, and their connections to the DP/RL methodology, it is useful to divide their design into two parts:

(a) **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.

(b) **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, focusing on AlphaZero for the most part.

1.1 Off-Line Training and Policy Iteration

An off-line training algorithm like the one used in AlphaZero is the part of the program that learns how to play through self-training that takes place before real-time play against any opponent. It is illustrated in Fig. 1.1, and it generates a sequence of chess players and position evaluators. A chess player assigns “probabilities” to all possible moves at any given chess position: these may be viewed as a measure of “effectiveness” of the corresponding moves. A position evaluator assigns a numerical score to any given chess position, and thus predicts quantitatively the performance of a player starting from any position. The chess player and the position evaluator are represented by neural networks, a policy network and a value network, which accept a chess position and generate a set of move probabilities and a position evaluation, respectively.†

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

The overall training algorithm is a form of policy iteration, a DP algorithm that will be of primary interest to us in this work. 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 into two stages (see Fig. 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.

† Here the neural networks play the role of function approximators. 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. Actually, AlphaZero uses the same neural network for training both value and policy. Thus there are two outputs of the neural net: value and policy. This is pretty much equivalent to having two separate neural nets and for the purposes of this work, we prefer to explain the structure as two separate networks. AlphaGo uses two separate value and policy networks. Tesauru’s backgammon programs use a single value network, and generate moves when needed by one-step or two-step lookahead minimization, using the value network as terminal position evaluator.

‡ 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 and algorithms extend to two-player games, although we will not discuss these extensions.
Figure 1.1 Illustration of the AlphaZero off-line 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.

(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 AlphaZero (as well as AlphaGo Zero, the version that plays the game of Go) the policy evaluation is done by using deep neural networks. The policy improvement uses a complicated algorithm called *Monte Carlo Tree Search* (MCTS for short), a form of randomized multistep lookahead minimization that enhances the efficiency of the multistep lookahead operation, by pruning intelligently the multistep lookahead tree. However, deep neural networks and MCTS, while leading to some performance gains, are not of fundamental importance. The approximation quality that a deep neural network can achieve can also be achieved with a shallow neural network, perhaps with reduced sample efficiency. Similarly MCTS cannot achieve better lookahead accuracy than standard exhaustive search, although it may be more efficient computationally. Indeed, policy improvement can be done more simply without MCTS, as in Tesauro’s TD-Gammon program: we try all possible move sequences from a given position, extend forward to some number of moves, and then evaluate the terminal position with the current 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.†

† 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...
Regardless of the use of deep neural networks and MCTS, it is important to note that the final policy and the corresponding policy evaluation produced by approximate policy iteration and neural network training in AlphaZero involve serious inaccuracies, due to the approximations that are inherent in neural network representations. The AlphaZero on-line player to be discussed next uses approximation in value space with multistep lookahead minimization, and does not involve any neural network, other than the one that has been trained off-line, so it is not subject to such inaccuracies. As a result, it plays much better than the off-line player.

1.2 On-Line Play and Approximation in Value Space - Truncated Rollout

Consider the “final” player obtained through the AlphaZero off-line training process. It can play against any opponent by generating move probabilities at any position using its off-line trained policy network, and then simply play the move of highest probability. 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 obtained from off-line training is embedded into another algorithm, which we refer to as the “on-line player.” In other words AlphaZero plays on-line much better than the best player it has produced with sophisticated off-line training. This phenomenon, policy improvement through on-line play, is centrally important for our purposes in this work.

Given the policy network/player obtained off-line and its value network/position evaluator, the on-line algorithm plays roughly as follows (see Fig. 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 value network. The middle portion, called “truncated rollout,” may be viewed as an economical substitute for longer lookahead minimization. Actually truncated rollout is not used in the published version of AlphaZero [SHS17]; the first portion (multistep lookahead minimization) is very long and implemented efficiently (partly through the use of MCTS), so that the rollout portion is not essential. Rollout has been used in AlphaGo, the AlphaZero predecessor [SHM16]. Moreover, chess and Go programs (including AlphaZero) typically use a well-known limited form of rollout, called “quiescence search,” which aims to resolve imminent threats and highly dynamic positions through simulated multi-move piece exchanges, before invoking the position evaluator. Rollout is instrumental in achieving high parameters to match the search probabilities and selfplay game outcome respectively.”

† Quoting from the paper [SSS17]: “The MCTS search outputs probabilities of playing each move. These search probabilities usually select much stronger moves than the raw move probabilities of the neural network.” To elaborate, this statement refers to the MCTS algorithm that is used on line to generate the move probabilities at each position encountered in the course of a given game. The neural network referred to is trained off-line, also using in part the MCTS algorithm.
Figure 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.

performance in Tesauro’s 1996 backgammon program [TeG96]. The reason is that backgammon involves stochastic uncertainty, so long lookahead minimization is not possible because of rapid expansion of the lookahead tree with every move.†

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 number of steps in 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 context. We will discuss further the structure of MPC and its similarities with the AlphaZero architecture in Section 5.2.

The DP ideas with cost function approximations that are similar to the on-line player illustrated in Fig. 1.2, are also known as *approximate dynamic programming*, or *neuro-dynamic programming*, and will be

---

† Tesauro’s rollout-based backgammon program [TeG96] uses only a value network, called TD-Gammon, which was trained using an approximate policy iteration scheme developed several years earlier [Tes94]. TD-Gammon is used to generate moves for the truncated rollout via a one-step or two-step lookahead minimization. Thus the value network also serves as a substitute for the policy network during the rollout operation. The terminal position evaluation used at the end of the truncated rollout is also provided by the value network. The middle portion of Tesauro’s scheme (truncated rollout) is important for achieving a very high quality of play, as it effectively extends the length of lookahead from the current position.

‡ The Matlab toolbox for MPC design explicitly allows the user to set these two intervals.
central for our purposes. They will be generically referred to as approximation in value space in this work.†

Note also that in general, off-line training and on-line policy implementation may be designed independently of each other. For example the off-line training portion may be very simple, such as using a known heuristic 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 following the lookahead minimization in a value space approximation scheme.

1.3 The Lessons of AlphaZero

The AlphaZero and TD-Gammon experiences reinforce an important conclusion that applies more generally to decision and control problems: despite the extensive off-line effort that may have gone into the design of a policy, its performance may be greatly improved by on-line approximation in value space, with extra lookahead involving minimization and/or with rollout using this policy, and terminal cost approximation.

In the following sections, we will aim to amplify on this theme and to focus on the principal characteristics of AlphaZero-like architectures, within a broader context of optimal decision and control. We will make use of intuitive visualization, and the central role of Newton’s method for solving Bellman’s equation‡.

† 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. However, as a methodological subject, approximation in policy space has little connection to the ideas of the present work.

‡ Bellman’s equation, the centerpiece of infinite horizon DP theory, is viewed here as a functional equation, whose solution is the cost of operating the system viewed as a function of the system’s initial state. We will give examples of Bellman’s equation in Section 2 for discounted and other problems, and we will also provide in Section 3 abstract forms of Bellman’s equation that apply more generally.
Briefly, our central point is that on-line approximation in value space 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; see Fig. 1.3. Moreover, this starting point may be enhanced by several types of on-line operations, including longer lookahead minimization, and on-line rollout with a policy obtained through off-line training, or heuristic approximations.

This interpretation will be the basis for powerful insights into issues of stability, performance, and robustness of the on-line generated policy. In particular, we will aim to show that feedback control, based on approximation in value space and the underlying off-line training/on-line play structure, offers benefits that go well beyond the conventional wisdom that “feedback corrects for uncertainty, and modeling errors.” The reason is that by overlaying on-line play on top of off-line training, we gain significantly in performance, by correcting (through the Newton step) for the errors that are inherent in off-line training with approximation architectures such as neural networks.

Our mathematical framework is couched on unifying principles of abstract DP, including abstract forms of Bellman’s equation, and the value and policy iteration algorithms (see the author’s books [Ber12], [Ber18a]). However, in this work, we will deemphasize mathematical proofs. There is considerable related analysis, which supports our conclusions and can be found in the author’s recent RL books [Ber19a], [Ber20a].

In summary, our discussion will aim to highlight the following points:

**Summary**

(a) Approximation in value space is a single exact step of Newton’s method for solving Bellman’s equation (perhaps preceded by preliminary on-line adjustments and/or value iterations to enhance the starting point of the Newton step).

(b) The starting point for the Newton step of (a) is obtained by some unspecified off-line methodology, which may involve the solution of a related but simpler problem, and/or training with data that makes use of neural networks or feature-based architectures.

(c) The on-line play and off-line training parts of the AlphaZero/TD-Gammon design structure correspond to (a) and (b) above, respectively.

(d) The on-line player of AlphaZero plays much better than its deep neural network-trained player for the same reason that the Newton step (a) improves substantially on its starting point (b), namely the underlying superlinear convergence property that is typical of Newton’s method.

(e) \( \ell \)-step lookahead minimization can be viewed as one-step lookahead minimization where \( \ell - 1 \) value iterations are used to enhance the starting point of the Newton step of (a) above.
The algorithmic processes for (a) and (b) above can be designed by a variety of methods, and independently of each other. For example:

1. The implementation of the Newton step (a) may or may not involve any of the following: truncated rollout, on-line Monte Carlo simulation, MCTS or other efficient tree search techniques, forms of continuous space optimization, on-line policy iteration, etc.

2. The computation of the starting point (b) may or may not involve any of the following: Q-learning, approximate policy iteration based on temporal differences or aggregation, neural networks, feature-based function approximation, policies trained off-line by approximation in policy space, including policy gradient methods or policy random search, etc. Moreover, the details of this computation may vary broadly without affecting significantly the effectiveness of the overall scheme, which is primarily determined by the Newton step (a).

An efficient implementation of the Newton step (a) is often critical in order to meet real-time constraints for generating controls, and to allow longer lookahead minimization, which typically enhances the starting point of the Newton step and its performance. By contrast, off-line training algorithms used for (b) have much less stringent real-time constraints, and the issues of sample efficiency and fine tuned performance, while important, are not critical.

The efficient implementation of the Newton step may benefit from the use of distributed computation and other simplifications. A case in point is multiagent problems, which we will discuss later (see Section 3.4).

Approximation in value space addresses effectively issues of robustness and on-line replanning for problems with changing parameters. The mechanism is similar to the one of indirect adaptive control: changing problem parameters are estimated on-line and a Newton step is used in place of an expensive full reoptimization of the controller. In the presence of changing parameters, the Bellman equation changes, but the Newton step itself remains powerful and aims at the optimal solution that corresponds to the estimated system parameters.

Model predictive control (MPC) has a conceptually similar structure to the AlphaZero-like programs, and entails an on-line play component involving multistep lookahead minimization, forms of truncated rollout, and an off-line training component to construct terminal cost approximations, and “safe” state space regions or reachability tubes to deal with state constraints. The success of MPC may be attributed to these similarities and to its resilience to changing problem parameters as per (i) above.
(k) On-line rollout with a stable policy yields a favorable starting point for the Newton step (a): it improves the stability properties of the policy obtained by approximation in value space, and it often provides an economical substitute for long lookahead minimization.

(l) Because the ideas outlined above are couched on principles of DP that often hold for arbitrary state and control spaces, they are valid within very general contexts: continuous-spaces control systems, discrete-spaces Markovian decision problems, control of hybrid systems, and discrete and combinatorial optimization.

The preceding points are meant to highlight the essence of the connections between AlphaZero and TD-Gammon, approximation in value space, and decision and control. Naturally in practice there are exceptions and modifications, which need to be worked out in the context of particular applications, under appropriate assumptions. Moreover, while some results and elaborations are available through the research that has been done on approximate DP and on MPC, several of the results suggested by the analysis and insights of the present work remain to be rigorously established and enhanced within the context of specific problems.

1.4 A New Conceptual Framework for Reinforcement Learning

In this work we will emphasize the distinct roles of off-line training and on-line play algorithms within the structure of approximate sequential decision making and approximation in value space schemes. In doing so, we will aim for a new conceptual framework for RL, which is based on the synergism and complementarity of off-line training and on-line play.

We will implicitly assume that the time available for off-line computation is very long (practically limitless), but that the problem at hand is such that exact DP algorithms, such as policy iteration and Q-learning, are impossible for one (or both) of the following two reasons:

(a) There are too many states (either an infinite number as in continuous space problems, or very large as in chess). As a result a lookup table representation of policies, value functions, and/or Q-factors is impossible, and the only practical alternative is a compact representation, via a neural network or some other approximation architecture.

(b) The system model is changing over time as in adaptive control, so even if an exactly optimal policy is computed off-line under some nominal problem assumptions, it becomes suboptimal when the problem parameters change.

On the other hand, we will assume that there is limited time for on-line decision making, because of hard practical constraints on the real time that is available between decisions. These constraints are highly
problem dependent: for some problems, following the observation of the state, we may need to produce the next decision within a fraction of a second, whereas for others we may have hours at our disposal. We will assume that *whatever time is available, it will be used to provide adequate quality of lookahead minimization, and time permitting, to extend as much as possible the combined length of the lookahead minimization and the truncated rollout with an off-line computed policy*. We will thus implicitly take it as given that longer lookahead minimization is better for the performance of the policy obtained, although the division of effort between lookahead minimization and truncated rollout with a policy is a design decision that may depend on the circumstances. Note that parallel and distributed computation can play an important role in mitigating practical on-line time constraints.

The central fact in our conceptual framework is that one-step lookahead minimization constitutes a single Newton step for solving Bellman’s equation. Contrary to other Newton-like steps that may have been part of the off-line training process, this single Newton step is accurate: *all the approximation has been shifted to its starting point*. Moreover, the Newton step can be very powerful, and its starting point can be enhanced by multistep lookahead minimization or by truncated rollout. From an algorithmic point of view, the Newton step converges superlinearly without the need for differentiability of the Bellman operator $T$: it takes advantage of the monotonicity and concavity structure of $T$ (see the discussion of Section 3).

To summarize, both off-line training and on-line play are subject to fundamental limits: the former’s limit is the constrained power of the approximation architecture, while the latter’s limit is the constrained on-line computation time. The former limit cannot be easily overcome, but the latter limit can be stretched a lot thanks to the power of the Newton step, supplemented by long lookahead minimization and truncated rollout, as well as through the use of parallel and distributed computation.

Our design philosophy in a nutshell is the following:

1. The major determinant of the quality of the controller obtained by our schemes is the single Newton step that is performed on-line. Stated somewhat bluntly, *off-line training is secondary by comparison*, in the sense that without on-line one-step or multistep lookahead minimization, the quality of the policy obtained by off-line training alone is often unacceptably poor. In particular, whether done by neural networks, feature-based linear architectures, temporal difference methods, aggregation, policy gradient, policy random search, or whatever other reasonable approach, off-line training principally serves the purpose of providing a good or reasonable starting point for the Newton step. This is the principal lesson from AlphaZero and TD-Gammon in our view. This philosophy also underlies MPC, where on-line lookahead minimization has traditionally been the principal focus, perhaps supplemented

---

† It is possible to construct artificial problems, where longer lookahead results in worse performance (see [Ber19a], Section 2.2), but such problems are rare in practice.
by truncated rollout, with off-line calculations playing a limited subsidiary role.†

(2) The Newton step is often powerful enough to smooth out differences in various off-line training methods. In particular, methods such as TD(\(\lambda\)) with different values of \(\lambda\), policy gradient, linear programming, etc, all give different, but more or less equally good starting points for the Newton step. The conclusion from this is that off-line training with a very large number of samples, and sophisticated ways to improve sample efficiency may not be very useful, beyond a certain point, because gains in efficiency and accuracy tend to be washed up by the Newton step.

(3) The on-line Newton step also works well in the context of adaptive control, as long as it is calculated on the basis of the currently correct model parameters (so this requires an on-line parameter identification algorithm). The reason is that when problem parameters change, the Bellman operator changes, but the Newton step is executed on the basis of the correct Bellman operator. This is also a principal reason why MPC schemes have been used with success in adaptive control contexts.

We will return to these points repeatedly in the course of our presentation.

The present work is structured as follows. In Section 2 we review the theory of classical infinite horizon optimal control problems, in order to provide some orientation and an analytical platform for what follows in subsequent sections. In Section 3, we introduce an abstract DP framework that will set the stage for the conceptual and visual interpretations of approximation in value space in terms of Newton’s method. In this section, we also discuss on-line policy iteration, which aims to improve the on-line approximation in value space algorithm by using training data that is collected on-line. In Section 4, we illustrate our analysis within the simple and intuitive framework of linear quadratic problems. In Section 5, we discuss various issues of changing problem parameters, adaptive control, and MPC. Extensions of the ideas of this work to discrete optimization problems, and particularly rollout algorithms and their variations, are briefly presented in the appendix, and have been discussed extensively in the author’s book [Ber20a].

2. DETERMINISTIC AND STOCHASTIC DYNAMIC PROGRAMMING

In this section we will describe a classical optimal control framework, illustrated in Fig. 2.1, which we will use as a principal example for a more abstract DP framework to be introduced in Section 3. This abstract framework will be used in turn as the starting point for our analysis and visualization of algorithmic issues, relating to approximation in value space, multistep lookahead, controller stability, truncated rollout, and policy iteration.

† Incidentally, this is a major reason why there is an apparent disconnect between the MPC community, which is mostly focused on-line play, and the RL community, which is mostly focused on off-line training.
2.1 Optimal Control Over an Infinite Horizon

Let us consider a familiar class of stochastic optimal control problems over an infinite horizon. We have a stationary system of the form

\[ x_{k+1} = f(x_k, u_k, w_k), \quad k = 0, 1, \ldots, \]

where \( x_k \) is an element of some state space \( X \) and the control \( u_k \) is an element of some control space \( U \); see Fig. 2.1. The system includes a random “disturbance” \( w_k \) with a probability distribution \( P(\cdot \mid x_k, u_k) \) that may depend explicitly on \( x_k \) and \( u_k \), but not on values of prior disturbances \( w_{k-1}, \ldots, w_0 \).† The control \( u_k \) is constrained to take values in a given subset \( U(x_k) \subset U \), which depends on the current state \( x_k \). We are interested in policies \( \pi = \{\mu_0, \mu_1, \ldots\} \), such that each function \( \mu_k \) maps states into controls, and satisfies \( \mu_k(x_k) \in U(x_k) \) for all \( k \). A stationary policy of the form \( \{\mu, \mu, \ldots\} \) will also be referred to as “policy \( \mu \).” We make no assumptions on the state, control, and disturbances, and indeed for most of the discussion of this work, these spaces can be arbitrary.

We aim to minimize the expected total cost over an infinite number of stages, given by

\[ J_{\pi}(x_0) = \lim_{N \to \infty} E \left\{ \sum_{k=0}^{N-1} \alpha^k g(x_k, \mu_k(x_k), w_k) \right\}, \quad (2.1) \]

where \( \alpha^k g(x_k, u_k, w_k) \) is the cost of stage \( k \), and \( \alpha \in (0, 1] \) is a discount factor. If \( \alpha = 1 \) we refer to the problem as undiscounted. The expected value in Eq. (2.1) is taken with respect to the random disturbances \( w_k, k = 0, 1, \ldots \). Here, \( J_{\pi}(x_0) \) denotes the cost associated with an initial state \( x_0 \) and a policy \( \pi = \{\mu_0, \mu_1, \ldots\} \). The cost function of a stationary policy \( \mu \) is denoted by \( J_{\mu} \). The optimal cost starting at state \( x \), \( \inf_{\pi} J_{\pi}(x) \), is denoted by \( J^*(x) \), and the function \( J^* \) is referred to as the optimal cost function.

Let us consider some special cases, which will be of primary interest in this work:

(a) Stochastic shortest path problems (SSP for short). Here, \( \alpha = 1 \) but there is a special cost-free termination state, denoted by \( t \); once the system reaches \( t \) it remains there at no further cost. Usually,

† We assume an introductory probability background on the part of the reader. For an account that is consistent with our use of probability in this book, see the text by Bertsekas and Tsitsiklis [BeT08].
the termination state $t$ represents a goal state that we are trying to reach at minimum cost; these are problems where the cost per stage is nonnegative, and will of primary interest in this work. In some other types of problems, $t$ may be a state that we are trying to avoid for as long as possible; these are problems where the cost per stage is nonpositive, and will not be specifically discussed in this work.

(b) Discounted stochastic problems. Here, $\alpha < 1$ and there need not be a termination state. However, there is a substantial connection between SSP and discounted problems. Aside from the fact that they are both infinite horizon total cost optimization problems, a discounted problem can be readily converted to an SSP problem. This can be done by introducing an artificial termination state to which the system moves with probability $1 - \alpha$ at every state and stage, thus making termination inevitable. Thus SSP and discounted problems share qualitative similarities in their respective theories.

(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

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

and

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

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

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

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].

An important special case is finite-state deterministic problems. Finite horizon versions of these problems include challenging discrete optimization problems, whose exact solution is practically impossible. It is possible to transform such problems to infinite horizon SSP problems, so that the conceptual framework developed here applies. The approximate solution of discrete optimization problems by RL methods, and particularly by rollout, has been discussed at length in the books [Ber19a] and [Ber20a].

Another important deterministic nonnegative cost problem is the classical continuous spaces problem where the system is linear, with no control constraints, and the cost function is quadratic. We will often refer to this problem and its extensions in what follows.

**Example 2.1 (Linear Quadratic Problems)**

Assume that the system is linear of the form

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

(2.5)
where \( x_k \) and \( u_k \) are elements of the Euclidean spaces \( \mathbb{R}^n \) and \( \mathbb{R}^m \), respectively, \( A \) is an \( n \times n \) matrix, and \( B \) is an \( n \times m \) matrix. We assume 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). It is well known that this problem admits a nice analytical solution, which we will discuss shortly, and we will use later for illustrations, examples, and counterexamples (see also [Ber17a], Section 3.1).

**Infinite Horizon Methodology**

Many of the analytical and computational issues regarding infinite horizon problems revolve around the relation between the optimal cost function \( J^* \) of the problem and the optimal cost function of the corresponding \( N \)-stage problem. 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 value iteration algorithm (VI for short)
\[
J_{k+1}(x) = \min_{u \in U(x)} E\{g(x, u, w) + \alpha J_k(f(x, u, w))\}, \quad k = 0, 1, \ldots,
\]
starting from \( J_0(x) \equiv 0 \) (see the Appendix). It is natural to speculate the following three basic properties:

† (1) The optimal infinite horizon cost is the limit of the corresponding \( N \)-stage optimal costs as \( N \to \infty \):
\[
J^*(x) = \lim_{N \to \infty} J_N(x)
\]
for all states \( x \).

(2) Bellman’s equation holds:
\[
J^*(x) = \min_{u \in U(x)} E\{g(x, u, w) + \alpha J^*(f(x, u, w))\}, \quad \text{for all } x.
\]
This equation can be viewed as the limit as \( k \to \infty \) of the VI algorithm (2.7), assuming property (1) above holds and guarantees that \( J_k(x) \to J^*(x) \) for all \( x \). There is also a Bellman equation for each stationary policy \( \mu \). It is given by
\[
J_\mu(x) = E\{g(x, \mu(x), w) + \alpha J_\mu(f(x, \mu(x), w))\}, \quad \text{for all } x,
\]
where \( J_\mu \) is the cost function of \( \mu \).

† Throughout this work, we will be using “min” instead of the more formal “inf,” even we are not sure that minimum is attained.
If \( \mu(x) \) attains the minimum in the right-hand side of the Bellman equation (2.9) for each \( x \), then the stationary policy \( \mu \) should be optimal.

All three of the preceding results hold for discounted problems, provided the expected cost per stage \( E\{g(x, u, w)\} \) is bounded over the set of possible values of \( (x, u, w) \) (see [Ber12], Chapter 1). They also hold for finite-state SSP problems under reasonable assumptions. For deterministic problems with possibly infinite state and control spaces, there is substantial analysis that provides assumptions under which the results (1)-(3) above hold (see e.g., the DP book [Ber12]).

The VI algorithm is also typically valid, 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 intuitive interpretation of the Bellman equation (2.9) is that it is the limit as \( k \to \infty \) of the VI algorithm (2.7) assuming that \( J_k \to J^* \). The optimality condition (3) indicates that optimal and near optimal policies can be obtained from within the class of stationary policies, something that is generally true for the problems that we discuss in this work, and that we will implicitly assume in what follows.

Aside from the VI algorithm, another fundamental algorithm is policy iteration (PI for short), which will be discussed in Section 3.3. It can be viewed as Newton’s method for solving Bellman’s equation, and is central for our purposes.

The author’s paper [Ber17b], and also the abstract DP book [Ber18a], provide a detailed analysis of the undiscounted special case of the problem (2.2)-(2.4), where there is a cost-free and absorbing termination state \( t \), the cost function is strictly positive for all other states, as in Eq. (2.4), and the objective is to reach or asymptotically approach the termination state. This analysis covers the preceding four properties, as well as the issue of convergence of PI, for the case of general state and control spaces (continuous or discrete or a mixture thereof). It delineates conditions under which favorable properties can be guaranteed.

**Example 2.1 (Linear Quadratic Problems - Continued)**

Consider again the linear quadratic problem defined by Eqs. (2.5)-(2.6). The Bellman equation is given by

\[
J(x) = \min_{u \in \mathbb{R}^m} \left\{ x'Qx + u'Ru + J(Ax + Bu) \right\},
\]

and turns out to have a unique solution within the space of quadratic functions of the form

\[
J(x) = x'Kx,
\]

where \( K \) is a positive semidefinite symmetric matrix [under our positive definiteness assumption on \( Q \) and \( R \), and an additional controllability assumption on the system (2.5); see [Ber17a], Section 3.1]. This unique solution can be shown to be the optimal cost function of the problem, and has the form

\[
J^*(x) = x'K^*x.
\]
We can obtain \( K^* \) by solving the matrix equation

\[
K = F(K),
\]

(2.14)

with \( F(K) \) defined over symmetric matrices \( K \) by

\[
F(K) = A' \left( K - KB(B'KB + R)^{-1}B'K \right) A + Q.
\]

(2.15)

The optimal policy is obtained by minimization in the Bellman equation (2.11) when \( J \) is replaced by the optimal cost function \( J^* \) of Eq. (2.13). It can be verified that it is linear of the form

\[
\mu^*(x) = Lx,
\]

where \( L \) is the matrix

\[
L = -(B'K^*B + R)^{-1}B'K^*A.
\]

The VI and PI algorithms are known to have favorable properties for our linear quadratic problem. In particular, the VI algorithm can be executed within the space of positive semidefinite symmetric matrices. The VI algorithm \( J_{k+1} = TJ_k \), when \( J_k \) has the form \( J_k(x) = x'K_kx \), yields for all \( x \),

\[
J_{k+1}(x) = x'K_{k+1}x \quad \text{with} \quad K_{k+1} = F(K_k),
\]

(2.16)

where \( F \) is given by Eq. (2.15). It can be shown that the sequence \( \{K_k\} \) converges to the optimal matrix \( K^* \) starting from any positive semidefinite symmetric matrix \( K_0 \) under the assumptions mentioned earlier. The PI algorithm also has favorable convergence properties (under the same assumptions) and its important connection with Newton’s method will be discussed later.

The preceding results are well known and they are given with proofs in several control theory texts, including the author’s DP books [Ber17a], Chapter 3, and [Ber12], Chapter 4. The equation \( K = F(K) \) is known as the Riccati equation. It can be viewed as the Bellman equation restricted to the subspace of quadratic functions of the form (2.12). Note that the Riccati equation can be shown to have solutions other than \( K^* \) (which necessarily are not positive definite symmetric). Illustrative examples will be given later.

† Actually the preceding formulas also hold even when the positive definiteness assumption on \( Q \) is replaced by other weaker conditions (see [Ber17a], Section 3.1). We will not go into the details of this, but we note that some condition on \( Q \) is needed for the preceding results to hold, as we will show later by example in Section 4.

‡ The Riccati differential equation, invented in its one-dimensional form by count Jacopo Riccati in the 1700s, 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.
2.2 Approximation in Value Space - One-Step and Multistep Lookahead - Stability

A principal RL approach to deal with the often intractable exact computation of $J^*$ is approximation in value space. 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

$$\tilde{\mu}(x) \in \arg \min_{u \in U(x)} E\{g(x, u, w) + \alpha \tilde{J}(f(x, u, w))\}; \quad (2.17)$$

(we implicitly assume that the minimum above is attained for all $x$). Note that the general theory of abstract DP is developed with the use of extended real-valued functions, and without the attainment of minimum assumption; see [Ber18a]. 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 $x$]. In the next section, the change from $\tilde{J}$ to $J_{\tilde{\mu}}$ will be interpreted as a step of Newton’s method for solving Bellman’s equation. Among others, this will suggest that $J_{\tilde{\mu}}$ is close to $J^*$ and obeys a superlinear convergence relation

$$\lim_{\tilde{J} \to J^*} \frac{J_{\tilde{\mu}}(x) - J^*(x)}{J(x) - J^*(x)} = 0,$$

for all states $x$. For specific types of problems, this relation represents a plausible result, which likely holds under appropriate conditions. This is similar to the use of Newton’s method in numerical analysis, where its global or local convergence is guaranteed only under some assumptions. Within our context of approximate DP, however, there is an important underlying structure, which is favorable and enhances the convergence properties of Newton’s method, namely the monotonicity and concavity properties of Bellman’s equation, as we will discuss in what follows.

While it is desirable that $J_{\tilde{\mu}}$ is close to $J^*$ in some sense, for classical control problems involving control to a goal 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. For the purposes of this work, we will focus on stability issues for just 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.,

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

Selecting $\tilde{J}$ so that $\tilde{\mu}$ is stable is a question of major interest, and will be addressed in Section 3.

$\ell$-Step Lookahead

An important extension of one-step lookahead minimization is $\ell$-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 Fig. 2.2). 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 $\ell$-step lookahead policy $\tilde{\mu}$ via $\tilde{\mu}(x_k) = \tilde{u}_k$. The sequence $\tilde{\mu}_{k+1}, \ldots, \tilde{\mu}_{k+\ell-1}$ is discarded.
Actually, we may view \( \ell \)-step lookahead minimization as the special case of its one-step counterpart where the lookahead function is the optimal cost function of an \((\ell - 1)\)-stage DP problem with a terminal cost \( \tilde{J}(x_{k+\ell}) \) on the state \( x_{k+\ell} \) obtained after \( \ell - 1 \) stages. In the next section, this will be interpreted as a step of Newton’s method for solving Bellman’s equation, starting from a function \( \hat{J} \), which is an “improvement” over \( \tilde{J} \). In particular, \( \hat{J} \) is obtained from \( \tilde{J} \) by applying \( \ell - 1 \) successive value iterations.

The motivation for \( \ell \)-step lookahead minimization is that by increasing the value of \( \ell \), 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 \( \ell \) becomes larger. This will be explained visually in the next section, and is also supported by error bounds, given for example in the books [Ber19a], [Ber20a]. 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. On the other hand, solving the multistep lookahead optimization problem, instead of the one-step lookahead counterpart of Eq. (2.17), is more time consuming.

Constructing Terminal Cost Approximations

A major issue in value space approximation is the construction of suitable approximate cost functions \( \tilde{J} \). 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 a sophisticated off-line training method, as discussed in Section 1, in connection with chess and backgammon. Alternatively, the approximate values \( \tilde{J}(x) \) are 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 terminal cost approximation. While the method by which we obtain \( \hat{J} \) will not be important for understanding the ideas of this work, for orientation purposes we briefly describe four broad types of approximation, and refer to the RL and approximate DP literature for further details:

(a) **Problem approximation**: Here the function \( \hat{J} \) is obtained 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, and ignoring various types of uncertainties. For example we may consider using as \( \hat{J} \) the cost function of a related deterministic problem, obtained through some form of “certainty equivalence,” thus allowing computation of \( \hat{J} \) 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 [Ber18b], [Ber18c]. 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 \( \hat{J} \) to the optimal cost function \( J^* \) of the original problem, using some form of interpolation.

(b) **On-line simulation**, as in rollout algorithms, where we use a suboptimal policy \( \mu \) to compute on-line when needed the values \( \hat{J}(x) \) to be exactly or approximately equal to \( J_\mu(x) \). The policy \( \mu \) may be obtained by any method, e.g., one based on heuristic reasoning, 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. This may be an important consideration for the practical implementation of rollout algorithms, particularly for stochastic problems.

(c) **On-line approximate optimization**, such as model predictive control (MPC), which will be discussed in more detail later. This approach involves the solution of a suitably constructed \( \ell \)-step version of the problem. It can be viewed as either approximation in value space with \( \ell \)-step lookahead, or as a form of rollout algorithm.

(d) **Parametric cost approximation**, where \( \hat{J} \) 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.
We refer to the neurodynamic programming book by Bertsekas and Tsitsiklis [BeT96], and the RL book by Sutton and Barto [SuB18], as well as the large number of subsequent RL and approximate DP books, which provide specific examples of cost function approximation methods and associated training algorithms.

**From Off-Line Training to On-Line Play**

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. 2.2. Let us now consider some additional possibilities, some of which involve the use of a policy $\mu$ 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 $\mu$ that has been obtained off-line, we may use as terminal cost approximation $\tilde{J}$ the cost function $J_{\mu}$ of the policy. This requires a policy evaluation operation, and can be done on-line, by computing (possibly by simulation) just the values of

$$E\left\{ J_{\mu}(f(x_k, u_k, w_k)) \right\}$$

that are needed for the case of one-step lookahead [cf. Eq. (2.17)], or the values of

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

that are needed for the case of $\ell$-step lookahead. This is the simplest form of rollout, and only requires the off-line construction of the policy $\mu$.

(b) Given a terminal cost approximation $\tilde{J}$ that has been obtained off-line, we may use it on-line to compute a one-step or multistep lookahead policy $\tilde{\mu}$. This policy can in turn be used for rollout as in (a) above. In a variation of this scheme, we may also use $\tilde{J}$ for truncated rollout, to approximate the tail end of the rollout process (an example of this is the rollout-based TD-Gammon algorithm discussed in Section 1.2).

(c) Given a policy $\mu$ 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 $\mu$ is approximated using the cost approximation $\tilde{J}$. This is similar to the truncated rollout scheme noted in (b) above, except that the policy $\mu$ is computed off-line rather than on-line using $\tilde{J}$ and one-step or multistep lookahead as in (b).

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 Section 5.2. Other examples of this type will be noted in the context of specific applications.

3. AN ABSTRACT VIEW OF REINFORCEMENT LEARNING

In this section we will use geometric constructions to obtain insight into Bellman’s equation, the value and policy iteration algorithms, approximation in value space, and some of the properties of the corresponding one-step or multistep lookahead policy \( \tilde{\mu} \). To understand these constructions, we need an abstract notational framework. In particular, we denote by \( TJ \) the function of \( x \) that appears in the right-hand side of Bellman’s equation. Its value at state \( x \) is given by†

\[
(TJ)(x) = \min_{u \in U(x)} E \left\{ g(x, u, w) + \alpha J(f(x, u, w)) \right\}, \quad \text{for all } x. \tag{3.1}
\]

Also for each policy \( \mu \), we introduce the corresponding function \( T_\mu J \), which has value at \( x \) given by

\[
(T_\mu J)(x) = E \left\{ g(x, \mu(x), w) + \alpha J(f(x, \mu(x), w)) \right\}, \quad \text{for all } x. \tag{3.2}
\]

Thus \( T \) and \( T_\mu \) can be viewed as operators (broadly referred to as the Bellman operators), which map functions \( J \) to other functions (\( TJ \) or \( T_\mu J \), respectively).‡

An important property of the operators \( T \) and \( T_\mu \) is that they are monotone, in the sense that if \( J \) and \( J' \) are two functions of \( x \) such that

\[ J(x) \geq J'(x), \quad \text{for all } x, \]

then we have

\[
(TJ)(x) \geq (TJ')(x), \quad (T_\mu J)(x) \geq (T_\mu J')(x), \quad \text{for all } x \text{ and } \mu. \tag{3.3}
\]

This is evident from Eqs. (3.1) and (3.2).

† Recall here our convention that we will be using “min” instead of the more formal “inf,” even we are not sure that minimum is attained.
‡ Within the context of this work, the functions \( J \) on which \( T \) and \( T_\mu \) operate will be real-valued functions of \( x \). We will assume throughout that the expected values in Eqs. (3.1) and (3.2) are well-defined and finite when \( J \) is real-valued. This implies that \( T_\mu J \) will also be real-valued functions of \( x \). On the other hand \( (TJ)(x) \) may take the value \(-\infty\) because of the minimization in Eq. (3.1). We allow this possibility, although our illustrations will primarily depict the case where \( TJ \) is real-valued. Note that the general theory of abstract DP is developed with the use of extended real-valued functions; see [Ber18a].
Another important property is that the Bellman operator $T_\mu$ is linear, in the sense that it has the form $T_\mu J = G + A_\mu J$, where $G \in R(X)$ is some function and $A_\mu : R(X) \mapsto R(X)$ is an operator such that for any functions $J_1, J_2$, and scalars $\gamma_1, \gamma_2$, we have††

$$A_\mu(\gamma_1 J_1 + \gamma_2 J_2) = \gamma_1 A_\mu J_1 + \gamma_2 A_\mu J_2.$$ 

Moreover, from the definitions (3.1) and (3.2), we have

$$(T J)(x) = \min_{\mu \in \mathcal{M}} (T_\mu J)(x), \quad \text{for all } x,$$

where $\mathcal{M}$ is the set of stationary policies. It follows that $(T J)(x)$ is a concave function of $J$ for every $x$, something that will be important for our interpretation of one-step and multistep lookahead as a Newton iteration for solving the Bellman equation $J = T J$.

**Example 3.1 (A Two-State and Two-Control Example)**

Assume that there are two states 1 and 2, and two controls $u$ and $v$. Consider the policy $\mu$ that applies control $u$ at state 1 and control $v$ at state 2. Then the operator $T_\mu$ takes the form

$$(T_\mu J)(1) = \sum_{y=1}^{2} p_{1y}(u) \bigl( g(1, u, y) + \alpha J(y) \bigr), \quad \text{and} \quad (T_\mu J)(2) = \sum_{y=1}^{2} p_{2y}(v) \bigl( g(2, v, y) + \alpha J(y) \bigr),$$

where $p_{xy}(u)$ and $p_{xy}(v)$ are the probabilities that the next state will be $y$, when the current state is $x$, and the control is $u$ or $v$, respectively. Clearly, $(T_\mu J)(1)$ and $(T_\mu J)(2)$ are linear functions of $J$. Also the operator $T$ of the Bellman equation $J = TJ$ takes the form

$$(T J)(1) = \min \left[ \sum_{y=1}^{2} p_{1y}(u) \bigl( g(1, u, y) + \alpha J(y) \bigr) \sum_{y=1}^{2} p_{1y}(v) \bigl( g(1, v, y) + \alpha J(y) \bigr) \right], \quad \text{and} \quad (T J)(2) = \min \left[ \sum_{y=1}^{2} p_{2y}(u) \bigl( g(2, u, y) + \alpha J(y) \bigr) \sum_{y=1}^{2} p_{2y}(v) \bigl( g(2, v, y) + \alpha J(y) \bigr) \right].$$

Thus, $(T J)(1)$ and $(T J)(2)$ are concave and piecewise linear as functions of the two-dimensional vector $J$ (with two pieces; more generally, as many linear pieces as the number of controls). This concavity property holds in general since $(T J)(x)$ is the minimum of a collection of linear functions of $J$, one for each $u \in U(x)$. Figure 3.1 illustrates $(T_\mu J)(1)$ for the cases where $\mu(1) = u$ and $\mu(1) = v$, $(T_\mu J)(2)$ for the cases where $\mu(2) = u$ and $\mu(2) = v$, $(T J)(1)$, and $(T J)(2)$, as functions of $J = (J(1), J(2))$.

†† An operator $T_\mu$ with this property is often called “affine,” but in this work we just call it “linear.” Also we use abbreviated notation to express pointwise equalities and inequalities, so that we write $J = J'$ or $J \geq J'$ to express the fact that $J(x) = J'(x)$ or $J(x) \geq J'(x)$, for all $x$, respectively.
Mathematically the concavity property of $T$ manifests itself in that the set

$$ C = \{ (J, \xi) \in R(X) \times R(X) \mid (TJ)(x) \geq \xi(x), \text{ for all } x \in X \} \quad (3.8) $$

is convex as a subset of $R(X) \times R(X)$, where $R(X)$ is the set of real-valued functions over the state space $X$. 

Figure 3.1  Geometric illustrations of the Bellman operators $T(\mu)$ and $T$ for states 1 and 2 in Example 3.1; cf. Eqs. (3.4)-(3.7). The problem’s transition probabilities are: $p_{11}(u) = 0.3$, $p_{12}(u) = 0.7$, $p_{21}(u) = 0.4$, $p_{22}(u) = 0.6$, $p_{11}(v) = 0.6$, $p_{12}(v) = 0.4$, $p_{21}(v) = 0.9$, $p_{22}(v) = 0.1$. The stage costs are $g(1, u, 1) = 3$, $g(1, u, 2) = 10$, $g(2, u, 1) = 0$, $g(2, u, 2) = 6$, $g(1, v, 1) = 7$, $g(1, v, 2) = 5$, $g(2, v, 1) = 3$, $g(2, v, 2) = 12$. The discount factor is $\alpha = 0.9$, and the optimal costs are $J^*(1) = 50.59$ and $J^*(2) = 47.41$. The optimal policy is $\mu^*(1) = v$ and $\mu^*(2) = u$. The figure also shows the one-dimensional slices of $T$ that pass through $J^*$. 
This convexity property is verified by showing that given \((J_1, \xi_1)\) and \((J_2, \xi_2)\) in \(C\), and \(\gamma \in [0, 1]\), we have
\[
(\gamma J_1 + (1 - \gamma)J_2, \gamma \xi_1 + (1 - \gamma)\xi_2) \in C.
\]
The proof of this is straightforward by using the concavity of \((T J)(x)\) for each \(x\).

Critical properties from the DP point of view are whether \(T\) and \(T\mu\) have fixed points; equivalently, whether the Bellman equations \(J = T J\) and \(J = T\mu J\) have solutions within the class of real-valued functions, and whether the set of solutions includes \(J^*\) and \(J\mu\), respectively. It may thus be important to verify that \(T\) or \(T\mu\) are contraction mappings. This is true for example in the benign case of discounted problems with bounded cost per stage. However, for undiscounted problems, asserting the contraction property of \(T\) or \(T\mu\) may be more complicated, and even impossible; the abstract DP book [Ber18a] deals extensively with such questions, and related issues regarding the solution sets of the Bellman equations.

**Geometrical Interpretations**

We will now interpret the Bellman operators geometrically, starting with \(T\mu\). Figure 3.2 illustrates its form. Note here that the functions \(J\) and \(T\mu J\) are multidimensional. They have as many scalar components \(J(x)\) and \((T\mu J)(x)\), respectively, as there are states \(x\), but they can only be shown projected onto one dimension. The function \(T\mu J\) for each policy \(\mu\) is linear. The cost function \(J\mu\) satisfies \(J\mu = T\mu J\mu\), so it is obtained from the intersection of the graph of \(T\mu J\) and the 45 degree line, when \(J\mu\) is real-valued. Later we will interpret the situation where \(J\mu\) is not real-valued with lack of system stability under \(\mu\) [we have \(J\mu(x) = \infty\) for some initial states \(x\)].

The form of the Bellman operator \(T\) is illustrated in Fig. 3.3. Again the functions \(J, J^*, T J, T\mu J, \) etc, are multidimensional, but they are shown projected onto one dimension. The Bellman equation \(J = T J\) may have one or many real-valued solutions. It may also have no real-valued solution in exceptional situations, as we will discuss later (see Section 3.8). The figure assumes a unique real-valued solution of the Bellman equations \(J = T J\) and \(J = T\mu J\), which is true if \(T\) and \(T\mu\) are contraction mappings, as is the case for discounted problems with bounded cost per stage. Otherwise, these equations may have no solution or multiple solutions within the class of real-valued functions (see Section 3.8). The equation \(J = T J\) typically has \(J^*\) as a solution, but may have more than one solution in cases where either \(\alpha = 1\) or \(\alpha < 1\), and the cost per stage is unbounded.

**Example 3.2 (A Two-State and Infinite Controls Problem)**

Let us consider the mapping \(T\) for a problem that involves two states, 1 and 2, but an infinite number of controls. In particular, the control space at both states is the unit interval, \(U(1) = U(2) = [0, 1]\). Here \((T J)(1)\) and \((T J)(2)\) are given by
\[
(T J)(1) = \min_{u \in [0, 1]} \left\{ g_1 + r_{11} u^2 + r_{12} (1 - u)^2 + \alpha u J(1) + \alpha (1 - u) J(2) \right\},
\]

24
Figure 3.2 Geometric interpretation of the linear Bellman operator $T_\mu$ and the corresponding Bellman equation. The graph of $T_\mu$ is a plane in the space $\mathbb{R}(X) \times \mathbb{R}(X)$, and when projected on a one-dimensional plane that corresponds to a single state and passes through $J_\mu$, it becomes a line. Then there are three cases:

(a) The line has slope less than 45 degrees, so it intersects the 45-degree line at a unique point, which is equal to $J_\mu$, the solution of the Bellman equation $J = T_\mu J$. This is true if $T_\mu$ is a contraction mapping, as is the case for discounted problems with bounded cost per stage.

(b) The line has slope greater than 45 degrees. Then it intersects the 45-degree line at a unique point, which is a solution of the Bellman equation $J = T_\mu J$, but is not equal to $J_\mu$. Then $J_\mu$ is not real-valued; we will call such $\mu$ unstable under $\mu$ in Section 3.2.

(c) The line has slope exactly equal to 45 degrees. This is an exceptional case where the Bellman equation $J = T_\mu J$ has an infinite number of real-valued solutions or no real-valued solution at all; we will provide examples where this occurs in Section 3.8.

\[
(TJ)(2) = \min_{u \in [0,1]} \left\{ g_2 + r_{21} u^2 + r_{22} (1-u)^2 + \alpha u J(1) + \alpha (1-u) J(2) \right\}.
\]

The control $u$ at each state $x = 1, 2$ has the meaning of a probability that we must select at that state. In particular, we control the probabilities $u$ and $(1-u)$ of moving to states $y = 1$ and $y = 2$, at a control cost that is quadratic in $u$ and $(1-u)$, respectively. For this problem $(TJ)(1)$ and $(TJ)(2)$ can be calculated in closed
Figure 3.3 Geometric interpretation of the Bellman operator $T$, and the corresponding Bellman equation. For a fixed $x$, the function $(TJ)(x)$ can be written as $\min_{\mu} (T_{\mu}J)(x)$, so it is concave as a function of $J$. The optimal cost function $J^*$ satisfies $J^* = T^*J^*$, so it is obtained from the intersection of the graph of $TJ$ and the 45 degree line shown, assuming $J^*$ is real-valued.

Note that the graph of $T$ lies below the graph of every operator $T_{\mu}$, and is in fact obtained as the lower envelope of the graphs of $T_{\mu}$ as $\mu$ ranges over the set of policies $\mathcal{M}$. In particular, for any given function $\tilde{J}$, for every $x$, the value $(T_{\tilde{J}})(x)$ is obtained by finding a support hyperplane/subgradient of the graph of the concave function $(TJ)(x)$ at $J = \tilde{J}$, as shown in the figure. This support hyperplane is defined by the control $\mu(x)$ of a policy $\tilde{\mu}$ that attains the minimum of $(T_{\mu}\tilde{J})(x)$ over $\mu$:

$$\tilde{\mu}(x) \in \arg \min_{\mu \in \mathcal{M}} (T_{\mu}\tilde{J})(x)$$

(there may be multiple policies attaining this minimum, defining multiple support hyperplanes).

This construction also shows how the minimization

$$(T\tilde{J})(x) = \min_{\mu \in \mathcal{M}} (T_{\mu}\tilde{J})(x)$$

corresponds to a linearization of the mapping $T$ at the point $\tilde{J}$.

Form, so they are easy to plot and understand. They are piecewise quadratic, unlike the corresponding plots of Fig. 3.1, which are piecewise linear; see Fig. 3.4.
Figure 3.4 Illustration of the Bellman operator $T$ for states 1 and 2 in Example 3.2. The parameter values are $g_1 = 5$, $g_2 = 3$, $r_{11} = 3$, $r_{12} = 15$, $r_{21} = 9$, $r_{22} = 1$, and the discount factor is $\alpha = 0.9$. The optimal costs are $J^*(1) = 49.7$ and $J^*(2) = 40.0$, and the optimal policy is $\mu^*(1) = 0.59$ and $\mu^*(2) = 0$. The figure also shows the one-dimensional slices of the operators at $J(1) = 15$ and $J(2) = 30$, together with the corresponding 45-degree lines.

Visualization of Value Iteration

The operator notation simplifies algorithmic descriptions, derivations, and proofs related to DP. For example, we can write the VI algorithm in the compact form

$$J_{k+1} = T J_k, \quad k = 0, 1, \ldots,$$

as illustrated in Fig. 3.5. Moreover, the VI algorithm for a given policy $\mu$ can be written as

$$J_{k+1} = T_\mu J_k, \quad k = 0, 1, \ldots,$$

and it can be similarly interpreted, except that the graph of the function $T_\mu J$ is linear. Also we will see shortly that there is a similarly compact description for the policy iteration algorithm.

To keep the presentation simple, we will focus our attention on the abstract DP framework as it applies to the optimal control problems of Section 2.1. In particular, we will assume without further mention that $T$ and $T_\mu$ have the monotonicity property (3.3), that $T_\mu J$ is linear for all $\mu$, and that (as a consequence) the component $(T J)(x)$ is concave as a function of $J$ for every state $x$. We note, however, that the abstract notation facilitates the extension of the infinite horizon DP theory to models beyond the ones that we discuss in this work. Such models include semi-Markov problems, minimax control problems, risk sensitive problems, Markov games, and others (see the DP textbook [Ber12], and the abstract DP monograph [Ber18a]).
3.1 Approximation in Value Space and Newton’s Method

Let us now consider approximation in value space and an abstract geometric interpretation, first provided in the author’s book [Ber20a]. By using the operators $T$ and $T_{\hat{\mu}}$, for a given $\tilde{J}$, a one-step lookahead policy $\tilde{\mu}$ is characterized by the equation

$$T_{\tilde{\mu}}\tilde{J} = T\tilde{J},$$

as in Fig. 3.6 [cf. Eq. (2.17)]. Furthermore, this equation implies that the graph of $T_{\tilde{\mu}}J$ just touches the graph of $TJ$ at $\tilde{J}$, as shown in the figure. In mathematical terms, the set

$$C_{\tilde{\mu}} = \{(J,\xi) \mid T_{\tilde{\mu}}J \geq \xi\},$$

contains the convex set $C$ of Eq. (3.8) (since $TJ \geq \xi$ implies that $T_{\tilde{\mu}}J \geq \xi$), and has a common point $(\tilde{J}, T_{\tilde{\mu}}\tilde{J})$ with $C$. Moreover, for each state $x \in X$ the hyperplane $H_{\tilde{\mu}}(x)$

$$H_{\tilde{\mu}}(x) = \left\{(J(x),\xi(x)) \mid (T_{\tilde{\mu}}J)(x) = \xi(x)\right\},$$

supports from above the convex set

$$\left\{(J(x),\xi(x)) \mid (TJ)(x) \geq \xi(x)\right\}.$$
Figure 3.6 Geometric interpretation of approximation in value space and the one-step lookahead policy \( \tilde{\mu} \) as a step of Newton’s method [cf. Eq. (2.17)]. Given \( \tilde{J} \), we find a policy \( \tilde{\mu} \) that attains the minimum in the relation \( TJ = \min_\mu T\tilde{\mu}\tilde{J} \). This policy satisfies \( TJ = T\tilde{\mu}J \), so the graph of \( TJ \) and \( T\tilde{\mu}J \) touch at \( \tilde{J} \), as shown. It may not be unique. Because \( TJ \) has concave components, the equation \( J = T\tilde{\mu}J \) is the linearization of the equation \( J = TJ \) at \( \tilde{J} \). The linearized equation is solved at the typical step of Newton’s method to provide the next iterate, which is just \( J_{\tilde{\mu}} \).

at the point \( (\tilde{J}(x), (T\tilde{J})(x)) \) and defines a subgradient of \( (TJ)(x) \) at \( \tilde{J} \). Note that the one-step lookahead policy \( \tilde{\mu} \) need not be unique, since \( T \) need not be differentiable.

Thus, the equation \( J = T\tilde{\mu}J \) is a pointwise (for each \( x \)) linearization of the equation \( J = TJ \) at \( \tilde{J} \), and its solution, \( J_{\tilde{\mu}} \), can be viewed as the result of a Newton iteration at the point \( \tilde{J} \). In summary, the Newton
iterate at $\tilde{J}$ is $J_\mu$, the solution of the linearized equation $J = T_\mu J$.

As noted earlier, approximation in value space with $\ell$-step lookahead using $\tilde{J}$ is the same as approximation in value space with one-step lookahead using the $(\ell - 1)$-fold operation of $T$ on $\tilde{J}$, $T^{\ell - 1}\tilde{J}$. Thus it can be interpreted as a Newton step starting from $T^{\ell - 1}\tilde{J}$, the result of $\ell - 1$ value iterations applied to $\tilde{J}$. This is illustrated in Fig. 3.7. In this connection, we note that several variants of Newton’s method that involve combinations of first-order iterative methods, such as the Gauss-Seidel and Jacobi algorithms, and Newton’s method, are well-known in numerical analysis. They belong to the general family of Newton-SOR methods (SOR stands for “successive over-relaxation”); see the classic book by Ortega and Rheinboldt [OrR70] (Section 13.4). Their convergence rate is superlinear, similar to Newton’s method, as long as they involve a pure Newton step, along with the first-order steps.

3.2 Region of Stability

For any control system design method, the stability of the policy obtained is of paramount importance. It is thus essential to investigate and verify the stability of controllers obtained through approximation in value space schemes. Historically, there have been several proposed definitions of stability in control theory. Within the context of this work, our focus on stability issues will be for problems with a termination state

$\dagger$ The classical 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 $\frac{\partial T(y_k)}{\partial y}$ is the $n \times n$ Jacobian matrix of $T$ evaluated at the vector $y_k$. The most commonly given convergence rate property of Newton’s method is quadratic convergence. It states that near the solution $y^*$, we have

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

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 superlinear convergence property of the method.

The structure of the Bellman operators (3.1) and (3.2), with their monotonicity and concavity properties, tends to enhance the convergence and rate of convergence properties of Newton’s method, even in the absence of differentiability, as evidenced by the convergence analysis of PI, and the extensive favorable experience with rollout, PI, and MPC. In this connection, it is worth noting that in the case of Markov games, where the concavity property does not hold, the PI method may oscillate, as shown by Pollatschek and Avi-Itzhak [PoA69], and needs to be modified to restore its global convergence; see the author’s paper [Ber21c].
$t$, which is cost-free, and with a cost per stage that is positive outside the termination state, such as the undiscounted positive cost deterministic problem introduced earlier [cf. Eqs. (2.2)-(2.4)]. Moreover, it is best for our purposes to adopt an optimization-based definition. In particular, we say that a policy $\mu$ is unstable if $J_\mu(x) = \infty$ for some states $x$. Equivalently, the policy $\mu$ is called stable if $J_\mu(x) < \infty$ for all states $x$. This definition has the advantage that it applies to general state and control spaces. Naturally, it can be made more specific in particular problem instances.\footnote{For the undiscounted positive cost deterministic problem introduced earlier [cf. Eqs. (2.2)-(2.4)], it can be shown that if a policy $\mu$ is stable, then $J_\mu$ is the “smallest” solution of the Bellman equation $J = T_\mu J$ within the class of nonnegative real-valued functions, and under mild assumptions it is the unique solution of $J = T_\mu J$ within the class of nonnegative real-valued functions $J$ with $J(t) = 0$; see the author’s paper [Ber17b]. Moreover, if $\mu$ is unstable, then the Bellman equation $J = T_\mu J$ has no solution within the class of nonnegative real-valued functions.}

In the context of approximation in value space we are interested in the region of stability, which is the set of cost approximations $\tilde{J} \in R(X)$ for which all the corresponding one-step or multistep lookahead policies $\tilde{\mu}$ are stable. For discounted problems with bounded cost per stage, all policies have real-valued cost functions, so questions of stability do not arise. In general, however, the region of stability may be a strict subset of the set of real-valued functions; this will be illustrated later for the undiscounted deterministic case of the linear quadratic problem of Section 2.1 [cf. Eqs. (2.5), (2.6)].

Figure 3.8 illustrates the region of stability for approximation in value space with one-step lookahead. An important fact is that the region of stability includes all real-valued nonnegative functions $\tilde{J}$ such that

$$T \tilde{J} \leq \tilde{J}.$$ 

Indeed if $\tilde{\mu}$ is the corresponding one-step lookahead policy, we have $T_{\tilde{\mu}} \tilde{J} = T \tilde{J} \leq \tilde{J}$, and from a well-known result on nonnegative cost infinite horizon problems [see [Ber12], Prop. 4.1.4(a)], we have

$$J_{\tilde{\mu}} \leq \tilde{J}.$$ 

Thus if $\tilde{J}$ is nonnegative and real-valued, $J_{\tilde{\mu}}$ is also real-valued, so $\tilde{\mu}$ is stable. It follows that $\tilde{J}$ belongs to the region of stability. This is a known result in specific contexts, such as MPC (see the book by Rawlings, Mayne, and Diehl [RMD17], Section 2.4, which contains extensive references to prior work on stability issues).

An important consequence of the preceding result is that in view of the fixed point equation $J_\mu = T_\mu J_\mu$, which implies that $J_\mu \geq TJ_\mu$, the region of stability includes the cost functions $J_\mu$ of all stable policies $\mu$. In particular, rollout with a stable policy provides a starting point $J_\mu$ for approximation in value space, which lies within the region of stability. This is also a critical property for the success of the PI algorithm, which consists of repeated application of rollout, and hence generates a sequence of stable policies if started with a stable initial policy.
Figure 3.8 Illustration of the regions of stability and instability for approximation in value space with one-step lookahead. The stability region is the set of all $J$ such that the policy $\tilde{\mu}$ with $T\tilde{J} = T\tilde{\mu}J$ satisfies $J_{\tilde{\mu}}(x) < \infty$ for all $x$.

An interesting observation from Fig. 3.8 is that if $\tilde{J}$ does not belong to the region of stability and $\tilde{\mu}$ is a corresponding one-step lookahead unstable policy, the Bellman equation $J = T\tilde{\mu}J$ may have real-valued solutions. However, these solutions will not be equal to $J_{\tilde{\mu}}$, as this would violate the definition of region of stability. Generally, if $T_{\mu}$ is not a contraction mapping, $T_{\mu}J$ may have real-valued fixed points, none of which is equal to $J_{\mu}$.

Figure 3.9 illustrates the region of stability for the case of multistep lookahead. The insights from this figure are similar to the one-step lookahead case of Fig. 3.8. However, the figure indicates that the region of stability of the $\ell$-step lookahead controller $\tilde{\mu}$ depends on $\ell$, and tends to become larger as $\ell$ increases. The reason is that $\ell$-step lookahead with terminal cost $\tilde{J}$ is equivalent to one-step lookahead with terminal cost $T^{\ell-1}\tilde{J}$, which tends to be closer to the optimal cost function $J^*$ than $\tilde{J}$ (assuming convergence of the VI method). A conjecture that should hold under reasonable conditions is that if the VI algorithm $J_k = T_k\tilde{J}$ converges to $J^*$ then $T^{\ell-1}\tilde{J}$ belongs to the region of stability for sufficiently large $\ell$. Related ideas have been discussed in the adaptive DP literature by Liu and his collaborators [HHL21], [LZZ21], [WLL16], and by Heydari [Hey17], [Hey18], who provide extensive references; see also Winnicki et al. [WLL21]. We will revisit this issue in the context of linear quadratic problems. A related conjecture is that given a stable policy $\mu$, $T^{m}\tilde{J}$ belongs to the region of stability for sufficiently large $m$.  

32
3.3 Policy Iteration, Rollout, and Newton’s Method

Another major class of infinite horizon algorithms is based on policy iteration (PI for short), which involves the repeated use of policy improvement, in analogy with the AlphaZero/TD-Gammon off-line training algorithms, described in Section 1. Each iteration of the PI algorithm starts with a stable policy (which we call current or base policy), and generates another stable policy (which we call new or rollout policy, respectively). For the infinite horizon problem of Section 2.1, given the base policy $\mu$, the iteration consists of two phases (see Fig. 3.10):

(a) Policy evaluation, which computes the cost function $J_\mu$. One possibility is to solve the corresponding Bellman equation

$$J_\mu(x) = E\left\{g(x, \mu(x), w) + \alpha J_\mu(f(x, \mu(x), w))\right\}, \quad \text{for all } x. \quad (3.9)$$

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 $x$. Other, more sophisticated possibilities include the use of specialized simulation-based methods, such as temporal difference methods, for which there is extensive literature (see e.g., the books [BeT96], [SuB98], [Ber12]).

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

$$\tilde{\mu}(x) \in \arg \min_{u \in U(x)} E\left\{g(x, u, w) + \alpha J_\mu(f(x, u, w))\right\}, \quad \text{for all } x. \quad (3.10)$$
It is generally expected (and can be proved under mild conditions) that the rollout policy is improved in the sense that $J_{\hat{\mu}}(x) \leq J_{\mu}(x)$ for all $x$.

Thus PI generates a sequence of stable policies $\{\mu^k\}$, by obtaining $\mu^{k+1}$ through a policy improvement operation using $J_{\mu^k}$ in place of $J_{\mu}$ in Eq. (3.10), which is obtained through policy evaluation of the preceding policy $\mu^k$ using Eq. (3.9). It is well known that PI has solid convergence properties, which hold even when the method is implemented (with appropriate modifications) in unconventional computing environments, involving asynchronous distributed computation, as shown in a series of papers by Bertsekas and Yu [BeY10], [BeY12], [YuB13].

In terms of our abstract notation, the PI algorithm can be written in a compact form. For the generated policy sequence $\{\mu^k\}$, the policy evaluation phase obtains $J_{\mu^k}$ from the equation

$$J_{\mu^k} = T_{\mu^k}J_{\mu^k},$$

(3.11)

while the policy improvement phase obtains $\mu^{k+1}$ through the equation

$$T_{\mu^{k+1}}J_{\mu^k} = TJ_{\mu^k}.$$  

(3.12)

As Fig. 3.11 illustrates, PI can be viewed as Newton’s method for solving the Bellman equation in the function space of cost functions $J$. In particular, the policy improvement Eq. (3.12) is the Newton step starting from $J_{\mu^k}$, and yields $\mu^{k+1}$ as the corresponding one-step lookahead/rollout policy. Figure 3.12 illustrates the rollout algorithm, which is just the first iteration of PI.

In contrast to approximation in value space, the interpretation of PI in terms of Newton’s method has a long history. We refer to the original works for linear quadratic problems by Kleinman [Klei68],† and for finite-state infinite horizon discounted and Markov game problems by Pollatschek and Avi-Itzhak [PoA69] (who

† This was part of Kleinman’s Ph.D. thesis [Kle67] at M.I.T., supervised by M. Athans. Kleinman gives credit for the one-dimensional version of his results to Bellman and Kalaba [BeK65]. Note also that the first proposal of the PI method was given by Bellman in his classic book [Be57], under the name “approximation in policy space.”
Figure 3.11 Geometric interpretation of a policy iteration. Starting from the stable current policy $\mu^k$, it evaluates the corresponding cost function $J_{\mu^k}$, and computes the next policy $\mu^{k+1}$ according to

$$T_{\mu^{k+1}}J_{\mu^k} = TJ_{\mu^k}.$$

The corresponding cost function $J_{\mu^{k+1}}$ is obtained as the solution of the linearized equation $J = T_{\mu^{k+1}}J$, so it is the result of a Newton step for solving the Bellman equation $J = TJ$, starting from $J_{\mu^k}$. Note that in policy iteration, the Newton step always starts at a function $J_{\mu^k}$, which satisfies $J_{\mu^k} \geq J^*$ as well as $TJ_{\mu^k} \leq J_{\mu^k}$ (cf. our discussion on stability in Section 3.2).

Rollout

Generally, rollout with a stable base policy $\mu$ can be viewed as a single iteration of Newton’s method starting...
from $J_\mu$, as applied to the solution of the Bellman equation (see Fig. 3.12). Because rollout is applied only once at each stage during the real-time operation of the system, it is well suited for on-line implementation, provided that the policy evaluation of the base policy can be done on-line as needed. This often involves deterministic or stochastic on-line simulation from each of the states $x_k$ generated by the system in real time.

As Fig. 3.12 illustrates, the cost function of the rollout policy $J_\mu$ is obtained by constructing a linearized version of Bellman’s equation at $J_\mu$ (its linear approximation at $J_\mu$), and then solving it. If the function $TJ$ is nearly linear (i.e., has small “curvature”) the rollout policy performance $J_\mu(x)$ is very close to the optimal $J^*(x)$, even if the base policy $\mu$ is far from optimal. This explains the large cost improvements that are typically observed in practice with the rollout algorithm.

An interesting question is how to compare the rollout performance $J_\mu(x)$ for a given initial state $x$, with the base policy performance $J_\mu(x)$. Clearly, we would like $J_\mu(x) - J_\mu(x)$ to be large, but this is not the right way to look at cost improvement. The reason is that $J_\mu(x) - J_\mu(x)$ will be small if its upper bound,
J_\mu(x) - J^*(x), is small, i.e., if the base policy is close to optimal. What is important is that the error ratio

$$\frac{J_\mu(x) - J^*(x)}{J_\mu(x) - J_\tilde{\mu}(x)}$$

is small. Indeed, this ratio becomes smaller as J_\mu(x) - J^*(x) approaches 0 because of the superlinear convergence rate of Newton’s method that underlies the rollout algorithm (cf. Fig. 3.12). Unfortunately, it is hard to evaluate this ratio, since we do not know J^*(x). On the other hand, we should not be underwhelmed if we observe a small performance improvement J_\mu(x) - J_\tilde{\mu}(x): the reason may be that the base policy is already near-optimal, and in fact we are likely doing very well in terms of the ratio (3.13).

**Truncated Rollout**

Variants of rollout may involve multistep lookahead, truncation, and terminal cost function approximation, as in the case of AlphaZero/TD-Gammon, cf. Section 1. These variants admit geometric interpretations that are similar to the ones given earlier; see Fig. 3.13. Truncated rollout uses m VIs with the base policy \( \mu \) and a terminal cost function approximation \( \tilde{J} \) to approximate the cost function \( J_\mu \).

In the case of one-step lookahead, the truncated rollout policy \( \tilde{\mu} \) is defined by

$$T_\tilde{\mu}(T_\mu^m \tilde{J}) = T(T_\mu^m \tilde{J}),$$
i.e., $\tilde{\mu}$ attains the minimum when the Bellman operator $T$ is applied to the function $T^m_\mu \tilde{J}$ (the cost obtained by using the base policy $\mu$ for $m$ steps followed by terminal cost approximation $\tilde{J}$); see Fig. 3.13. In the case of $\ell$-step lookahead, the truncated rollout policy $\tilde{\mu}$ is defined by

$$T_{\tilde{\mu}}(T^{\ell-1}_\mu T^m_\mu \tilde{J}) = T(T^{\ell-1}_\mu T^m_\mu \tilde{J}).$$

Truncated rollout is related to a variant of PI called optimistic. This variant approximates the policy evaluation step by using $m$ value iterations using the base policy $\mu$; see [BeT96], [Ber12], [Ber19a] for a more detailed discussion of this relation.

As noted earlier, variants of Newton’s method that involve multiple fixed point iterations, before and after each Newton step, but without truncated rollout, i.e.,

$$T_{\tilde{\mu}}(T^{\ell-1}\tilde{J}) = T(T^{\ell-1}\tilde{J}),$$

are well-known. The numerical analysis book by Ortega and Rheinboldt [OrR70] (Sections 13.3 and 13.4) provides various convergence results, under assumptions that include differentiability and convexity of the components of $T$, and nonnegativity of the inverse Jacobian of $T$. These assumptions, particularly differentiability, may not be satisfied within our DP context. Moreover, for methods of the form (3.14), the initial point must satisfy an additional assumption, which ensures that the convergence to $J^*$ is monotonic from above (in this case, if in addition the Jacobian of $T$ is isotone, an auxiliary sequence can be constructed that converges to $J^*$ monotonically from below; see [OrR70], 13.3.4, 13.4.2). This is similar to existing convergence results for the optimistic PI method in DP; see e.g., [BeT96], [Ber12].

Geometrical interpretations such as the ones of Fig. 3.13 suggest, among others, that:

(a) The cost improvement $J_\mu - J_{\tilde{\mu}}$, from base to rollout policy, tends to become larger as the length $\ell$ of the lookahead increases.

(b) Truncated rollout with $\ell$-step lookahead minimization, followed by $m$ steps of a base policy $\mu$, and then followed by terminal cost function approximation $\tilde{J}$ may be viewed, under certain conditions, as an economic alternative to $(\ell + m)$-step lookahead minimization using $\tilde{J}$ as terminal cost function approximation.

We next discuss the issues of selection of $\ell$ and $m$.

**Lookahead Length Issues in Truncated Rollout**

A question of practical interest is how to choose the lookahead lengths $\ell$ and $m$ in truncated rollout schemes. It is clear that large values $\ell$ for lookahead minimization are beneficial (in the sense of producing improved lookahead policy cost functions $J_{\tilde{\mu}}$), since additional VI iterations bring closer to $J^*$ the starting point $T^{\ell-1}\tilde{J}$.
of the Newton step. On the other hand, large values $m$ for truncated rollout bring the starting point for the Newton step closer to $J_\mu$, and not necessarily closer to $J^*$. Indeed computational experiments suggest that increasing values for $m$ may be counterproductive beyond some threshold, and that this threshold generally depends on the problem and the terminal cost approximation $\hat{J}$. This is also consistent with long standing experience with optimistic policy iteration, which bears a close connection with truncated rollout, as noted earlier. Unfortunately, however, there is no analysis that can illuminate this issue, and the available error bounds for truncated rollout (see [Ber19a], [Ber20a]) are conservative and provide limited guidance in this regard.

An interesting property, which holds in some generality, is that truncated rollout with a stable policy has a beneficial effect on the stability properties of the lookahead policy. The reason is that the cost function $J_\mu$ of the base policy $\mu$ lies well inside the region of stability, as noted in Section 3.2. Moreover value iterations with $\mu$ (i.e., truncated rollout) tend to push the starting point of the Newton step towards $J_\mu$. Thus a sufficient number of these value iterations will bring the starting point of the Newton step within the region of stability. An important fact to keep in mind is that the truncated rollout steps are much less demanding computationally than the lookahead minimization steps. In particular, large values of $m$ may be computationally tolerable, but even relatively small values of $m$ can be computationally prohibitive. This is especially true for stochastic problems where the width of the lookahead tree tends to grow quickly. Thus, with other concerns weighing equally, it is computationally preferable to use large values of $m$ rather than large values of $\ell$ (this was the underlying motivation for truncated rollout in Tesauro’s TD-Gammon [TeG96]).

The preceding discussion suggests the following qualitative question: is lookahead by rollout an economic substitute for lookahead by minimization? The answer to this seems to be a qualified yes: for a given computational budget, judiciously balancing the values of $m$ and $\ell$ tends to give better lookahead policy performance than simply increasing $\ell$ as much as possible, while setting $m = 0$ (which corresponds to no rollout). This is consistent with intuition obtained through geometric constructions such as Fig. 3.13, but it is difficult to establish with conclusive analysis.

### 3.4 How Sensitive is On-Line Play to the Off-Line Training Process?

An important issue to consider in approximation in value space is errors in the one-step or multistep minimization, or in the choice of terminal cost approximation $\hat{J}$. Such errors are often unavoidable because the control constraint set $U(x)$ is infinite, or because the minimization is simplified for reasons of computational expediency (see our subsequent discussion of multiagent problems). Moreover, to these errors, we may add the effect of errors due to rollout truncation, and errors due to changes in problem parameters, which are reflected in changes in Bellman’s equation (see our subsequent discussion of robust, adaptive, and model
**Figure 3.14** Schematic illustration of approximate PI. Either the policy evaluation and policy improvement phases (or both) are approximated with a value or a policy network, respectively. These could be neural networks, which are trained with (state, cost function value) data that is generated using the current base policy $\mu$, and with (state, rollout policy control) data that is generated using the rollout policy $\tilde{\mu}$.

Note that there are three different types of approximate implementation involving: 1) a value network but no policy network (here the value network defines a policy via one-step or multistep lookahead), or 2) a policy network but no value network (here the policy network has a corresponding value function that can be computed by rollout), or 3) both a policy and a value network (the approximation architecture of AlphaZero is a case in point).

Predictive control).

Under these circumstances the linearization of the Bellman equation at the point $\tilde{J}$ in Fig. 3.13 is perturbed, and the corresponding point $T^{m}_{\tilde{\mu}}\tilde{J}$ in Fig. 3.13 is also perturbed. However, the effect of these perturbations tends to be mitigated by the Newton step that produces the policy $\tilde{\mu}$ and the corresponding cost function $J_{\tilde{\mu}}$. The Newton step has a superlinear convergence property, so for an $O(\epsilon)$-order error (i.e., $O(\epsilon)/\epsilon$ stays bounded as $\epsilon \to 0$) in the calculation of $T^{n}_{\tilde{\mu}}\tilde{J}$, the error in $J_{\tilde{\mu}}$ will be of the much smaller order $o(\epsilon)$ (i.e., $o(\epsilon)/\epsilon \to 0$ as $\epsilon \to 0$), when $J_{\tilde{\mu}}$ is near $J^*$. This is a significant insight, as it suggests that extreme accuracy and fine-tuning of the choice of $\tilde{J}$ may not produce significant effects in the resulting performance of the one-step and particularly a multistep lookahead policy.

**Approximate Policy Iteration and Implementation Errors**

Both policy evaluation and policy improvement can be approximated, possibly by using training with data and approximation architectures, such as neural networks; see Fig. 3.14. Other approximations include simulation-based methods such as truncated rollout, and temporal difference methods for policy evaluation, which involve the use of basis functions. Moreover, multistep lookahead may be used in place of one-step lookahead, and simplified minimization, based for example on multiagent rollout, may also be used. Let us also mention the possibility of a combined rollout and PI algorithm, whereby we use PI for on-line policy improvement of the base policy, by using data collected during the rollout process. This idea is relatively
new and has not been tested extensively; see the subsequent discussion in Section 3.7 and the author’s paper [Ber21a].

Long-standing practical experience with approximate PI is consistent with the view of the effect of implementation errors outlined above, and suggests that substantial changes in the policy evaluation and policy improvement operations often have small but largely unpredictable effects on the performance of the policies generated. For example, when TD(\(\lambda\))-type methods are used for policy evaluation, the choice of \(\lambda\) has a large effect on the generated policy cost function approximations, but often has little and unpredictable effect on the performance of the generated policies. A plausible conjecture here is that the superlinear convergence property of the exact Newton step “smooths out” the effect of off-line approximation errors.

### 3.5 Why Not Just Train a Policy Network and Use it Without On-Line Play?

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? More generally, it is possible to use approximation in policy space, a major alternative approach to approximation in value space, whereby we 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. There are quite a few methods for performing this type of training, such as policy gradient and random search methods (see the books [SuB18] and [Ber19a] for an overview). Alternatively, some approximate DP or classical control system design method may be used.

An important advantage of approximation in policy space is that once the parametrized policy is obtained, the on-line computation of controls \(\mu(x, r)\) is often much faster compared with on-line lookahead minimization. For this reason, approximation in policy space can be used to provide an approximate implementation of a known policy (no matter how obtained) for the purpose of convenient use. 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 this book, 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). Figure 3.15 illustrates this fact with a one-dimensional linear-quadratic example, and compares the performance of a linear policy, defined by a scalar parameter, with its corresponding one-step lookahead policy.
Figure 3.15  Illustration of the performance enhancement obtained by rollout with an off-line trained base policy for the linear quadratic problem. Here the system equation is \( x_{k+1} = x_k + 2u_k \), and the cost function parameters are \( q = 1 \), \( r = 0.5 \). The optimal policy 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 consider policies of the form \( \mu(x) = Lx \), where \( L \) is the parameter, with cost function of the form \( J_\mu(x) = K_Lx^2 \). The figure shows the quadratic cost coefficient differences \( K_L - K^* \) and \( K_\tilde{\mu} - K^* \) as a function of \( L \), where \( K_L \) and \( K_\tilde{\mu} \) are the quadratic cost coefficients of \( \mu \) (without one-step lookahead/Newton step) and the corresponding one-step lookahead policy \( \tilde{\mu} \) (with one-step lookahead/Newton step).

3.6 Multiagent Problems and Multiagent Rollout

A major difficulty in the implementation of value space approximation with one-step lookahead is the minimization operation over \( U(x_k) \). When \( U(x_k) \) is infinite, or even when it is finite but has a very large number of elements, the minimization may become prohibitively time consuming. In the case of multistep lookahead the computational difficulty becomes even more acute. In this section we discuss how to deal with this difficulty when the control \( u \) consists of \( m \) components, \( u = (u_1, \ldots, u_m) \), with a separable control constraint for each component, \( u_\ell \in U_\ell(x), \ell = 1, \ldots, m \). Thus the control constraint set is the Cartesian product

\[
U(x) = U_1(x) \times \cdots \times U_m(x),
\]

where the sets \( U_\ell(x) \) are given. This structure is inspired by applications involving distributed decision making by multiple agents with communication and coordination between the agents; see Fig. 3.16.

To illustrate our approach, let us consider the discounted infinite horizon problem, and for the sake of the following discussion, assume that each set \( U_\ell(x) \) is finite. Then the one-step lookahead minimization of the standard rollout scheme with base policy \( \mu \) is given by

\[
\tilde{u} \in \arg\min_{u \in U(x)} E \left\{ g(x, u, w) + \alpha J_\mu(f(x, u, w)) \right\},
\]

where \( g(x, u, w) \) is the discount factor and \( f(x, u, w) \) is the state transition function.
and involves as many as \( n^m \) terms, 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 (3.15)]. Thus the standard rollout algorithm requires an exponential [order \( O(n^m) \)] number of 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 (3.16) is replaced by a sequence of \( m \) successive minimizations, one-agent-at-a-time, with the results incorporated into the subsequent minimizations. In particular, at state \( x \) we perform the sequence of minimizations

\[
\tilde{\mu}_1(x) \in \arg\min_{u_1 \in U_1(x)} E_w \left\{ g(x, u_1, \mu_2(x), \ldots, \mu_m(x), w) + \alpha J_\mu \left( f(x, u_1, \mu_2(x), \ldots, \mu_m(x), w) \right) \right\},
\]

\[
\tilde{\mu}_2(x) \in \arg\min_{u_2 \in U_2(x)} E_w \left\{ g(x, \tilde{\mu}_1(x), u_2, \mu_3(x), \ldots, \mu_m(x), w) + \alpha J_\mu \left( f(x, \tilde{\mu}_1(x), u_2, \mu_3(x), \ldots, \mu_m(x), w) \right) \right\},
\]

\[
\vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots
\]

\[
\tilde{\mu}_m(x) \in \arg\min_{u_m \in U_m(x)} E_w \left\{ g(x, \tilde{\mu}_1(x), \tilde{\mu}_2(x), \ldots, \tilde{\mu}_{m-1}(x), u_m, w) + \alpha J_\mu \left( f(x, \tilde{\mu}_1(x), \tilde{\mu}_2(x), \ldots, \tilde{\mu}_{m-1}(x), u_m, w) \right) \right\}.
\]

Thus each agent component \( u_\ell \) 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 \( u_{\ell+1}, \ldots, u_m \)
Figure 3.17 Equivalent formulation of the stochastic optimal control problem for the case where the control \( u \) consists of \( m \) components \( u_1, u_2, \ldots, u_m \):

\[
u = (u_1, \ldots, u_m) \in U_1(x_k) \times \cdots \times U_m(x_k).
\]

The figure depicts the \( k \)th stage transitions. Starting from state \( x \), we generate the intermediate states

\[
(x, u_1), (x_k, u_1, u_2), \ldots, (x, u_1, \ldots, u_{m-1}),
\]

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.

fixed at the values given by the base policy. This algorithm requires order \( O(nm) \) 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 (3.16) are proportional to the number of controls in the set \( U_k(x_k) \) 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 \( m \) sequential component decisions, thereby reducing the complexity of the control space while increasing the complexity of the state space. The potential advantage is that the extra state space complexity does not affect the computational requirements of some RL algorithms, including rollout.

To this end, we introduce a modified but equivalent problem, involving one-at-a-time agent control selection. At the generic state \( x \), we break down the control \( u \) into the sequence of the \( m \) controls \( u_1, u_2, \ldots, u_m \), and between \( x \) and the next state \( \bar{x} = f(x, u, w) \), we introduce artificial intermediate “states” \( (x, u_1), (x, u_1, u_2), \ldots, (x, u_1, \ldots, u_{m-1}) \), and corresponding transitions. The choice of the last control component \( u_m \) at “state” \( (x, u_1, \ldots, u_{m-1}) \) marks the transition to the next state \( \bar{x} = f(x, u, w) \) according to the system equation, while incurring cost \( g(x, u, w) \); see Fig. 3.17.
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 \((\mu_1(x), \ldots, \mu_m(x))\) of the original problem, is admissible for the reformulated problem, and has the same cost function for the original as well as the reformulated problem. Reversely, every policy for the reformulated problem can be converted into a policy for the original problem that produces the same state and control trajectories and has the same cost function.

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 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 minimization (3.16) is performed for just one state at each stage.

A major fact that follows from the preceding reformulation is that multiagent rollout still achieves cost improvement:

\[J_{\hat{\mu}}(x) \leq J_\mu(x), \quad \text{for all } x,
\]

where \(J_\mu(x)\) is the cost function of the base policy \(\mu\), and \(J_{\hat{\mu}}(x)\) is the cost function of the rollout policy \(\hat{\mu} = (\hat{\mu}_1, \ldots, \hat{\mu}_m)\), starting from state \(x\). Furthermore, this cost improvement property can be extended to multiagent PI schemes that involve one-agent-at-a-time policy improvement operations, and have sound convergence properties (see the book [Ber20a], Chapters 3 and 5, as well as the author’s papers [Ber19b], [Ber19c], [Ber20b], [Ber21b], and [BKB20]).

Another fact that follows from the preceding reformulation is that multiagent rollout may be viewed as a Newton step applied to the Bellman equation that corresponds to the reformulated problem. This is very important for our purposes. In particular, the superlinear cost improvement of the Newton step can still be obtained through multiagent rollout, even though the amount of computation for the lookahead minimization is dramatically reduced through one-agent-at-a-time minimization. This explains experimental results given in the paper by Bhattacharya et al. [BKB20], which have shown comparable performance for multiagent and standard rollout in the context of a large-scale multi-robot POMDP application.

We finally note that multiagent rollout can become 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], Section 5.3.4, and the paper [Ber21b]).

3.7 On-Line Simplified Policy Iteration

In this section, we describe some variants of the PI algorithm, introduced in the author’s recent paper [Ber21a], which are consistent with the approximation in value space theme of this work. The salient feature
of these variants is that they involve an exact Newton step and are suitable for on-line implementation, while still maintaining the principal character of PI, which we have viewed so far as an off-line algorithm.

Thus the algorithms of this section are on-line play-type of algorithms, which are capable of self-improvement through on-line experience. They are also simplified relative to standard PI in two ways:

(a) They perform policy improvement operations only for the states that are encountered during the on-line operation of the system.

(b) The policy improvement operation is simplified in that it uses approximate minimization in the Bellman equation of the current policy at the current state.

Despite these simplifications, we show that our algorithms generate a sequence of improved policies, which converge to a policy with a local optimality property. Moreover, with an enhancement of the policy improvement operation, which involves a form of exploration, they converge to a globally optimal policy.

The motivation comes from the rollout algorithm, which starts from some available base policy and implements on-line an improved rollout policy. In the algorithm of the present section, the data accumulated from the rollout implementation are used to improve on-line the base policy, and to asymptotically obtain a policy that is either locally or globally optimal.

We focus on a finite-state discounted Markovian decision problem, with states 1, . . . , n, and we use a transition probability notation. We denote states by the symbol x and successor states by the symbol y. The control/action is denoted by u, and is constrained to take values in a given finite constraint set U(x), which may depend on the current state x. The use of a control u at state x specifies the transition probability p_{xy}(u) to the next state y, at a cost g(x, u, y).

The cost of a policy μ starting from state x_0 is given by

\[ J_\mu(x_0) = \lim_{N \to \infty} E \left\{ \sum_{k=0}^{N-1} \alpha^k g(x_k, \mu(x_k), x_{k+1}) \right\} \bigg| \ x_0, \mu, \ x_0 = 1, \ldots, n, \]

where \( \alpha < 1 \) is the discount factor. As earlier, \( J_\mu \) is viewed as the vector in the n-dimensional Euclidean space \( \mathbb{R}^n \), with components \( J_\mu(1), \ldots, J_\mu(n) \).

In terms of our abstract notation, for each policy μ, the Bellman operator \( T_\mu \) maps a vector \( J \in \mathbb{R}^n \) to the vector \( T_\mu J \in \mathbb{R}^n \) that has components

\[ (T_\mu J)(x) = \sum_{y=1}^{n} p_{xy}(\mu(x)) \left( g(x, \mu(x), y) + \alpha J(y) \right), \quad x = 1, \ldots, n. \]  

(3.17)

The Bellman operator \( T : \mathbb{R}^n \mapsto \mathbb{R}^n \) is given by

\[ (TJ)(x) = \min_{u \in U(x)} \sum_{y=1}^{n} p_{xy}(u) \left( g(x, u, y) + \alpha J(y) \right), \quad x = 1, \ldots, n. \]  

(3.18)
It is well-known that for this problem, the operators $T_{\mu}$ and $T$ are sup-norm contractions (generally this is true for discounted problems with bounded cost per stage [Ber18a]; in our context, the number of states is finite, so the cost per stage is bounded). Thus $J_{\mu}$ is the unique solution of Bellman’s equation $J = T_{\mu}J$, or equivalently

$$J_{\mu}(x) = \sum_{y=1}^{n} p_{xy}(\mu(x)) \left( g(x, \mu(x), y) + \alpha J_{\mu}(y) \right), \quad x = 1, \ldots, n.$$  \tag{3.19}

Thus, $J^*$ is the unique solution of Bellman’s equation $J = TJ$, so that

$$J^*(x) = \min_{u \in U(x)} \sum_{y=1}^{n} p_{xy}(u) \left( g(x, u, y) + \alpha J^*(y) \right), \quad x = 1, \ldots, n.$$  \tag{3.20}

Moreover, the following optimality conditions hold

$$T_{\mu}J^* = TJ^* \quad \text{if and only if} \quad \mu \text{ is optimal},$$  \tag{3.21}

$$T_{\mu}J_{\mu} = TJ_{\mu} \quad \text{if and only if} \quad \mu \text{ is optimal}.$$  \tag{3.22}

The contraction property also implies that the VI algorithms

$$J^{k+1} = T_{\mu}J^k, \quad J^{k+1} = TJ^k$$

generate sequences $\{J^k\}$ that converge to $J_{\mu}$ and $J^*$, respectively, from any starting vector $J^0 \in \mathbb{R}^n$.

As discussed in Section 3, in the PI algorithm, the current policy $\mu$ is improved by finding $\tilde{\mu}$ that satisfies

$$T_{\tilde{\mu}}J_{\mu} = TJ_{\mu}$$

[i.e., by minimizing for all $x$ in the right-hand side of Eq. (3.18) with $J_{\mu}$ in place of $J$]. The improved policy $\tilde{\mu}$ is evaluated by solving the linear $n \times n$ system of equations $J_{\tilde{\mu}} = T_{\tilde{\mu}}J_{\tilde{\mu}}$, and then $(J_{\tilde{\mu}}, \tilde{\mu})$ becomes the new cost vector-policy pair, which is used to start a new iteration. Thus the PI algorithm starts with a policy $\mu^0$ and generates a sequence $\{\mu^k\}$ according to

$$J_{\mu^k} = T_{\mu^k}J_{\mu^k}, \quad T_{\mu^{k+1}}J_{\mu^k} = TJ_{\mu^k}.$$  \tag{3.23}

We now introduce an on-line variant of PI, which starts at time 0 with a state-policy pair $(x_0, \mu^0)$ and generates on-line a sequence of state-policy pairs $(x_k, \mu^k)$. We view $x_k$ as the current state of a system operating on line under the influence of the policies $\mu^1, \mu^2, \ldots$. In our algorithm, $\mu^{k+1}$ may differ from $\mu^k$ only at state $x_k$. The control $\mu^{k+1}(x_k)$ and the state $x_{k+1}$ are generated as follows:

We consider the right-hand sides of Bellman’s equation for $\mu^k$ (also known as $Q$-factors of $\mu^k$)

$$Q_{\mu^k}(x_k, u) = \sum_{y=1}^{n} p_{xky}(u) \left( g(x_k, u, y) + \alpha J_{\mu^k}(y) \right),$$  \tag{3.24}
and we select the control $\mu^{k+1}(x_k)$ from within the constraint set $U(x_k)$ with sufficient accuracy to satisfy the following sequential improvement condition

$$Q_{\mu^k}(x_k, \mu^{k+1}(x_k)) \leq J_{\mu^k}(x_k),$$  \hspace{1cm} (3.25)

with strict inequality whenever this is possible.† For $x \neq x_k$ the policy controls are not changed:

$$\mu^{k+1}(x) = \mu^k(x) \quad \text{for} \quad x \neq x_k.$$

The next state $x_{k+1}$ is generated randomly according to the transition probabilities $p_{x_kx_{k+1}}(\mu^{k+1}(x_k))$.

We first show that the current policy is monotonically improved, i.e., that

$$J_{\mu^{k+1}}(x) \leq J_{\mu^k}(x), \quad \text{for all} \quad x \text{ and } k,$$

with strict inequality for $x = x_k$ (and possibly other values of $x$) if

$$\min_{u \in U(x_k)} Q_{\mu^k}(x_k, u) < J_{\mu^k}(x_k).$$

To prove this, we note that the policy update is done under the condition (3.25). By using the monotonicity of $T_{\mu^{k+1}}$, we have for all $\ell \geq 1$,

$$T_{\mu^{k+1}}^{\ell+1} J_{\mu^k} \leq T_{\mu^{k+1}}^{\ell} J_{\mu^k} \leq J_{\mu^k},$$ \hspace{1cm} (3.27)

so by taking the limit as $\ell \to \infty$ and by using the convergence property of VI ($T_{\mu^{k+1}}^{\ell} J \to J_{\mu^{k+1}}$ for any $J$), we obtain $J_{\mu^{k+1}} \leq J_{\mu^k}$. Moreover, the algorithm selects $\mu^{k+1}(x_k)$ so that

$$(T_{\mu^{k+1}} J_{\mu^k})(x_k) = Q_{\mu^k}(x_k, u_k) < J_{\mu^k}(x_k) \quad \text{if} \quad \min_{u \in U(x_k)} Q_{\mu^k}(x_k, u) < J_{\mu^k}(x_k),$$

[cf. Eq. (3.26)], so that by using Eq. (3.27), we have $J_{\mu^{k+1}}(x_k) < J_{\mu^k}(x_k)$.

† By this we mean that if $\min_{u \in U(x_k)} Q_{\mu^k}(x_k, u) < J_{\mu^k}(x_k)$ we select a control $u_k$ that satisfies

$$Q_{\mu^k}(x_k, u_k) < J_{\mu^k}(x_k),$$ \hspace{1cm} (3.26)

and set $\mu^{k+1}(x_k) = u_k$, and otherwise we set $\mu^{k+1}(x_k) = \mu^k(x_k)$ [so Eq. (3.25) is satisfied]. Such a control selection may be obtained by a number of schemes, including brute force calculation and random search based on Bayesian optimization. The needed values of the Q-factor $Q_{\mu^k}$ and cost $J_{\mu^k}$ may be obtained in several ways, depending on the problem at hand, including by on-line simulation.
Local Optimality

We next discuss the convergence and optimality properties of the algorithm. We introduce a definition of local optimality of a policy, whereby the policy selects controls optimally only within a subset of states.

Given a subset of states $S$ and a policy $\mu$, we say that $\mu$ is locally optimal over $S$ if $\mu$ is optimal for the problem where the control is restricted to take the value $\mu(x)$ at the states $x \notin S$, and is allowed to take any value $u \in U(x)$ at the states $x \in S$.

Roughly speaking, $\mu$ is locally optimal over $S$, if $\mu$ is acting optimally within $S$, but under the (incorrect) assumption that once the state of the system gets to a state $x$ outside $S$, there will be no option to select control other than $\mu(x)$. Thus if the choices of $\mu$ outside of $S$ are poor, its choices within $S$ may also be poor.

Mathematically, $\mu$ is locally optimal over $S$ if

$$J_{\mu}(x) = \min_{u \in U(x)} \sum_{y=1}^{n} p_{xy}(u) (g(x, u, y) + \alpha J_{\mu}(y)),$$

for all $x \in S$,

$$J_{\mu}(x) = \sum_{y=1}^{n} p_{xy}(\mu(x)) \left( g(x, \mu(x), y) + \alpha J_{\mu}(y) \right),$$

for all $x \notin S$,

which can be written compactly as

$$(T_{\mu}J_{\mu})(x) = (T_{\mu})(x), \quad \text{for all } x \in S.$$ (3.28)

Note that this is different than (global) optimality of $\mu$, which holds if and only if the above condition holds for all $x = 1, \ldots, n$, rather than just for $x \in S$ [cf. Eq. (3.22)]. However, it can be seen that a (globally) optimal policy is also locally optimal within any subset of states.

Our main convergence result is the following.

**Proposition 3.1:** Let $\overline{S}$ be the subset of states that are repeated infinitely often within the sequence $\{x_k\}$. Then the corresponding sequence $\{\mu^k\}$ converges finitely to some policy $\overline{\mu}$ in the sense that $\mu^k = \overline{\mu}$ for all $k$ after some index $k$. Moreover $\overline{\mu}$ is locally optimal within $\overline{S}$, while $\overline{S}$ is invariant under $\overline{\mu}$, in the sense that

$$p_{xy}(\overline{\mu}(x)) = 0 \quad \text{for all } x \in \overline{S} \text{ and } y \notin \overline{S}.$$

**Proof:** The cost function sequence $\{J_{\mu_k}\}$ is monotonically nondecreasing, as shown earlier. Moreover, the number of policies $\mu$ is finite in view of the finiteness of the state and control spaces. Therefore, the number
of corresponding functions $J_\mu$ is also finite, so $J_{\mu^k}$ converges in a finite number of steps to some $\bar{J}$, which in view of the algorithm’s construction (selecting $u_k = \mu^k(x_k)$ if $\min_{u \in U(x_k)} Q_{\mu^k}(x_k, u) = J_{\mu^k}(x_k)$; cf. Eq. (3.26)), implies that $\mu^k$ will remain unchanged at some $\bar{\mu}$ with $J_{\bar{\mu}} = \bar{J}$ after some sufficiently large $k$.

We will show that the local optimality condition (3.28) holds for $S = \bar{S}$ and $\mu = \bar{\mu}$. In particular, we have $x_k \in \bar{S}$ and $\mu^k = \bar{\mu}$ for all $k$ greater than some index, while for every $x \in \bar{S}$, we have $x_k = x$ for infinitely many $k$. It follows that for all $x \in \bar{S}$,

$$Q_{\bar{\mu}}(x, \bar{\mu}(x)) = J_{\bar{\mu}}(x),$$  \hfill (3.29)

while by the construction of the algorithm,

$$Q_{\bar{\mu}}(x, u) \geq J_{\bar{\mu}}(x), \quad \text{for all } u \in U(x),$$  \hfill (3.30)

since the reverse would imply that $\mu^{k+1}(x) \neq \mu^k(x)$ for infinitely many $k$ [cf. Eq. (3.26)]. Condition (3.29) can be written as $J_{\bar{\mu}}(x) = (T_{\bar{\mu}} J_{\bar{\mu}})(x)$ for all $x \in \bar{S}$, and combined with Eq. (3.30), implies that

$$(T_{\bar{\mu}} J_{\bar{\mu}})(x) = (T J_{\bar{\mu}})(x), \quad \text{for all } x \in \bar{S}.$$  

This is the local optimality condition (3.28) with $S = \bar{S}$ and $\mu = \bar{\mu}$.

To show that $\bar{S}$ is invariant under $\bar{\mu}$, we argue by contradiction: if this were not so, there would exist a state $x \in \bar{S}$ and a state $y \notin \bar{S}$ such that $p_{xy}(\bar{\mu}(x)) > 0$, implying that $y$ would be generated following the occurrence of $x$ infinitely often within the sequence $\{x_k\}$, and hence would have to belong to $\bar{S}$ (by the definition of $\bar{S}$). Q.E.D.

Note an implication of the invariance property of the set $\bar{S}$ shown in the preceding proposition. We have that $\bar{\mu}$ is (globally) optimal under the assumption that for every policy there does not exist any strict subset of states that is invariant.

A Counterexample to Global Optimality

The following deterministic example shows that the policy $\bar{\mu}$ obtained by the algorithm need not be (globally) optimal. \† Here there are three states 1, 2, and 3. From state 1 we can go to state 2 at cost 1, and to state 3 at cost 0, from state 2 we can go to states 1 and 3 at cost 0, and from state 3 we can go to state 2 at cost 0 or stay in 3 at a high cost (say 10). The discount factor is $\alpha = 0.9$. Then it can be verified that the optimal policy is

$$\mu^*(1) : \text{Go to 3,} \quad \mu^*(2) : \text{Go to 3,} \quad \mu^*(3) : \text{Go to 2},$$

\† This example was given to us by Yuchao Li.
with optimal costs

\[ J^*(1) = J^*(2) = J^*(3) = 0, \]

while the policy

\[ \pi(1) : \text{Go to 2}, \quad \pi(2) : \text{Go to 1}, \quad \pi(3) : \text{Stay at 3}, \]

is strictly suboptimal, but is locally optimal over the set of states \( S = \{1, 2\} \). Moreover our on-line PI algorithm, starting from state 1 and the policy \( \mu^0 = \pi \), oscillates between the states 1 and 2, and leaves the policy \( \mu^0 \) unchanged. Note also that \( S \) is invariant under \( \pi \), consistently with Prop. 3.1.

**On-Line Variants of Policy Iteration with Global Optimality Properties**

To address the local versus global convergence issue illustrated by the preceding example, we consider an alternative scheme, whereby in addition to \( u_k \), we generate an additional control at a randomly chosen state \( x_k \neq x_k \).

† It is also possible to choose multiple additional states at time \( k \) for a policy improvement operation, and this is well-suited for the use of parallel computation.

In particular, assume that at each time \( k \), in addition to \( u_k \) and \( x_{k+1} \) that are generated according to Eq. (3.26), the algorithm generates randomly another state \( \tau_k \) (all states are selected with positive probability), performs a policy improvement operation at that state as well, and modifies accordingly \( \mu^{k+1}(\tau_k) \). Thus, in addition to a policy improvement operation at each state within the generated sequence \( \{x_k\} \), there is an additional policy improvement operation at each state within the randomly generated sequence \( \{\tau_k\} \).

Because of the random mechanism of selecting \( \tau_k \), it follows that at every state there will be a policy improvement operation infinitely often, which implies that the policy \( \pi \) ultimately obtained is (globally) optimal. Note also that we may view the random generation of the sequence \( \{\tau_k\} \) as a form of exploration. The probabilistic mechanism for generating the random sequence \( \{\tau_k\} \) may be guided by some heuristic reasoning, which aims to explore states with high cost improvement potential.

Finally let us mention the possibility of approximate implementations of the algorithms described above. In particular, one may start with some base policy, which may be periodically updated using some approximation in policy space scheme, while incorporating the policy improvement data generated so far. As long as the most recent policy improvement results are maintained for the states that have been encountered in the past, the convergence results described above will be maintained.

### 3.8 Exceptional Cases

Let us now consider situations where exceptional behavior occurs. One such situation is when the Bellman equation \( J = TJ \) has multiple solutions. Then the VI algorithm, when started at one of these solutions

\[ \pi(1) : \text{Go to 2}, \quad \pi(2) : \text{Go to 1}, \quad \pi(3) : \text{Stay at 3}, \]

is strictly suboptimal, but is locally optimal over the set of states \( S = \{1, 2\} \). Moreover our on-line PI algorithm, starting from state 1 and the policy \( \mu^0 = \pi \), oscillates between the states 1 and 2, and leaves the policy \( \mu^0 \) unchanged. Note also that \( S \) is invariant under \( \pi \), consistently with Prop. 3.1.

**On-Line Variants of Policy Iteration with Global Optimality Properties**

To address the local versus global convergence issue illustrated by the preceding example, we consider an alternative scheme, whereby in addition to \( u_k \), we generate an additional control at a randomly chosen state \( x_k \neq x_k \).

† It is also possible to choose multiple additional states at time \( k \) for a policy improvement operation, and this is well-suited for the use of parallel computation.

Finally let us mention the possibility of approximate implementations of the algorithms described above. In particular, one may start with some base policy, which may be periodically updated using some approximation in policy space scheme, while incorporating the policy improvement data generated so far. As long as the most recent policy improvement results are maintained for the states that have been encountered in the past, the convergence results described above will be maintained.

### 3.8 Exceptional Cases

Let us now consider situations where exceptional behavior occurs. One such situation is when the Bellman equation \( J = TJ \) has multiple solutions. Then the VI algorithm, when started at one of these solutions
will stay at that solution. More generally, it may be unclear whether the VI algorithm with converge to $J^*$, even when started at seemingly favorable initial conditions. Other types of exceptional behavior may also occur, including cases where the Bellman equation has no solution within the set of real-valued functions. The most unusual case of all is when $J^*$ is real-valued but does not satisfy the Bellman equation $J = TJ$, which in turn has other real-valued solutions; see [BeY16] and [Ber18a], Section 3.1. This is a highly unusual phenomenon, which will not be discussed here. It need not be of practical concern, as it arises only in artificial examples; see [BeY16]. Still it illustrates the surprising range of exceptional situations that should be taken into account in theoretical analyses and computational studies.

In this section we provide some examples that illustrate the mechanism by which exceptional behavior in infinite horizon DP can occur, and we highlight the need for rigorous analysis of RL methods when used in contexts that are beyond the well-behaved discounted case, where the Bellman operator is a contraction mapping. For further discussion and analysis that address exceptional behavior, including the frameworks of semicontractive and noncontractive DP, we refer to the author’s abstract DP monograph [Ber18a].

The Blackmailer’s Dilemma

This is a classical example involving a profit maximizing blackmailer. We formulate it as an SSP problem involving cost minimization, with a single state $x = 1$, in addition to the termination state $t$. We are in state 1 when the victim is compliant, and we are in state $t$ when the victim refuses to yield to the blackmailer’s demand (a refusal is permanent, in the sense that once the blackmailer’s demand is refused, all subsequent demands are assumed to be refused, so $t$ is a termination state). At state 1 we can choose a control $u \in (0, 1]$, which we regard as the demand made by the blackmailer. The problem is to find the blackmailer’s policy that maximizes his expected total gain.

To formulate this problem as a minimization problem, we will use $(-u)$ as the cost per stage. In
particular, upon choosing \( u \in (0, 1] \), we move to state \( t \) with probability \( u^2 \), and stay in state 1 with probability \( 1 - u^2 \); see Fig. 3.18. The idea is to optimally balance the blackmailer’s desire for increased demands (large \( u \)) with keeping his victim compliant (small \( u \)).

For notational simplicity, let us abbreviate \( J(1) \) and \( \mu(1) \) with just the scalars \( J \) and \( \mu \), respectively. Then in terms of abstract DP we have \( X = \{1\} \), \( U = (0, 1] \), and for every stationary policy \( \mu \), the corresponding Bellman operator \( T_\mu \), restricted to state 1 is given by

\[
T_\mu J = -\mu + (1 - \mu^2)J;
\]

(3.31)

(at the state \( t \), \( T_\mu \) is identically 0). Clearly \( T_\mu \) is linear, maps the real line \( \mathbb{R} \) to itself, and is a contraction with modulus \( 1 - \mu^2 \). Its unique fixed point within \( \mathbb{R} \), \( J_\mu \), is the solution of

\[
J_\mu = T_\mu J_\mu = -\mu + (1 - \mu^2)J_\mu,
\]

which yields

\[
J_\mu = \frac{-1}{\mu};
\]

see Fig. 3.19. Here all policies are stable and lead asymptotically to \( t \) with probability 1, and the infimum of \( J_\mu \) over \( \mu \in (0, 1] \) is \(-\infty\), implying also that \( J^* = -\infty \). However, there is no optimal policy.

The Bellman operator \( T \) is given by

\[
TJ = \min_{0 < u \leq 1} \{ -u + (1-u^2)J \},
\]

which after some calculation can be shown to have the form

\[
TJ = \begin{cases} 
-1 & \text{for } -\frac{1}{2} \leq J, \\
J + \frac{1}{4} & \text{for } J \leq -\frac{1}{2}.
\end{cases}
\]
The form of $T$ is illustrated in Fig. 3.19. It can be seen from this figure that the Bellman equation $J = TJ$ has no real-valued solution (the optimal cost $J^* = -\infty$ is a solution within the set of extended real numbers $[-\infty, \infty]$). Moreover, the VI algorithm will converge to $J^*$ starting from any $J \in \mathbb{R}$. It can be verified also that the PI algorithm, starting from any policy $\mu^0 \in (0, 1]$, produces the ever improving sequence of policies $\{\mu^k\}$ with $\mu^{k+1} = \mu^k / 2$. Thus $\mu^k$ converges to 0, which is not a feasible policy. Also $J_{\mu^k} = -1/\mu^k$, and we have $J_{\mu^k} \downarrow -\infty = J^*$, so the PI algorithm gives in the limit the infinite optimal cost. For additional related examples and discussion relating to the blackmailer problem see [Ber18a], Section 3.1.

A Shortest Path Problem

Another exceptional type of example is provided by shortest path problems that contain cycles of zero length; see the monograph [Ber18a], Section 3.1. In this case there are infinitely many solutions to Bellman’s equation, and the VI and PI algorithms, as well as the approximation in value space process exhibit unusual behavior. We demonstrate this with a shortest path problem involving a single state, denoted 1, in addition to the cost-free destination state $t$.

In particular, let $X = \{t, 1\}$, and assume that at state 1 there are two options: we can stay at 1 at cost 0, or move to $t$ at cost 1. Here $J^*(t) = J^*(1) = 0$, and there are just two policies, which correspond to the two options at state 1 and are stable. The optimal policy starting at state 1 is to stay at 1. If we restrict attention to cost functions $J$ with $J(t) = 0$, the Bellman operator is

$$(TJ)(1) = \min \{J(1), 1\},$$

and Bellman’s equation, written as an equation in $J(1)$, has the form

$$J(1) = \min \{J(1), 1\}.$$

The set of solutions of this equation is the interval $(-\infty, 1]$ and it is infinite; see Fig. 3.20. The optimal value $J^*(1) = 0$ lies in the interior of this set, and cannot be obtained by the VI algorithm, unless the algorithm is started at the optimal value.

Let us consider approximation in value space with cost approximation $\tilde{J}(1)$. Then it can be seen that if $\tilde{J}(1) < 1$, the one-step lookahead policy is to stay at state 1, which is optimal. If $\tilde{J}(1) > 1$, the one-step lookahead policy is to move from state 1 to state $t$, which is suboptimal. If $\tilde{J}(1) = 1$, either one of the two policies can be the one-step lookahead policy.

Consider also the PI algorithm, starting from the suboptimal policy $\mu$ that moves from state 1 to state $t$. Then $J_\mu(t) = 0$, $J_\mu(1) = 1$, and it can be seen that $\mu$ satisfies the policy improvement equation

$$\mu(1) \in \arg \min \{J_\mu(t), 1 + J_\mu(1)\},$$

54
Figure 3.20 Illustration of the Bellman equation for a shortest path problem in the exceptional case where there is a cycle of zero length. Restricted within the set of $J$ with $J(t) = 0$, the Bellman operator has the form

$$(TJ)(1) = \min \{J(1), 1\}.$$  

The set of solutions of Bellman’s equation, $J(1) = (TJ)(1)$ is the interval $(-\infty, 1]$ and contains $J^*(1) = 0$ in its interior.

(the same is true for the optimal policy that stays at state 1). Thus the PI algorithm may stop with the suboptimal policy $\mu$.

Problems where exceptional behavior occurs arise often in Markovian decision problems, once one departs from the most commonly discussed and best behaved paradigm of discounted cost with bounded cost per stage, where the mappings $T_\mu$ of all policies $\mu$ have favorable contraction properties. Moreover, problems arising in decision and control, such as those that have been addressed with MPC, often give rise to exceptional behavior. Further research and computational experimentation is expected to provide improved guidelines for the solution of such problems.

4. LINEAR QUADRATIC PROBLEM - ILLUSTRATIONS

In this section, we will use linear quadratic problems as a vehicle for graphical illustrations and insight into the suboptimal control ideas developed so far. This is possible because linear quadratic problems admit closed form solutions. Our discussion applies to multidimensional linear quadratic problems of the form (2.5)-(2.6) (cf. Example 2.1), but we will focus on the one-dimensional case to demonstrate graphically the approximation in value space ideas of this section. In particular, we will consider the system

$$x_{k+1} = ax_k + bu_k,$$

and the cost function

$$\sum_{k=0}^{\infty} (qx_k^2 + ru_k^2).$$
where $a, b, q, r$ are scalars with $b \neq 0$, $q > 0$, $r > 0$. It can be verified computationally (and also with some analysis) that the insights obtained from the one-dimensional case are generally correct for the multidimensional case of the linear quadratic problem, where the state cost weighting matrix $Q$ is positive definite. In Section 3.6 we obtained related insights about what happens in the exceptional case where we have $q = 0$.

4.1 Optimal Solution

The optimal solution was given for the multidimensional case of the linear quadratic problem in Example 2.1. For the one-dimensional case considered here, the optimal cost function has the form

$$J^*(x) = K^*x^2,$$

where the scalar $K^*$ solves a fixed point equation of the form

$$K = F(K),$$

with $F$ defined by

$$F(K) = \frac{a^2rK}{r + b^2K} + q. \quad (4.2)$$

This is the Riccati equation, which is equivalent to the Bellman equation $J = TJ$, restricted to the subspace of quadratic functions of the form $J(x) = Kx^2$. Essentially, by replacing the Bellman operator $T$ with the Riccati equation operator $F$ of Eq. (4.2), we can analyze the action of $T$ on this subspace.

The optimal cost function $J^*$ corresponds to $K^*$, which is the unique solution of the Riccati equation (4.1) within the nonnegative real line (this equation has another solution within the negative real line, which is of no interest); see Fig. 4.1. The optimal policy is a linear function of the state and has the form

$$\mu^*(x) = L^*x,$$

where $L^*$ is the scalar given by

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

4.2 Cost Functions of Stable Suboptimal Linear Policies

Suppose that we are given a linear policy of the form

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

where $L$ is a scalar. The corresponding closed loop system is

$$x_{k+1} = (a + bL)x_k = (a + bL)^k x_0,$$
Figure 4.1 Graphical construction of the solutions of the Riccati equation (4.1)-(4.2) for the linear quadratic problem. The optimal cost function is $J^*(x) = K^*x^2$, where the scalar $K^*$ solves the fixed point equation $K = F(K)$, where $F$ is the function given by

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

Because $F$ 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 $K = F(K)$ has one positive solution $K^*$ and one negative solution, denoted $\bar{K}$.

and the cost $J_\mu(x_0)$ is calculated as

$$\sum_{k=0}^{\infty} (q(a + bL)^2x_0^2 + rL^2(a + bL)^{2k}x_0^2) = \lim_{N \to \infty} \sum_{k=0}^{N-1} (q + rL^2)(a + bL)^{2k}x_0^2,
$$

Assuming $|a + bL| < 1$, i.e., that the closed loop system is stable, the above summation yields

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

for every initial state $x$, where

$$KL = \frac{q + rL^2}{1 - (a + bL)^2}. \quad (4.3)$$

With a straightforward calculation it can be verified that $KL$ is the unique solution of the linear equation

$$K = FL(K),$$

where

$$FL(K) = (a + bL)^2K + q + rL^2;$$

see Fig. 4.2. Again, by replacing the Bellman operator $T_\mu$ of the stable policy $\mu(x) = Lx$ with the Riccati equation operator $FL$, we can analyze the action of $T_\mu$ on the subspace of quadratic functions $J(x) = Kx^2$. 

57
Figure 4.2 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

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

where $K_L$ is 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 $L$ (i.e., the analog of $T_\mu$). If $\mu$ is unstable, we have $J_\mu(x) = \infty$ for $x \neq 0$.

Note that when $|a + bL| > 1$, so that $\mu$ is unstable, we have $J_\mu(x) = \infty$ for all $x \neq 0$, and the graph of $F_L$ intersects the 45-degree line at a negative $K$; the equation $K = F_L(K)$ has the negative solution $\frac{q + rL^2}{1 - (a + bL)^2}$, but this solution is unrelated to the cost function $J_\mu(\cdot)$, which has infinite value for all $x \neq 0$.

The one-dimensional problem of this section is well suited for geometric interpretations such as the ones we gave earlier in this section, because approximation in value space, and the VI, rollout, and PI algorithms, involve quadratic cost functions $J(x) = K x^2$, which can be represented by one-dimensional graphs as functions of just the number $K$. In particular, Bellman’s equation can be replaced by the Riccati equation (4.2). Similarly, the approximation in value space with one-step and multistep lookahead Figs. 3.6-3.7, the region of stability Figs. 3.8-3.9, and the rollout and PI Figs. 3.12-3.13 can be represented by one-dimensional graphs. We will next present these graphs and obtain corresponding geometrical insights.

Note that our discussion applies qualitatively to multidimensional linear quadratic problems of the form (2.5)-(2.6), and can be verified to a great extent by analysis, but an effective geometrical illustration is only
Figure 4.3 Graphical illustration of value iteration for the linear quadratic problem. It has the form $K_{k+1} = F(K_k)$ where

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

It is essentially equivalent to the VI algorithm with a quadratic starting function $J_0(x) = K_0x^2$. The algorithm converges to $K^*$ starting from anywhere in the interval $(\bar{K}, \infty)$, where $\bar{K}$ is the negative solution, as shown in the figure. Starting from values $K_0$ with $-r/b^2 < K_0 \leq \bar{K}$, the algorithm converges to the negative solution $\bar{K}$.

possible when the system is one-dimensional.

4.3 Bellman’s Equation and Value Iteration

The VI algorithm for the one-dimensional linear quadratic problem is illustrated in Fig. 4.3. It can be shown, and can also be seen from the figure, that the algorithm, which has the form

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

converges to $K^*$ starting from anywhere in the interval $(\bar{K}, \infty)$, where $\bar{K}$ is the negative solution. In particular, the algorithm converges to $K^*$ starting from any nonnegative value of $K$. It is interesting to note that, starting from values $K_0$ with

$$-\frac{r}{b^2} < K_0 \leq \bar{K},$$

the algorithm converges to the negative solution $\bar{K}$. When $K_0 \leq -r/b^2$, for the corresponding function $J(x) = K_0x^2$ we have $(TJ)(x) = -\infty$ for all $x$, and the algorithm is undefined. The literature on linear quadratic problems universally assumes that the iterative solution of the Riccati equation is started with nonnegative $K_0$, since to select a negative starting $K_0$ makes little sense: it slows down the convergence of the iteration without offering any apparent advantage.
4.4 Approximation in Value Space with One-Step and Multistep Lookahead - Stability Issues

Let us consider approximation in value space with a quadratic terminal cost approximation $\tilde{J}(x) = \tilde{K}x^2$; cf. Fig. 4.4. Then the one-step or multistep lookahead policy can be obtained analytically. In particular, for the one-step lookahead minimization, we have

$$\tilde{\mu}(x) \in \arg\min_u [qx^2 + ru^2 + \tilde{K}(ax + bu)^2],$$

which after a straightforward calculation, yields

$$\tilde{\mu}(x) = \tilde{L}x,$$

with the linear policy coefficient given by

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

Note, however, that this policy will not be stable if $|a + b\tilde{L}| \geq 1$, or equivalently if

$$\left|a - \frac{ab^2\tilde{K}}{r + ab^2\tilde{K}}\right| \geq 1.$$
Figure 4.5 Illustration of approximation in value space with two-step lookahead for the linear quadratic problem. Starting with a terminal cost approximation $J = \tilde{K}x^2$ ($\tilde{K} = 0$ in the figure), we obtain $K_1$ using a single value iteration. We then compute the corresponding linear policy $\tilde{\mu}(x) = \tilde{L}x$, where

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

and the corresponding cost function $K_1\tilde{L}x^2$, using the Newton step shown.

We may also construct the linearization of the function $F$ at $\tilde{K}$, and solve the corresponding linearized problem with a Newton step, as illustrated in Fig. 4.4. The case of $\ell$-step lookahead minimization can be similarly interpreted. 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 $\tilde{K}$. Figure 4.5 depicts the case $\ell = 2$.

Figure 4.5 also suggests that if $K$ is a scalar that lies strictly within the region of stability, i.e.,

$$
\left| a - \frac{ab^2K}{r + ab^2K} \right| < 1,
$$

then there exists a constant $c >$ such that for all $\tilde{K} \geq K$ we have

$$
|K_L - K^*| \leq c |\tilde{K} - K^*|^2, \quad (4.4)
$$

where $\tilde{L}$ corresponds to the policy obtained by approximation in value space with one-step lookahead and terminal cost approximation $\tilde{J}(x) = \tilde{K}x^2$. This fact follows from the quadratic convergence rate property of Newton’s method.

In the case of $\ell$-step lookahead this result takes a stronger form, whereby the quadratic convergence rate estimate (4.4) has the form

$$
|K_L - K^*| \leq c |F^{\ell-1}(\tilde{K}) - K^*|^2,
$$

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 $|K_L - K^*|$ is obtained, but when $\tilde{K}$ is close to $K^*$, the performance gain is relatively insignificant.
4.5 Rollout and Policy Iteration

The rollout algorithm with a stable base policy $\mu$ is illustrated in Fig. 4.6. The PI algorithm is simply the repeated application of rollout. Let us derive the algorithm starting from a linear base policy of the form

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

where $L_0$ is a scalar. We require that $L_0$ is such that the closed loop system

$$x_{k+1} = (a + bL_0)x_k,$$  \hspace{1cm} (4.5)

is stable, i.e., $|a + bL_0| < 1$. This is necessary for the policy $\mu^0$ to keep the state bounded and the corresponding costs $J_{\mu^0}(x)$ finite. We will see that the PI algorithm generates a sequence of linear stable policies.

To describe the policy evaluation and policy improvement phases for the starting policy $\mu^0$, we first calculate $J_{\mu^0}$ by noting that it involves the uncontrolled closed loop system (4.5) and a quadratic cost function. Similar to our earlier calculations, it has the form

$$J_{\mu^0}(x) = K_0 x^2,$$  \hspace{1cm} (4.6)
where

\[ K_0 = \frac{q + rL_0^2}{1 - (a + bL_0)^2}. \]  

(4.7)

Thus, the policy evaluation phase of PI for the starting linear policy \( \mu^0(x) = L_0x \) yields \( J_{\mu^0} \) in the form (4.6)-(4.7). The policy improvement phase involves the quadratic minimization

\[ \mu^1(x) \in \operatorname{arg\,min}_u \left[ qx^2 + ru^2 + K_0(ax + bu)^2 \right], \]

and after a straightforward calculation yields \( \mu^1 \) as the linear policy \( \mu^1(x) = L_1x \), where

\[ L_1 = -\frac{abK_0}{r + ab^2K_0}. \]

It can also be verified that \( \mu^1 \) is a stable policy. An intuitive way to get a sense of this is via the cost improvement property of PI: we have \( J_{\mu^1}(x) \leq J_{\mu^0}(x) \) for all \( x \), so \( J_{\mu^1}(x) \) must be finite, which implies stability of \( \mu^1 \).

The preceding calculation can be continued, so the PI algorithm yields the sequence of linear policies

\[ \mu^k(x) = L_kx, \quad k = 0, 1, \ldots, \]

where \( L_k \) is generated by the iteration

\[ L_{k+1} = -\frac{abK_k}{r + ab^2K_k}, \]

with \( K_k \) given by

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

[cf. Eq. (4.7)].

The corresponding cost function sequence has the form \( J_{\mu^k}(x) = K_kx^2 \). It can be shown to converge to the optimal cost function \( J^* \), while the generated sequence of linear policies \( \{\mu^k\} \), where \( \mu^k(x) = L_kx \), converges to the optimal policy. The convergence rate of the sequence \( \{K_k\} \) can be shown to be quadratic, i.e., there exists a constant \( c \) such that

\[ |K_{k+1} - K^*| \leq c|K_k - K^*|^2, \]

for all \( k \), assuming that the initial policy is linear and stable. This is consistent with the interpretation of PI as Newton’s method (see Kleinman [Kle68], who considered the continuous-time version of the linear quadratic problem, and other references noted earlier).
Figure 4.7 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 the number of rollout steps is $m = 4$.

4.6 Truncated Rollout - Tradeoffs with Lookahead Minimization

Truncated rollout with a stable linear base policy $\mu(x) = Lx$ and terminal cost approximation $\tilde{J}(x) = \tilde{K}x^2$ is illustrated in Fig. 4.7. The rollout policy $\tilde{\mu}$ is obtained from the equation

$$T_{\mu}T^{\ell-1}T^m_{\mu} \tilde{J} = T^\ell T^m_{\mu} \tilde{J},$$

where $\ell \geq 1$ is the length of the lookahead minimization and $m \geq 0$ is the length of the rollout lookahead, with $m = 0$ corresponding to no lookahead by rollout (in Fig. 4.7, we have $\ell = 1$ and $m = 4$).

We mentioned some interesting performance issues in our discussion of truncated rollout in Section 3.3, and we will now revisit these issues within the context of our linear quadratic problem. In particular we noted that:

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

(b) 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
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. 4.7.

In particular, let us consider our one-dimensional linear quadratic problem, the corresponding value $K^*$ (the optimal cost coefficient), and the value $K_s$, which demarcates the region of stability, i.e., one-step lookahead yields a stable policy if and only if $\hat{K} > K_s$. Consider also two parameters of the truncated rollout method: $K_{\mu}$ (the cost coefficient of the base policy), and $\tilde{K}$ (the terminal cost approximation coefficient).

We have $K_s \leq K^* \leq K_{\mu}$, so the three parameters $K_s, K^*, K_{\mu}$ divide the real line in the four intervals I through IV, depicted in Fig. 4.8. Then by examining Fig. 4.7, we see that the behavior of truncated rollout depends on the interval in which the terminal cost coefficient $\tilde{K}$ lies. In particular:

(a) **For $\hat{K}$ in interval I**: Long total $(\ell + m)$-step lookahead is needed to get the starting point of the Newton step within the region of stability. Best and computationally economical results are obtained by taking $\ell = 1$ and $m$ large enough to bring the starting point of the Newton step within the region of stability, and hopefully close to $K^*$.

(b) **For $\hat{K}$ in interval II**: $\ell = 1$ and $m \geq 0$ are sufficient for stability. Best results are obtained when $\ell = 1$ and $m$ is the (generally unknown) value that brings the starting point of the Newton step close to $K^*$.

(c) **For $\hat{K}$ in interval III**: $\ell = 1$ and $m \geq 0$ are sufficient for stability. Best results are obtained when $\ell = 1$ and $m = 0$ (since the rollout lookahead is counterproductive and takes the starting point of the Newton step away to $K^*$ and towards $K_{\mu}$). Still, however, even with $m > 0$, we have the cost improvement property $K_{\tilde{\mu}} \leq K_{\mu}$.
\[ F(K) = \frac{a^2 r K}{r + b^2 K} \]

Figure 4.9  Illustration of the Bellman equation and the VI algorithm

\[ K_{k+1} = F(K_k) \]

for the linear quadratic problem in the exceptional case where \( q = 0 \).

(d) For \( \tilde{K} \) in interval IV: \( \ell = 1 \) and \( m \geq 0 \) are sufficient for stability. Best results are obtained for values of \( m \) and \( \ell \), which depend on how far \( \tilde{K} \) is from \( K_{\mu} \). Here, values that likely work well are the ones for which \( m \) is fairly large, and \( \ell \) is close to 1 (large enough \( m \) will bring the starting point of the Newton step close to \( K_{\mu} \); \( \ell = 1 \) corresponds to the Newton step, \( \ell > 1 \) improves the starting point of the Newton step by value iteration, but is likely not worth the extra computational cost).

Of course, a practical difficulty here is that we don’t know the interval in which \( \tilde{K} \) lies. However, it is clear that by using rollout with \( m \geq 1 \) works well in most cases as an economical substitute for long lookahead minimization. In particular, when \( \tilde{K} \) lies in intervals I, II, and IV, using \( m > 0 \) provides a stronger stability guarantee and improved performance through a better starting point for the Newton step. Even in the case where \( \tilde{K} \) lies in interval III, using \( m > 0 \) is not very damaging: we still obtain performance that is no worse than the base policy, i.e., \( K_{\tilde{\mu}} \leq K_{\mu} \). An interesting research question is to investigate analytically as well as computationally, the multidimensional analogs of the intervals I-IV (which will now become subsets of the set of symmetric matrices). While it seems that the preceding discussion should generalize broadly, an investigation of the multidimensional case promises to be both challenging and illuminating.

4.7 Exceptional Behavior in Linear Quadratic Problems

It turns out that exceptional behavior can occur even for one-dimensional linear quadratic problems, when
the positive definiteness assumption on the matrix $Q$ is violated. In particular, let us consider the system

$$x_{k+1} = ax_k + bu_k,$$

and the cost function

$$\sum_{k=0}^{\infty} ru_k^2,$$

where $a, b, r$ are scalars with $b \neq 0$, $r > 0$. Here there is no penalty for the state being nonzero (i.e., $Q = 0$), while the system is unstable when left uncontrolled (i.e., $a > 1$).

In this case, since the cost does not depend on the state, it is optimal to apply control $u = 0$ at any state $x$, i.e., $\mu^*(x) \equiv 0$, and the optimal cost function is $J^*(x) \equiv 0$. The Riccati equation is given by

$$K = F(K),$$

where $F$ defined by

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

As shown in Fig. 4.9, it has two nonnegative solutions:

$$K^* = 0 \quad \text{and} \quad \hat{K} = \frac{r(a^2 - 1)}{b^2}.$$

The solution $K^*$ corresponds to the optimal cost function. It turns out that the solution $\hat{K}$ is also interesting: it can be shown to be the optimal cost function within the subclass of linear policies that are stable. A proof of this is given in the author’s monograph [Ber18a], Section 3.1.

Consider also the VI algorithm

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

starting from some $K_0 > 0$ [cf. Eq. (2.16)]. As shown from Fig. 4.9, it generates a positive scalar sequence that converges to $\hat{K}$. If the VI algorithm is started at the optimal $K^* = 0$, it stays at $K^*$. It can also be verified that the PI algorithm generates a sequence of linear stable policies starting from a linear stable policy. The sequence converges to the optimal stable policy that corresponds to $\hat{K}$. In summary, the PI algorithm starting from a linear stable policy converges to $\hat{J}$, the optimal cost function over linear stable policies, but not to the optimal policy.

Regarding rollout, it is essential to use a stable base policy, for otherwise the cost function of the base policy is infinite for some states $x$, and the rollout policy cannot be defined. Finally, the region of stability is of the form $(\tilde{K}, \infty)$, where $\tilde{K}$ is some critical level, similar to the standard case where $q > 0$. In fact $\tilde{K}$ is such that the derivative $dF(K)/dK$ is equal to 1. By solving the equation

$$\frac{dF(K)}{dK} = \frac{a^2 r^2}{(r + b^2 K)^2} = 1,$$
we obtain
\[ \tilde{K} = \frac{(a - 1)r}{b^2}. \]

Thus, when the system is unstable if left uncontrolled \((a > 1)\), to obtain a stable system by one-step lookahead, we must use a quadratic cost function approximation that exceeds \(\tilde{K}x^2\).

5. ROBUSTNESS, ADAPTIVE CONTROL, AND MODEL PREDICTIVE CONTROL

Our discussion so far dealt with problems with a known and unchanging 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. 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 involves parameters that are either not known exactly or may change over time. In such cases it is important to design controllers that take the parameter changes into account. The methodology for doing so is generally known as adaptive control, an intricate and multifaceted subject, with many and diverse applications, and a long history.†

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 book by Sutton and Barto [SuB18], “learning from interaction with the environment is a foundational idea underlying nearly all theories of learning and intelligence.” The idea of interaction with the environment is typically 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. This is one of the most challenging areas of stochastic optimal and suboptimal control, and has been studied extensively since the early 1960s.

† The difficulties of designing adaptive controllers are often underestimated. Among others, they complicate the balance between off-line training and on-line play, which we discussed in Section 1 in connection to 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!
In what follows in this section, we will briefly review some of the principal types of adaptive control methods. We will then focus on schemes that are based on on-line replanning, including the use of rollout.

**Robust and PID 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 design a controller that is adequate for the entire range of the changing parameters. This is sometimes called a *robust controller*. A robust controller makes no effort to keep track of changing problem parameters. It is just designed so that it is resilient to parameter changes.

An important time-honored robust control approach for continuous-state problems is the *PID (Proportional-Integral-Derivative) controller*; see e.g., the books by Aström and Hagglund [AsH95], [AsH06]. 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 by System Identification and On-Line Replanning**

In robust control schemes, such as PID control, no attempt is made to maintain a mathematical model and to track unknown model parameters as they change. Alternatively we may introduce into the controller a mechanism for measuring or estimating the unknown or changing system parameters, and make suitable control adaptations in response.†

Let us note here that updating problem parameters need not require an elaborate algorithm. In many cases the set of problem parameters may take a known finite set of values (for example each set of parameter

† In the adaptive control literature, schemes that involve parameter estimation are sometimes called *indirect*, while schemes that do not involve parameter estimation (like PID control) are called *direct*. To quote from the book by Aström and Wittenmark [AsW08], “indirect methods are those in which the estimated parameters are used to calculate required controller parameters” (see Fig. 5.1). The methods subsequently described in this section, and the rollout-based adaptive control methods discussed in the next section should be viewed as indirect. For accounts of adaptive control, we refer to the books by Bodson [Bod20], Goodwin and Sin [GoS84], Ioannou and Sun [IoS96], Jiang and Jiang [JiJ17], Krstic, Kanellakopoulos, and Kokotovic [KKK95], Kokotovic [Kok91], Kumar and Varaiya [KuV86], Liu, et al. [LWW17], Lavretsky and Wise [LaW13], Narendra and Annaswamy [NaA12], Sastry and Bodson [SaB11], Slotine and Li [SIL91], and Vrabie, Vamvoudakis, and Lewis [VVL13].
values may correspond to a distinct maneuver of a vehicle, motion of a robotic arm, flying regime of an aircraft, etc). Once the control scheme detects a change in problem parameters, it can incorporate the change into the approximation in value space scheme, and in the case of policy rollout, it may switch to a corresponding predesigned policy.

In what follows in this section (including our discussion of MPC in Section 5.2), we will assume that there is a mechanism to learn (perhaps imperfectly and by some unspecified procedure) the model of the system as it evolves over time. We will loosely refer to this learning process with the classical name system identification, but we will not go into specific identification methods, keeping in mind that such methods could be imprecise and challenging, but could also be fast and simple, depending on the problem at hand.

An apparently reasonable scheme 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 the system’s operation in order to take into account subsequent changes of the parameters. Note that 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 generated in the background, and introduced into the control process, whenever this is thought to be desirable; see Fig. 5.1.

One drawback of this approach is that it is not always easy to determine when to terminate one phase and start the other. A second difficulty, of a more fundamental nature, is that the control process may
make some of the unknown parameters invisible to the estimation process. This is known as the problem of parameter identifiability, which is discussed in the context of adaptive control in several sources. On-line parameter estimation algorithms, which address among others the issue of identifiability, have been discussed extensively in the control theory literature, but the corresponding methodology is complex and beyond our scope in this book. However, assuming that we can make the estimation phase work somehow, we are free to reoptimize the controller using the newly estimated parameters, in a form of 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 and/or data-intensive 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 in the next section.

5.1 Approximation in Value Space, Rollout, and Adaptive Control

We will now consider an approach for dealing with unknown or changing parameters, which is based on rollout and on-line replanning. We have already noted this approach in Section 1, where we stressed the importance of fast on-line policy improvement.

Let us assume that some problem parameters change over time and the controller becomes aware of the changes, perhaps after a suitable delay for data collection and estimation. The method by which the problem parameters are recalculated or become known is immaterial for the purposes of the following discussion. It may involve a limited form of parameter estimation, whereby the unknown parameters are “tracked” by data collection over a few time stages, with due attention paid to issues of parameter identifiability; or it may involve new features of the control environment, such as a changing number of servers and/or tasks in a service system.

We thus assume away/ignore the detailed 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 some base policy is particularly attractive. The base policy may be either a fixed robust controller (such as some form of PID control) or it may be updated over time (in the background, on the basis of some unspecified rationale), in which case the rollout policy will be revised both in response to the changed base policy and in response to the changing parameters.

Here the advantage of rollout is that it is simple, reliable, and relatively fast. In particular, it does not require a complicated training procedure, 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 available controls at the current state are compared through a one-step or multistep minimization, with cost function approximation provided by the base policy (cf. Fig. 5.2).
One-step lookahead minimization 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 estimates of their values are incorporated into the lookahead minimization and rollout operations. Truncated rollout with multistep lookahead minimization and terminal cost approximation is also possible. The base policy may also be revised based on various criteria. For the discussion of this section, we may assume that all the changing parameter information is provided by some computation and sensor “cloud” that is beyond our control.

Another issue to consider is the stability and robustness properties of the rollout policy. In this connection, it can be generally proved, under mild conditions, that if the base policy is stable within a range of parameter values, the same is true for the rollout policy; this can also be inferred from Fig. 3.12. Related ideas have a long history in the control theory literature; see Beard [Bea95], Beard, Saridis, and Wen [BSW99], Jiang and Jiang [JiJ17], Kalise, Kundu, Kunisch [KKK20].

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. In this connection, we note that accelerated/truncated or simplified versions of rollout, as well as parallel computation, can be used to meet this time constraint.

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.

The following example illustrates on-line replanning with the use of rollout in the context of the simple one-dimensional linear quadratic problem that we discussed earlier. The purpose of the example is to show analytically how rollout with a base 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; for example linear quadratic problems with constraints.

**Example 5.1 (On-Line Replanning for Linear Quadratic Problems Based on Rollout)**

Consider a deterministic undiscounted infinite horizon linear quadratic problem involving the linear system

\[ x_{k+1} = x_k + bu_k, \]

and the quadratic cost function

\[ \lim_{N \to \infty} \sum_{k=0}^{N-1} (x_k^2 + ru_k^2). \]

This is the one-dimensional problem of the preceding section for the special case where \( a = 1 \) and \( q = 1 \). The optimal cost function is given by

\[ J^*(x) = K^* x^2, \]

where \( K^* \) is the unique positive solution of the Riccati equation

\[ K = \frac{rK}{r + b^2K} + 1. \tag{5.1} \]

The optimal policy has the form

\[ \mu^*(x) = L^* x, \tag{5.2} \]

where

\[ L^* = \frac{-bK^*}{r + b^2K^*}. \tag{5.3} \]

As an example, consider the optimal policy that corresponds to the nominal problem parameters \( b = 2 \) and \( r = 0.5 \): this is the policy (5.2)-(5.3), with \( K \) computed as the positive solution of the quadratic Riccati Eq. (5.1) for \( b = 2 \) and \( r = 0.5 \) [cf. Eq. (4.2)]. For these nominal parameter values, we have

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

From Eq. (5.3) we then also obtain

\[ L = \frac{2 + \sqrt{6}}{5 + 2\sqrt{6}}. \tag{5.4} \]

We will now consider changes of the values of \( b \) and \( r \) while keeping \( L \) constant to the preceding value, 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. (5.1).

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

\[
\mu_L(x) = Lx,
\]

where \( L \) is given by Eq. (5.4). Here, we have [cf. Eq. (4.3)]

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

(5.5)

(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 \( \mu_L \) as base policy. Using the formulas derived earlier, we have [cf. Eq. (5.5)]

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

and [cf. Eq. (4.3)]

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

Figure 5.3 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 \( \mu_L \), i.e., the performance loss incurred by ignoring the changes in the values of \( b \) and \( r \), and continuing to use the policy \( \mu_L \), 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 \( \mu_L \) unchanged.

Note that Fig. 5.3 illustrates the behavior of the error ratio \( \frac{J - J^*}{J} \), where for a given initial state, \( J \) 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 superlinear/quadratic convergence rate of Newton’s method that underlies the rollout algorithm.

5.2 Approximation in Value Space, Rollout, and Model Predictive Control

In this section, we briefly discuss the MPC methodology, with a view towards its connection with approximation in value space and the rollout algorithm. We will focus on the undiscounted infinite horizon deterministic problem, which involves the system

\[
x_{k+1} = f(x_k, u_k),
\]
Figure 5.3 Illustration of 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. (5.4)]. The rollout policy performance is very close to optimal, even when the base policy is far from optimal.

Note that, as the figure illustrates, we have
\[
\lim_{J \to J^*} \frac{\tilde{J} - J^*}{J - J^*} = 0,
\]
where for a given initial state, $\tilde{J}$ is the rollout performance, $J^*$ is the optimal performance, and $J$ is the base policy performance. This is a consequence of the superlinear/quadratic convergence rate of Newton’s method that underlies rollout, and guarantees that the rollout performance approaches the optimal much faster than the base policy performance does.
whose state $x_k$ and control $u_k$ are finite-dimensional vectors. The cost per stage is assumed nonnegative

$$g(x_k, u_k) \geq 0, \quad \text{for all } (x_k, u_k),$$

(e.g., a positive definite quadratic cost). There are control constraints $u_k \in U(x_k)$, and to simplify the following discussion, we will initially consider no state constraints. We assume that the system can be kept at the origin at zero cost, i.e.,

$$f(0, \overline{u}_k) = 0, \quad g(0, \overline{u}_k) = 0 \quad \text{for some control } \overline{u}_k \in U(0).$$

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

The scope of the MPC algorithm has grown over time to include problem and algorithm variations and extensions, and the literature on MPC is voluminous. For detailed accounts, we refer to the textbooks by 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]. We will first focus on the original form of the MPC algorithm proposed in the form given here by Keerthi and Gilbert [KeG88].

In this algorithm, at each encountered state $x_k$, we apply a control $\tilde{u}_k$ that is computed as follows; see Fig. 5.4:

(a) We solve an $\ell$-stage optimal control problem involving the same cost function and the requirement that the state after $\ell$ steps is driven to 0, i.e., $x_{k+\ell} = 0$. This is the problem

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

subject to the system equation constraints

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

the control constraints

$$u_t \in U(x_t), \quad t = k, \ldots, k + \ell - 1,$$
Figure 5.4 Illustration of the problem solved by a classical form of MPC at state $x_k$. We minimize the cost function over the next $\ell$ 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.

and the terminal state constraint

$$x_{k+\ell} = 0. \quad (5.9)$$

Here $\ell$ 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. (5.6)] 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 $\ell$ 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 yields a stable policy, as we have discussed in Section 3.2.

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.

Note also that when faced with changing problem parameters, it is natural to consider on-line replanning as per our earlier discussion. In particular, once new estimates of system and/or cost function parameters become available, MPC can adapt accordingly by introducing the new parameter estimates into the \( \ell \)-stage optimization problem in (a) above.

**Variants of MPC - Terminal Cost Approximation**

In a common variant of MPC, the requirement of driving the system state to 0 in \( \ell \) steps in the \( \ell \)-stage MPC problem (5.6), is replaced by a terminal cost \( G(x_{k+\ell}) \). Thus at state \( x_k \), we solve the problem

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

(5.10)

instead of problem (5.6) where we require that \( x_{k+\ell} = 0 \). This variant can also 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 as approximation in value space with \( \ell \)-step lookahead and terminal cost approximation given by \( G \). Thus our discussion of Section 3 relating to the region of stability of such schemes applies, and relates to results that are known within the MPC framework under various conditions (see the paper by Mayne at al. [MRR00], the MPC book [RMD17], and the author’s book [Ber20a], Section 3.1.2).

Note that the preceding MPC controller may outperform substantially its base policy (in relative terms), particularly if the base policy is close to optimal, in the sense that

\[
G(x_{k+\ell}) \approx J^*(x_{k+\ell}) + \text{a constant}.
\]

This is due to the superlinear/quadratic convergence rate of Newton’s method that underlies approximation in value space, as we have discussed in Section 3.

An important question is to choose the terminal cost approximation so that the resulting MPC controller is stable. A common approach for linear systems and quadratic cost is to introduce truncated rollout with a stable base policy; see Magni et al. [MDM01]. Our discussion of stability in Section 4.4 is relevant within this context.

Another variant of MPC involves the use of truncated rollout, whereby the values \( G(x_{k+\ell}) \) in Eq. (5.10) are computed by running some base policy for a number of steps \( m \) followed by some terminal cost function approximation. This is quite similar to standard truncated rollout, except that the computational solution of the lookahead minimization problem (5.10) may become complicated when the control space is infinite.
Moreover, the base policy may be used to address state constraints; see the papers by Rosolia and Borelli [RoB17], [RoB19], and the discussion in the author’s RL book [Ber20a].

A Rollout Variant of MPC with Multiple Terminal States and Base Policies

In another variation of MPC, proposed in the paper by Li et al. [LJM21], instead of driving the state to 0 at the end of \( \ell \) steps, we consider multiple terminal system states at the end of the \( \ell \)-step horizon, as well as the use of multiple base policies for rollout. In particular, in this scheme we have a finite set of states \( \mathcal{X} \) and a finite set of stable base policies \( \mathcal{M} \), and we assume that we have computed off-line the cost function values \( J_\mu(x) \) for all \( x \in \mathcal{X} \) and \( \mu \in \mathcal{M} \). At state \( x_k \), to compute the MPC control \( \hat{u}_k \), we solve for each \( x \in \mathcal{X} \) a problem that is the same as the problem (5.6)-(5.9) that is solved by the classical form of MPC, except that the terminal state \( x_{k+\ell} \) is equal to \( x \) instead of \( x_{k+\ell} = 0 \). This is the problem

\[
\min_{u_t, t=k,...,k+\ell-1} \sum_{t=k}^{k+\ell-1} g(x_t, u_t),
\]

subject to the system equation constraints

\[
x_{t+1} = f(x_t, u_t), \quad t = k, \ldots, k + \ell - 1,
\]

the control constraints

\[
u_t \in U(x_t), \quad t = k, \ldots, k + \ell - 1,
\]

and the terminal state constraint

\[
x_{k+\ell} = x.
\]

Let \( V(x_k;x) \) be the optimal value of this problem. Having computed \( V(x_k;x) \) for all \( x \in \mathcal{X} \), we compare all values

\[
V(x_k;x) + J_\mu(x), \quad x \in \mathcal{X}, \mu \in \mathcal{M},
\]

and find the pair \((\mathbf{x}, \mu)\) that yields the minimal value of \( V(x_k;x) + J_\mu(x) \). We then define the MPC control \( \hat{u}_k \) to be the control \( u_k \) that attains the minimum in the corresponding problem (5.11)-(5.14) with \( x = \mathbf{x} \).

Thus, in this variant of MPC we solve multiple problems of the type that is solved in the classical form of MPC, for multiple values of the terminal state \( x_{k+\ell} \), and we then compute the MPC control based on the “best” terminal state \( x \in \mathcal{X} \), assuming that the “best” base policy \( \pi \) will be used after state \( k + \ell \). It is possible to show, under appropriate conditions,\(^\dagger\) that the cost function \( J_{\hat{\mu}} \) of the MPC policy \( \hat{\mu} \), which applies \( \hat{\mu}(x_k) = \hat{u}_k \) as described above, has the cost improvement property

\[
J_{\hat{\mu}}(x) \leq J_\mu(x), \quad \text{for all } x \in \mathcal{X}, \mu \in \mathcal{M};
\]

\(^\dagger\) These conditions include that for every \( x \in \mathcal{X} \), we have \( f(x, \mu(x)) \in X \) for some \( \mu \in \mathcal{X} \), which plays the same role as the assumption that the origin is cost free and absorbing in the classical form of MPC.
see [LJM21]. Moreover, based on this property and the fact that the base policies \( \mu \in \mathcal{M} \) are stable, it follows that the MPC policy \( \tilde{\mu} \) thus obtained is also stable.

The preceding variation can also be used for systems with arbitrary state and control spaces, continuous as well as discrete. It is also well-suited for addressing state constraints, provided the base policies are designed to satisfy these constraints; see [LJM21]. In this case, the state constraints are included in the constraints of the \( \ell \)-step problems (5.11)-(5.14). We refer to the paper [LJM21], which provides supporting analysis, extensions to the case where \( X \) is an infinite set, as well as computational results involving several types of problems, with both discrete and continuous state and control spaces.

We mention that the idea of using multiple base policies to evaluate the available controls at a given state, and selecting the control that yields the least cost, has been known since the original proposal of the paper [BTW97]. The main result for such schemes is a cost improvement property, whereby the rollout policy outperforms simultaneously all the base policies; cf. Eq. (5.15). This property is also discussed in the Appendix (Sections 9.3 and 9.4), as well as the books [Ber17a], [Ber19a], [Ber20a].

Finally, let us note 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.

**State Constraints, Target Tubes, 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 we have discussed; cf. Eqs. (2.5)-(2.6).

The treatment of state constraints is connected to the theory of reachability of target tubes, first studied by the author in his Ph.D. thesis [Ber71], and subsequent papers [BeR71], [Ber72]; see the books [Ber17a], [Ber19a], [Ber20a] for a discussion that is consistent with the viewpoint of this section. Target tubes consist of subsets \( \tilde{X} \) of the state constraint set \( X \), within which the state can be kept indefinitely with feasible control choices, assuming that the initial state belongs to \( \tilde{X} \). In other words, the problem (5.10) may not be feasible for every \( x_k \in X \), once the constraint \( x_t \in X \) for all \( t = k + 1, k + 2, \ldots \), is added in problem (5.10). However, a suitable target tube is one specified by a subset \( \tilde{X} \subset X \) such that the problem (5.10) is feasible under the constraint \( x_t \in \tilde{X} \) for all \( t = k + 1, k + 2, \ldots \), provided \( x_k \in \tilde{X} \).

There are several ways to compute sets \( \tilde{X} \) with this property, 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]. The important point here is that the computation of a target tube 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 $G$).

Given an off-line training process, which provides a target tube constraint $x_k \in \tilde{X}$ for all $k$, a terminal cost function $G$, and possibly one or more base policies for truncated rollout, MPC becomes an on-line play algorithm for which our earlier discussion applies. Significantly, MPC is an effective way to deal with indirect adaptive control contexts, where a model is estimated on-line as it is changing; see our discussion of Section 5.1.

6. CONCLUDING REMARKS

While the ideas of approximation in value space, rollout, and PI have a long history, their significance has been highlighted by the success of AlphaZero, and the earlier but just as impressive TD-Gammon program. Both programs were trained off-line extensively using sophisticated approximate PI algorithms and neural networks. Yet the players obtained off-line were greatly improved by on-line play, as we have discussed.

We have argued that this performance enhancement by on-line play defines a new paradigm for decision and control, which is couched on the AlphaZero/TD-Gammon design principles: on-line decision making, using approximation in value space with multistep lookahead, and rollout. There is an additional benefit of policy improvement by approximation in value space, not observed in the context of games (which have stable rules and environment). It is well-suited for on-line replanning and changing problem parameters, as in the context of indirect adaptive control, and also MPC, which in fact embodies several of the AlphaZero/TD-Gammon design ideas.

In this work, we have aimed to provide the mathematical framework, analysis, and insights (often based on visualization), which facilitate the use of on-line decision making on top of off-line training. In particular, through a unified abstract DP analysis, we have shown that the principal ideas of approximation in value space and rollout apply very broadly to deterministic and stochastic optimal control problems, involving both discrete and continuous search spaces. These ideas can be effectively integrated with adaptive control, MPC, and other important methodologies such as decentralized and multiagent control, discrete and Bayesian optimization, neural network-based value and policy approximations, and heuristic algorithms for discrete optimization, as we have discussed in greater detail in the books [Ber19a] and [Ber20a]. We have also emphasized that while the ideas of on-line play and off-line training are implicit in several decision and control contexts, and particularly MPC, much remains to be gained by a more systematic view of the dichotomy between off-line training and on-line play, and by its incorporation into control system design contexts.

A key idea of this work is the interpretation of approximation in value space with one-step lookahead
Concluding Remarks

as a step of Newton’s method. This idea has been known for a long time within the more restrictive contexts of policy iteration and rollout. The extensions of this idea, including multistep lookahead and truncated rollout, which are provided in this work, aim to promote the view that Newton’s method and other classical algorithms, such as Newton-SOR, are central conceptual elements of the RL methodology.

A major supplementary idea of this work is our interpretation of off-line training of policies and cost approximations as means for enhancement of the initial condition of the Newton step. Among others, this interpretation supports the idea that the Newton step/on-line player is the key determinant of the overall scheme’s performance, and that the initial condition adjustment/off-line training plays a subsidiary role.

Finally, we have noted that while our focus in this work has been on infinite horizon problems, approximation in value space and rollout can be applied to finite horizon problems as well, and can be similarly interpreted in terms of Newton’s method. One way to approach finite horizon problems analytically is to convert them to infinite horizon stochastic shortest path problems with a termination state that corresponds to the end of the horizon. Once this is done, the conceptual framework of the present work can be applied to provide insight on the connections between approximation in value space, rollout, and Newton’s method. In particular, our ideas find application beyond the infinite horizon DP context, and apply to the solution of classical discrete and combinatorial optimization problems, which can be transformed to sequential finite horizon optimization problems; this was the basis for the original proposal of the use of rollout for discrete and combinatorial optimization problems in the paper [BTW97]. Rollout has been used with great success for such problems in the past. We have summarized the main ideas in an appendix, and we refer to the book [Ber20a] for a fuller presentation.

The book [Ber20a] also contains many examples of application of rollout to discrete optimization and provides references to many works spanning the period from the late 90s to the present. These works 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.
7. REFERENCES


mont, MA.

1368-1374.


Rensselaer Polytechnic Institute.


Uncertainty,” Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, MA (can be downloaded 
from the author’s website).


pp. 610-616.

ning, Vol. 27, pp. 107-120.


Scientific, Belmont, MA.


Scientific, Belmont, MA.


(can be downloaded from the author’s website).

and Some New Implementations,” Lab. for Information and Decision Systems Report, MIT; arXiv preprint


References


References


8. APPENDIX: FINITE HORIZON DETERMINISTIC PROBLEMS - DISCRETE OPTIMIZATION

In this appendix, we discuss finite horizon deterministic problems, focusing primarily on the case where the state and control spaces are finite. After we introduce these problems, we will argue that they can be transformed to infinite horizon SSP problems, through the use of an artificial cost-free termination state $t$ that the system moves into at the end of the horizon. Once the problem is transformed to an infinite horizon SSP problem, the ideas of approximation in value space, off-line training and on-line play, and Newton’s method, which we have developed earlier, become applicable. Moreover the ideas of MPC are easily adapted within the finite horizon discrete optimization framework.

An interesting aspect of our methodology for discrete deterministic problems is that it admits generalizations that we have not discussed so far. These include 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.

Another interesting aspect of discrete deterministic problems is that they can serve as a framework for an important class of commonly encountered discrete optimization problems, including integer programming/combinatorial problems such as scheduling, assignment, routing, etc. This will bring to bear the methodology of rollout, approximation in value space, and MPC, and provide effective suboptimal solution methods for this class of problems. The present appendix will provide a brief summary, aimed to make the connection with approximation in value space and Newton’s method. Additional discussion may be found in the author’s rollout and policy iteration book [Ber20a], on which this appendix is based.

8.1 Deterministic Discrete Spaces Finite Horizon Problems

In deterministic finite horizon DP problems, the state is generated nonrandomly over $N$ stages, and involves a system of the form

$$x_{k+1} = f_k(x_k, u_k), \quad k = 0, 1, \ldots, N - 1,$$  \hspace{1cm} (8.1)

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 $k + 1$. 

90
The state space $X_k$ and control space $U_k$ can be any 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

$$J(x_0; u_0, \ldots, u_{N-1}) = g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k),$$  \hspace{1cm} (8.2)$$

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 (8.1); see Figure 8.2. We want to minimize the cost (8.2) over all sequences \{u_0, \ldots, u_{N-1}\} that satisfy the control constraints, thereby obtaining the optimal value as a function of $x_0$

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

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 $\mu_k$ 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 appendix: instead of using a policy for rollout, we will allow the use of more general heuristics for choosing future controls.

Discrete Optimal Control Problems

There are many situations where the control space is naturally discrete and consists of a finite number of elements. In this appendix, whenever we assume that the control space is finite, we will also assume implicitly
Figure 8.3 Transition graph for a deterministic system with finite control space and a finite number of initial states. 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 transition cost $g_k(x_k, u_k)$ is viewed as the length of this arc. The problem is equivalent to finding a shortest path from initial nodes of stage 0 to an artificial terminal node $t$.

Appendix: Finite Horizon Deterministic Problems - Discrete Optimization

A single or at most a finite number of possible initial states, so the number of states that can be generated at each stage is also finite. A problem of this type can be 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. 8.3. 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 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. It also turns out that the reverse is true: every shortest path problem involving a graph whose cycles have positive length can be transformed into a discrete optimal control problem. This fact is important, but will not be useful to us, so we will not consider it further here (see the textbook [Ber17a], Chapter 2, for a detailed discussion).

The connection of finite state and control spaces finite horizon deterministic problem with a shortest path problem is important for our purposes, because it provides the bridge to an SSP problem with an infinite horizon, and by extension, with our earlier development of approximation in value space, Newton’s method, rollout, and the PI algorithm. It is also important to recognize that this SSP problem has a few additional special characteristics. These are:
Appendix: Finite Horizon Deterministic Problems - Discrete Optimization

(a) The equivalent SSP involves a deterministic system, has a finite number of states and controls, and involves an acyclic graph. The states of the SSP are the state-time pairs \((x_k, k)\), \(k = 0, 1, \ldots, N\), where \(x_k\) is one of the finite number of elements of \(X_k\) that are reachable from one of the finite number of initial states \(x_0\) using a feasible sequence of controls. The possible transitions from states \((x_k, k)\) to states \((x_{k+1}, k+1)\) correspond to controls \(u_k \in U_k(x_k)\) such that \(x_{k+1} = f_k(x_k, u_k)\).

(b) The state space of the SSP tends to expand as the horizon \(N\) becomes longer. While this tends to complicate the use of the PI algorithm, it does not materially affect the use of rollout.

(c) The optimal cost function of the SSP is obtained from the optimal cost functions of the finite horizon problem, which are generated by the DP algorithm to be presented shortly. This DP algorithm can be viewed as the Bellman equation of the SSP.

The Exact Dynamic Programming Algorithm

The DP algorithm for finite horizon deterministic problems rests on a simple idea, the principle of optimality, which 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.

By translating into mathematical terms the principle of optimality, we obtain the DP algorithm. It 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. 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 a state \(x_N\).

<table>
<thead>
<tr>
<th>DP Algorithm for Deterministic Finite Horizon Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start with</td>
</tr>
<tr>
<td>[ J^*_N(x_N) = g_N(x_N), \quad \text{for all } x_N, ] (8.3)</td>
</tr>
<tr>
<td>and for (k = 0, \ldots, N-1), let</td>
</tr>
<tr>
<td>[ J^<em><em>k(x_k) = \min</em>{u_k \in U_k(x_k)} \left[ g_k(x_k, u_k) + J^</em>_{k+1}(f_k(x_k, u_k)) \right], \quad \text{for all } x_k. ] (8.4)</td>
</tr>
</tbody>
</table>

Note that at stage \(k\), the calculation in Eq. (8.4) must be done for all states \(x_k\) before proceeding to
stage $k - 1$. The key fact about the DP algorithm is that for every initial state $x_0$, the number $J^*_0(x_0)$ obtained at the last step, is equal to the optimal cost $J^*(x_0)$. Indeed, a more general fact can be shown, namely that for all $k = 0, 1, \ldots, N - 1$, and all states $x_k$ at time $k$, we have

$$J^*_k(x_k) = \min_{u_m \in U_m(x_m)} J(x_k; u_k, \ldots, u_{N-1}),$$  \hspace{1cm} (8.5)$$

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

$$J(x_k; u_k, \ldots, u_{N-1}) = g_k(x_k, u_k) + g_{k+1}(x_{k+1}, u_{k+1}) + \cdots + g_{N-1}(x_{N-1}, u_{N-1}) + g_N(x_N).$$  \hspace{1cm} (8.6)$$

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 (8.5) 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 $k$.

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

$$u_0^* \in \arg \min_{u_0 \in U_0(x_0)} \left[ g_0(x_0, u_0) + J^*_1(f_0(x_0, u_0)) \right],$$

and

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

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

$$u_k^* \in \arg \min_{u_k \in U_k(x_k^*)} \left[ g_k(x_k^*, u_k) + J^*_{k+1}(f_k(x_k^*, u_k)) \right],$$ \hspace{1cm} (8.7)$$

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. (8.3)-(8.4)], which is followed by “on-line play” in real-time to obtain $u_k^*$ [cf. Eq. (8.7)]. This is analogous to the two algorithmic processes described in Section 1 in connection with chess and backgammon.
Appendix: Finite Horizon Deterministic Problems - Discrete Optimization

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. This formulation often leads to an intractable DP computation because of an exponential explosion of the number of states as time progresses. However, a DP formulation 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. We illustrate the reformulation by an example and then generalize.

Example 8.1 (The Traveling Salesman Problem)

An important model for scheduling a sequence of operations is the classical traveling salesman problem. Here we are given \( N \) cities and the travel time between each pair of cities. We wish to find a minimum time travel that visits each of the cities exactly once and returns to the start city. To convert this problem to a DP problem, we form a graph whose nodes are the sequences of \( k \) distinct cities, where \( k = 1, \ldots, N \). The \( k \)-city sequences correspond to the states of the \( k \)th stage. The initial state \( x_0 \) consists of some city, taken as the start (city A in the example of Fig. 8.4). 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. 8.4. Each sequence of \( N \) cities is connected to an artificial terminal node \( t \) with an arc of cost equal to the travel time from the last city of the sequence to the starting city, thus completing the transformation to a DP problem.

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

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

\[
\begin{align*}
\text{minimize} & \quad G(u) \\
\text{subject to} & \quad u \in U,
\end{align*}
\]

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 \( k \)th stage of the finite horizon DP problem shown in Fig. 8.5. In particular, for \( k = 1, \ldots, N \), we view as the states of the \( k \)th 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

\[
U_0 = \{ u_0 \mid \text{there exists a solution of the form } (u_0, \tilde{u}_1, \ldots, \tilde{u}_{N-1}) \in U \}.
\]
Thus $U_0$ is the set of choices of $u_0$ that are consistent with feasibility.

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

$$U_k(u_0, \ldots, u_{k-1}) = \{ u_k \mid \text{for some } \pi_{k+1}, \ldots, \pi_{N-1} \text{ we have } (u_0, \ldots, u_{k-1}, u_k, \pi_{k+1}, \ldots, \pi_{N-1}) \in U \}.$$  \hfill (8.9)

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. 8.5. All other transitions in this DP problem formulation have cost 0.

Let $J_k^*(u_0, \ldots, u_{k-1})$ denote the optimal cost starting from the $k$-solution $(u_0, \ldots, u_{k-1})$, i.e., the optimal cost of the problem over solutions whose first $k$ components are constrained to be equal to $u_0, \ldots, u_{k-1}$.
Appendix: Finite Horizon Deterministic Problems - Discrete Optimization

Figure 8.5 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.

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

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

(8.10)

where $U_0$ is given by Eq. (8.8), and $U_k$ is given by Eq. (8.9): first compute $u_0^*$, then $u_1^*$, and so on up to $u_{N-1}^*$; cf. Eq. (8.7).

Of course here the number of states typically grows exponentially with $N$, but we can use the DP minimization (8.10) as a starting point for the use of approximation methods. For example we may try to use approximation in value space, whereby we replace $J^*_{k+1}$ with some suboptimal $\tilde{J}_{k+1}$ in Eq. (8.10). 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.
Appendix: Finite Horizon Deterministic Problems - Discrete 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 structure, which is consistent with a sequential decision making process. The traveling salesman Example 8.1 is a case in point, where $G$ consists of $N$ components (the intercity travel costs), one per stage.

8.2 Approximation in Value Space

The forward optimal control sequence construction of Eq. (8.7) 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 idea of approximation in value space that we have encountered earlier in connection with infinite horizon problems. 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 procedure (8.7). Based on our infinite horizon analysis of Section 3 and the interpretation of the deterministic finite horizon problem as an infinite horizon SSP, the cost function of the corresponding one-step lookahead policy can be viewed as the result of a Newton step for solving Bellman’s equation, i.e., the DP algorithm (8.4), starting from the point $(\tilde{J}_1, \tilde{J}_2, \ldots, \tilde{J}_N)$.

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

Start with

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

and set

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

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

$$\tilde{u}_k \in \arg \min_{u_k \in U_k(\tilde{x}_k)} \left[ g_k(\tilde{x}_k, u_k) + \tilde{J}_{k+1}(f_k(\tilde{x}_k, u_k)) \right], \quad (8.11)$$

and

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

Thus 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 $\{u^*_0, \ldots, u^*_{N-1}\}$ [cf. Eq. (8.7)], and is independent of how the functions $\tilde{J}_k$ are computed.
An alternative (and equivalent) form of the DP algorithm (8.4), 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

\[
Q^*_k(x_k, u_k) = g_k(x_k, u_k) + J^*_{k+1}(f_k(x_k, u_k)).
\]

(8.12)

Thus the optimal Q-factors are simply the expressions that are minimized in the right-hand side of the DP equation (8.4).

Note that the optimal cost function \( J^*_k \) can be recovered from the optimal Q-factor \( Q^*_k \) by means of the minimization

\[
J^*_k(x_k) = \min_{u_k \in U_k(x_k)} Q^*_k(x_k, u_k).
\]

(8.13)

Moreover, the DP algorithm (8.4) can be written in an essentially equivalent form that involves Q-factors only [cf. Eqs. (8.12)-(8.13)]:

\[
Q^*_k(x_k, u_k) = g_k(x_k, u_k) + \min_{u_{k+1} \in U_{k+1}(f_k(x_k, u_k))} Q^*_{k+1}(f_k(x_k, u_k), u_{k+1}).
\]

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.

The expression

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

which is minimized in approximation in value space [cf. Eq. (8.11)] is known as the (approximate) Q-factor of \((x_k, u_k)\). Note that the computation of the suboptimal control (8.11) can be done through the Q-factor minimization

\[
\tilde{u}_k \in \arg \min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k).
\]

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 (8.11) 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 (8.11) involves a one-step lookahead minimization, since it solves a one-stage DP problem for each \(k\). We may also consider \(\ell\)-step lookahead, which involves the solution of an \(\ell\)-step DP problem, where \(\ell\) is an integer, \(1 < \ell < N - k\), with a terminal cost function approximation \(\tilde{J}_{k+\ell}\). This is similar to the infinite horizon case that we discussed in Section 2.2. As we have noted in that section, multistep lookahead typically 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. On the negative side, the solution of the multistep lookahead optimization problem, instead of the one-step lookahead counterpart of Eq. (8.11), becomes more time consuming.
Appendix: Finite Horizon Deterministic Problems - Discrete Optimization

**Rollout**

Similar to infinite horizon problems, a major issue in the value space approximation (8.11) is the construction of suitable approximate cost-to-go functions $\tilde{J}_{k+1}$. This can be done in many different ways, including some of the principal RL methods. For example, $\tilde{J}_{k+1}$ may be constructed with a sophisticated off-line training method, as discussed in Section 1.1, in connection with chess and backgammon. Forms of approximate PI method can be applied in particular, possibly with the use of neural networks, once the problem is viewed as an infinite horizon SSP problem. Another possibility is the fitted value iteration method, which is described in Section 4.3 of the book [Ber19a], and Section 4.3.1 of the book [Ber20a].

Alternatively, $\tilde{J}_{k+1}$ may be obtained on-line with *rollout*, whereby the approximate values $\tilde{J}_{k+1}(x_{k+1})$ are obtained when needed by running a heuristic control scheme, called *base heuristic*, for a suitably large number of steps, starting from $x_{k+1}$.† The base heuristic need not be a policy. It could be any method, which starting from a state $x_{k+1}$ generates a sequence 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 appendix:

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

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 (8.11). An important point here is that deterministic problems hold a special attraction for rollout, as they do not require expensive on-line Monte Carlo simulation to calculate the cost function values $\tilde{J}_{k+1}(x_{k+1})$.

There are also several variants of rollout, involving for example truncation, multistep lookahead, and other possibilities. In particular, truncated rollout 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 heuristic (i.e., $m = 0$), one recovers as a special case the general

† For deterministic problems we prefer to use the term “base heuristic” rather than “base policy” for reasons to be explained later in this section, in the context of the notion of sequential consistency (the heuristic may not qualify as a legitimate DP policy). In particular, if the base heuristic, when stated at state $x_k$, generates the sequence \{$\tilde{u}_k, \tilde{x}_{k+1}, \tilde{u}_{k+1}, \tilde{x}_{k+2}, \ldots, \tilde{u}_{N-1}, \tilde{x}_N$\}, it is not necessarily true that, when stated at state $\tilde{x}_{k+1}$, it will generate the sequence \{$\tilde{u}_{k+1}, \tilde{x}_{k+2}, \ldots, \tilde{u}_{N-1}, \tilde{x}_N$\} (which would be true if the heuristic were a legitimate policy). More generally, the method used by the base heuristic to complete the system’s trajectory starting from some state may be very different than the method used to complete the trajectory starting at another state. In any case, if the base heuristic is not a legitimate policy, then the use of $H_{k+1}(x_{k+1})$ as terminal cost function approximation, yields a type of approximation in value space scheme, which can still be interpreted as a Newton step.
Appendix: Finite Horizon Deterministic Problems - Discrete Optimization

Figure 8.7 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

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

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

approximation in value space scheme. Versions of truncated rollout with multistep lookahead minimization are also possible. Other variants of rollout include versions involving multiple heuristics, combinations with other forms of approximation in value space methods, and multistep lookahead, which will be described later in this appendix. We next discuss variants of rollout in greater detail.

8.3 Rollout Algorithms for Discrete Optimization

We will now develop in more detail the theory of rollout for deterministic problems, including the central issue of cost improvement. We will also illustrate several variants of the method, and we will consider issues of efficient implementation. We will then discuss examples of discrete optimization applications.

Let us consider a deterministic DP problem with a finite number of controls and a given initial state (so the number of states that can be reached from the initial state is also finite). We first focus on the pure form of rollout that uses one-step lookahead and no terminal cost approximation. 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

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

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 \( N \):

\[
g_k(x_k, u_k) + H_{k+1}(x_{k+1}).
\]

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

\[
\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k), \tag{8.14}
\]

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

\[
\tilde{Q}_k(x_k, u_k) = g_k(x_k, u_k) + H_{k+1}(f_k(x_k, u_k)); \tag{8.15}
\]

see Fig. 8.7. Rollout 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 (8.14).

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 8.2 (Traveling Salesman Problem)

Let us consider the traveling salesman problem of Example 8.1, 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 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 \( \{c_0, c_1, \ldots, c_k\} \) 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
\[
g(c_0, c_1) + \cdots + g(c_{N-2}, c_{N-1}) + g(c_{N-1}, c_0).
\] (8.16)

We can formulate the traveling salesman problem as a DP problem as we discussed in Example 8.1. We choose a starting city, say \( c_0 \), as the initial state \( x_0 \). Each state \( x_k \) corresponds to a partial tour \( (c_0, c_1, \ldots, c_k) \)
Appendix: Finite Horizon Deterministic Problems - Discrete Optimization

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. (8.16). 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 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. 8.8. The overall computation for the rollout solution is bounded by a polynomial in $N$, and is much smaller than the exact DP computation. Figure 8.9 provides an example where the nearest neighbor heuristic and the corresponding rollout algorithm are compared.

Cost Improvement with Rollout - Sequential Consistency

The definition of the rollout algorithm leaves open the choice of the base heuristic. There are several types of suboptimal solution methods that can be used as base heuristics, such as greedy algorithms, local search, genetic algorithms, 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, and we later show how to modify the algorithm to deal with the case where these conditions are not met.

Definition 8.1: 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 $x_{k+1}$. 

104
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 8.2 is sequentially consistent. Similar examples include the use of many types of greedy/myopic heuristics (Section 6.4 of the book [Ber17a] provides some additional examples). Generally most heuristics used in practice satisfy the sequential consistency condition at “most” states $x_k$. However, some heuristics of interest may violate this condition at some states.

A sequentially consistent base heuristic can be recognized by the fact that it will apply the same control $u_k$ at a state $x_k$, no matter what position $x_k$ occupies in a trajectory generated by the base heuristic. Thus a base heuristic is sequentially consistent if and only if it defines a legitimate DP policy. This is the policy that moves from $x_k$ to the state $x_{k+1}$ that lies on the state trajectory $\{x_k, x_{k+1}, \ldots, x_N\}$ that the base heuristic generates.

We will now show that the rollout algorithm obtained with a sequentially consistent base heuristic yields no worse cost than the base heuristic.

**Proposition 8.1: (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

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

(8.17)

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 $\pi_k$ be the control applied by the base heuristic at $x_k$. Then we have

$$J_{k,\tilde{\pi}}(x_k) = g_k(x_k, \tilde{\mu}_k(x_k)) + J_{k+1,\tilde{\pi}}(f_k(x_k, \tilde{\mu}_k(x_k)))$$

$$\leq g_k(x_k, \tilde{\mu}_k(x_k)) + H_{k+1}(f_k(x_k, \tilde{\mu}_k(x_k)))$$

$$= \min_{u_k \in U_k(x_k)} \left[ g_k(x_k, u_k) + H_{k+1}(f_k(x_k, u_k)) \right]$$

(8.18)

$$\leq g_k(x_k, \pi_k) + H_{k+1}(f_k(x_k, \pi_k))$$

$$= H_k(x_k),$$

105
where:

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

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

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

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

This completes the induction proof of the cost improvement property (8.17). \textbf{Q.E.D.}

\textit{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. This is a sequential improvement condition that is related to the one we have discussed earlier [cf. Eq. (3.25)]. Let us recall that the rollout algorithm $\tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$ is defined by the minimization

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

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

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

[cf. Eqs. (8.15)], and $H_{k+1} (f_k(x_k, u_k))$ denotes the cost of the trajectory of the base heuristic starting from state $f_k(x_k, u_k)$.

\begin{definition}
We say that the base heuristic is \textit{sequentially improving} if for all $x_k$ and $k$, we have

$$\min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k) \leq H_k(x_k). \quad (8.19)$$
\end{definition}

In words, the sequential improvement property (8.19) states that

Minimal heuristic Q-factor at $x_k$ \leq Heuristic cost at $x_k$.

Note that \textit{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
Appendix: Finite Horizon Deterministic Problems - Discrete Optimization

Q-factor of the control $\mathbf{u}_k$ that the heuristic applies at $x_k$,

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

which is greater or equal to the minimal Q-factor at $x_k$. This implies Eq. (8.19). We will now show that a sequentially improving heuristic yields policy improvement.

**Proposition 8.2: (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

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

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

**Proof:** Follows from the calculation of Eq. (8.18), by replacing the last two steps (which rely on sequential consistency) with Eq. (8.19). \textbf{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. 8.10) that

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

(8.20)

107
Figure 8.10 Proof of the monotonicity property (8.20). At $\tilde{x}_k$, the $k$th 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 $Q_k(\tilde{x}_k, \tilde{u}_k)$ of the base heuristic starting from $(\tilde{x}_k, \tilde{u}_k)$. The sequential improvement condition guarantees that

$$H_k(\tilde{x}_k) \geq Q_k(\tilde{x}_k, \tilde{u}_k),$$

which implies that

$$\text{Cost of } T_k \geq \text{Cost of } T_{k+1}.$$
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

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

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 \( \tilde{u}_1 \) at \( x_1^* \) (there is nothing to prevent this from happening, since the base heuristic is arbitrary), and moreover the cost \( g_1(x_1^*, \tilde{u}_1) + g_2(\tilde{x}_2) \), which is equal to \( H_1(x_1^*) \) is high enough.

Let us also verify that if the inequality (8.21) holds then the heuristic is not sequentially improving at \( x_0 \), i.e., that

\[
H_0(x_0) < \min \left\{ g_0(x_0, u_0^*) + H_1(x_1^*), \; g_0(x_0, \tilde{u}_0) + H_1(\tilde{x}_1) \right\}.
\]

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 (8.20) 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.

Figure 8.11 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 \( x_0 \).
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 \(\ell\)th of these, given a state \(x_{k+1}\), produces a trajectory

\[
\hat{T}_k^\ell = \{x_{k+1}, \hat{u}_k, x_{k+2}, \ldots, \hat{x}_{N-1}, \hat{x}_N\},
\]

and corresponding cost \(C(\hat{T}_k^\ell)\). The superheuristic then produces at \(x_{k+1}\) the trajectory \(\hat{T}_k^\ell\) for which \(C(\hat{T}_k^\ell)\) is minimum. The rollout algorithm selects at state \(x_k\) the control \(u_k\) that minimizes the minimal Q-factor:

\[
\hat{u}_k \in \arg\min_{u_k \in U_k(x_k)} \min_{\ell = 1, \ldots, m} \hat{Q}_k^\ell(x_k, u_k),
\]

where

\[
\hat{Q}_k^\ell(x_k, u_k) = g_k(x_k, u_k) + C(\hat{T}_k^\ell)
\]

is the cost of the trajectory \((x_k, u_k, \hat{T}_k^\ell)\). 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. 8.10. Indeed, let us write the sequential improvement condition (8.19) for each of the base heuristics

\[
\min_{u_k \in U_k(x_k)} \hat{Q}_k^\ell(x_k, u_k) \leq H_k^\ell(x_k), \quad \ell = 1, \ldots, m,
\]

and all \(x_k\) and \(k\), where \(\hat{Q}_k^\ell(x_k, u_k)\) and \(H_k^\ell(x_k)\) are Q-factors and heuristic costs that correspond to the \(\ell\)th heuristic. Then by taking minimum over \(\ell\), we have

\[
\min_{\ell = 1, \ldots, m} \left\{ \min_{u_k \in U_k(x_k)} \hat{Q}_k^\ell(x_k, u_k) \right\} \leq \min_{\ell = 1, \ldots, m} H_k^\ell(x_k),
\]

† 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.

110
for all \( x_k \) and \( k \). By interchanging the order of the minimizations of the left side, we then obtain

\[
\min_{u_k \in U_k(x_k)} \min_{\ell=1, \ldots, m} \tilde{Q}_k^\ell(x_k, u_k) \leq \min_{\ell=1, \ldots, m} H_k^\ell(x_k),
\]

which is precisely the sequential improvement condition (8.19) for the superheuristic.

**Simplified Rollout Algorithms**

We will now consider a rollout variant, called *simplified rollout algorithm*, 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

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

[cf. Eqs. (8.14) and (8.15)], may be unwieldy, since the number of Q-factors

\[
\tilde{Q}_k(x_k, u_k) = g_k(x_k, u_k) + H_{k+1}(f_k(x_k, u_k))
\]
is accordingly infinite or large.

To remedy this situation, we may replace \( U_k(x_k) \) with a smaller finite subset \( \bar{U}_k(x_k) \):

\[
\bar{U}_k(x_k) \subset U_k(x_k).
\]
The rollout control \( \bar{\mu}_k(x_k) \) in this variant is one that attains the minimum of \( \tilde{Q}_k(x_k, u_k) \) over \( u_k \in \bar{U}_k(x_k) \):

\[
\bar{\mu}_k(x_k) \in \arg \min_{u_k \in \bar{U}_k(x_k)} \tilde{Q}_k(x_k, u_k).
\]

An example is when \( \bar{U}_k(x_k) \) 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 \( \bar{U}_k(x_k) \) of promising controls, and to save computation, we restrict attention to this subset. A related possibility is to generate \( \bar{U}_k(x_k) \) 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. (8.22)].

It turns out that the proof of the cost improvement property of Prop. 8.2,

\[
J_{k,\#}(x_k) \leq H_k(x_k), \quad \text{for all } x_k \text{ and } k,
\]
goes through if the following modified sequential improvement property holds:

\[
\min_{u_k \in \bar{U}_k(x_k)} \tilde{Q}_k(x_k, u_k) \leq H_k(x_k).
\]

This can be seen by verifying that Eq. (8.24) is sufficient to guarantee that the monotone improvement Eq. (8.20) is satisfied. The condition (8.24) is very simple to satisfy if the base heuristic is sequentially consistent, in which case the control \( \bar{\pi}_k \) selected by the base heuristic satisfies

\[
\tilde{Q}_k(x_k, \bar{\pi}_k) = H_k(x_k).
\]
Appendix: Finite Horizon Deterministic Problems - Discrete Optimization

In particular, for the property (8.24) to hold, it is sufficient that $\overline{U}(x_k)$ contains the base heuristic choice $\overline{u}_k$.

The idea of replacing the minimization (8.22) by the simpler minimization (8.23) 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 $\tilde{Q}_k(x_k, \tilde{\mu}_k(x_k)) \leq H_k(x_k)$, for all $x_k$ and $k$, guarantees the modified sequential improvement property (8.24), 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 $U^1_k(x_k) \times \cdots \times U^m_k(x_k)$ is replaced by a sequence of single component minimizations, one-component-at-a-time; cf. Section 3.5.

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 $P_k = \{x_0, u_0, \ldots, u_{k-1}, x_k\}$ that has been constructed by rollout, called permanent trajectory, and we also store a tentative best trajectory $T_k = \{x_0, u_0, \ldots, u_{k-1}, x_k, \overline{u}_k, \overline{x}_{k+1}, \overline{u}_{k+1}, \ldots, \overline{u}_{N-1}, \overline{x}_N\}$ with corresponding cost

$$C(T_k) = \sum_{t=0}^{k-1} g_t(x_t, u_t) + g_k(x_k, \overline{u}_k) + \sum_{t=k+1}^{N-1} g_t(\overline{x}_t, \overline{u}_t) + g_N(\overline{x}_N).$$

The tentative best trajectory $T_k$ is the best end-to-end trajectory computed up to stage $k$ of the algorithm. Initially, $T_0$ 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 $T_k$, and use $T_k$ instead (see Fig. 8.12).†

† The fortified rollout algorithm can actually be viewed as the ordinary rollout algorithm applied to a modified version of the original problem and modified base heuristic that has the sequential improvement property. This construction is somewhat technical and unintuitive and will not be given; we refer to Bertsekas, Tsitsiklis, and Wu [BTW97], and the DP textbook [Ber17a], Section 6.4.2.
Figure 8.12  Schematic illustration of fortified rollout. After \( k \) steps, we have constructed the permanent trajectory

\[
\mathcal{P}_k = \{x_0, u_0, \ldots, u_{k-1}, x_k\},
\]

and the tentative best trajectory

\[
\mathcal{T}_k = \{x_0, u_0, \ldots, u_{k-1}, x_k, \overline{u}_k, x_{k+1}, \overline{u}_{k+1}, \ldots, \overline{u}_{N-1}, x_N\},
\]

the best end-to-end trajectory computed so far. We now run the rollout algorithm at \( x_k \), i.e., we find the control \( \overline{u}_k \) 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

\[
\mathcal{T}_k = \{x_0, u_0, \ldots, u_{k-1}, x_k, \overline{u}_k, x_{k+1}, \overline{u}_{k+1}, \ldots, \overline{u}_{N-1}, x_N\}. 
\]

If the cost of the end-to-end trajectory \( \mathcal{T}_k \) is lower than the cost of \( \mathcal{T}_k \), we add \((\overline{u}_k, \overline{x}_{k+1})\) to the permanent trajectory and set the tentative best trajectory to \( \mathcal{T}_{k+1} = \mathcal{T}_k \). Otherwise we add \((\overline{u}_k, \overline{x}_{k+1})\) to the permanent trajectory and keep the tentative best trajectory unchanged: \( \mathcal{T}_{k+1} = \mathcal{T}_k \).

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 \( \overline{u}_k \) that gives the best trajectory, denoted

\[
\mathcal{T}_k = \{x_0, u_0, \ldots, u_{k-1}, x_k, \overline{u}_k, x_{k+1}, \overline{u}_{k+1}, \ldots, \overline{u}_{N-1}, x_N\}
\]

with corresponding cost

\[
C(\mathcal{T}_k) = \sum_{t=0}^{k-1} g_t(x_t, u_t) + g_k(x_k, \overline{u}_k) + \sum_{t=k+1}^{N-1} g_t(x_t, \overline{u}_k) + g_N(x_N).
\]

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

\[
x_{k+1} = \overline{x}_{k+1}, \quad \mathcal{T}_{k+1} = \mathcal{T}_k,
\]

113
and if \( C(\mathcal{T}_k) > C(\tilde{\mathcal{T}}_k) \) it sets the next state and corresponding tentative best trajectory to

\[
x_{k+1} = \tilde{x}_{k+1}, \quad \tilde{T}_{k+1} = \tilde{T}_k.
\]

In other words the fortified rollout at \( x_k \) follows the current tentative best trajectory \( \mathcal{T}_k \) 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 8.4**

Let us consider the application of the fortified rollout algorithm to the problem of Example 8.3 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.,

\[
u_k = (u_k^1, \ldots, u_k^n), \quad U_k(x_k) = U_k^1(x_k) \times \cdots \times U_k^n(x_k).
\]

Then, to take advantage of distributed computation, it may be attractive to decompose the optimization over \( u_k \) in the rollout algorithm,

\[
\hat{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \left[ g_k(x_k, u_k) + H_{k+1}(f_k(x_k, u_k)) \right],
\]

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.\[\hat{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \left[ g_k(x_k, u_k) + H_{k+1}(f_k(x_k, u_k)) \right],\]

\[\hat{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \left[ g_k(x_k, u_k) + H_{k+1}(f_k(x_k, u_k)) \right],\]

\[\hat{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \left[ g_k(x_k, u_k) + H_{k+1}(f_k(x_k, u_k)) \right],\]

\[\hat{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \left[ g_k(x_k, u_k) + H_{k+1}(f_k(x_k, u_k)) \right],\]

\[\hat{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \left[ g_k(x_k, u_k) + H_{k+1}(f_k(x_k, u_k)) \right],\]

\[\hat{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \left[ g_k(x_k, u_k) + H_{k+1}(f_k(x_k, u_k)) \right],\]

\[\hat{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \left[ g_k(x_k, u_k) + H_{k+1}(f_k(x_k, u_k)) \right],\]

\[\hat{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \left[ g_k(x_k, u_k) + H_{k+1}(f_k(x_k, u_k)) \right],\]

\[\hat{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \left[ g_k(x_k, u_k) + H_{k+1}(f_k(x_k, u_k)) \right],\]

\[\hat{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \left[ g_k(x_k, u_k) + H_{k+1}(f_k(x_k, u_k)) \right],\]

\[\hat{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \left[ g_k(x_k, u_k) + H_{k+1}(f_k(x_k, u_k)) \right],\]

\[\hat{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \left[ g_k(x_k, u_k) + H_{k+1}(f_k(x_k, u_k)) \right],\]

\[\hat{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \left[ g_k(x_k, u_k) + H_{k+1}(f_k(x_k, u_k)) \right],\]

\[\hat{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \left[ g_k(x_k, u_k) + H_{k+1}(f_k(x_k, u_k)) \right],\]

\[\hat{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \left[ g_k(x_k, u_k) + H_{k+1}(f_k(x_k, u_k)) \right],\]

\[\hat{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \left[ g_k(x_
even if 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.

**Rollout with Multistep Lookahead - Truncated Rollout**

We will now consider incorporating multistep lookahead into the rollout framework. To describe two-step lookahead for deterministic problems, suppose that after \( k \) steps we have reached state \( x_k \). We then consider the set of all possible two-step-ahead states \( x_{k+2} \), we run the base heuristic starting from each of them, and compute the two-stage cost to get from \( x_k \) to \( x_{k+2} \), plus the cost of the base heuristic from \( x_{k+2} \). We select the state, say \( \tilde{x}_{k+2} \), that is associated with minimum cost, compute the controls \( \tilde{u}_k \) and \( \tilde{u}_{k+1} \) that lead from \( x_k \) to \( \tilde{x}_{k+2} \), choose \( \tilde{u}_k \) as the next rollout 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 \( \ell \)-step ahead states \( x_{k+\ell} \), etc; see Fig. 8.13. In variants of the algorithm, the \( \ell \)-step lookahead minimization may involve approximations aimed at simplifying the associated computations.
Appendix: Finite Horizon Deterministic Problems - Discrete Optimization

An important variation for problems with a long horizon 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 tree, 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. 8.15. One possibility that works well for many problems, particularly when the combined lookahead for minimization and base heuristic simulation is long, is to simply set the terminal cost approximation to zero. Alternatively, the terminal cost function approximation can be obtained by problem approximation or by using some sophisticated off-line training process that may involve an approximation architecture such as a neural network.

An important observation is that the preceding algorithms can be viewed as lookahead minimization with terminal cost approximation, so they 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 into an equivalent infinite horizon SSP problem; cf. the discussion of Section 8.2. Thus the algorithm inherits the fast convergence property of the Newton step, which we have discussed in the context of infinite horizon problems.

**Simplified Multistep Rollout - Double Rollout**

The main difficulty in applying $\ell$-step lookahead rollout stems from the rapid growth of the lookahead tree as $\ell$ increases, and the accordingly large number of applications of the base heuristic. In these circumstances,
Figure 8.14 Illustration of a form of deterministic rollout with selective depth lookahead. After \( k \) steps of the algorithm, we have a trajectory that starts at the initial state \( x_0 \) and ends at state \( x_k \). We then generate the set of all possible next states (states \( x_{k+1}^1, x_{k+1}^2, x_{k+1}^3, x_{k+1}^4 \) in the figure). We “evaluate” these states using the base heuristic, and select some of them for “expansion,” i.e., we generate their next states \( x_{k+2} \), evaluate them using the base heuristic, and continue. In the end we have a selective depth tree of next states, and the base heuristic costs from the leaves of the tree. The state \( x_{k+1} \) that corresponds to the smallest overall cost is chosen by the selective depth lookahead rollout algorithm. For problems with a large number of stages, we can also truncate the rollout trajectories and add a terminal cost function approximation as compensation for the resulting error; cf. Fig. 8.15.

we may consider simplified rollout, which is based on “pruning” the lookahead tree, so that the number of its leaves becomes manageable. By this we mean disregarding some of the states that are \( \ell \) steps or less ahead, which are judged less promising according to some criterion (for example the costs of the base heuristic after a one-step lookahead); see Fig. 8.14.

Simplified rollout aims to limit the number of times that the base heuristic is applied, which can grow exponentially as the length of lookahead is increased. In some contexts, it may also be viewed as selective depth lookahead, whereby the lookahead tree is expanded nonuniformly (lookahead is deeper from some states than for others).

An interesting idea for selective depth lookahead and tree pruning is based on applying rollout to the solution of the \( \ell \)-step lookahead minimization; after all, this is also a discrete optimization problem that can be addressed by any suboptimal method, including rollout. Thus we may use a second base heuristic...
to generate a promising trajectory through the $\ell$-step lookahead tree by using one-step lookahead rollout. This “double rollout” algorithm requires a number of heuristic applications that grows linearly rather than exponentially with $\ell$. In particular, at each stage, the number of heuristic applications of the $\ell$-step rollout and of the “double rollout” algorithm will be bounded by $n^\ell$ and by $n \cdot \ell$, respectively, where $n$ is a bound on the number of control choices at each state. Note that the two base heuristics need not be related to each other, as they are applied to different problems: one is used for the $(N-k)$-step minimization/tail subproblem that starts at state $x_k$, while the second is used for the $\ell$-step lookahead minimization that starts at state $x_k$.

**Fortified Multistep Rollout**

Among other variations of deterministic multistep rollout, let us mention a fortified version, which guards against lack of sequential improvement, and the possibility that the rollout algorithm gets sidetracked along an inferior trajectory. The fortified algorithm maintains a tentative best trajectory, from which it will not deviate until the rollout algorithm generates a less costly trajectory, similar to the one-step lookahead case.

Let us finally mention a variant of deterministic rollout that maintains multiple trajectories, extending from a given state $x_k$ to possibly multiple next states $x_{k+1}$. These states are the ones considered “most promising” based on the current results of the multistep minimization (like being “$\epsilon$-best”), but may be discarded later based on subsequent computations. Such extended forms of rollout can be combined with a fortified rollout scheme to ensure cost improvement over the base heuristic. They are restricted to deterministic problems, and tend to be problem-dependent.

### 8.4 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 non-increasing 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 starting from the initial condition \( x_0 \). Thus *to apply reliably the constrained rollout algorithm, we only only need to know a single feasible solution*, i.e., a trajectory \( T \) that starts at \( x_0 \) and satisfies the constraint \( T \in C \).

*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; simplified versions of the algorithm do not require the finiteness condition. A sequence of the form

\[
T = (x_0, u_0, x_1, u_1, \ldots, u_{N-1}, x_N),
\]

where

\[
x_{k+1} = f_k(x_k, u_k), \quad k = 0, 1, \ldots, N - 1,
\]

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

\[
\min_{T \in C} G(T),
\]

where \( G \) is some cost function.

Note that \( G \) need not have the additive form

\[
G(T) = g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k),
\]
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. (8.26).†

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 (8.28), 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

$$
g^m_N(x_N) + \sum_{k=0}^{N-1} g^m_k(x_k, u_k) \leq b^m, \quad m = 1, \ldots, M, \quad (8.29)$$

where $g^m_k$ and $b^m$ are given functions and scalars, respectively. An example where difficult trajectory constraints arise is when the control contains some discrete components, which once chosen must remain fixed for multiple time periods.

Here is another discrete optimization example involving the traveling salesman problem.

**Example 8.5 (A Constrained Form of the Traveling Salesman Problem)**

Let us consider a constrained version of the traveling salesman problem of Example 2.3.1. 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. 8.16. This constraint need not have the additive structure of Eq. (8.29). 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 or equal to 10. Note that the (unconstrained) minimum cost tour, ABDCA, does not satisfy the safety constraint.

**Transforming Constrained DP Problems to Unconstrained Problems**

Generally, a constrained DP problem can be transformed to an unconstrained DP problem, at the expense of a complicated reformulation of the state and the system equation. The idea is to redefine the state at stage $k$ to be the partial trajectory

$$
y_k = (x_0, u_0, x_1, \ldots, u_{k-1}, x_k),$$

† Actually 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 an internal process that is unknown to us) without assigning numerical values to them.
which evolves according to a redefined system equation:

\[ y_{k+1} = (y_k, u_k, f_k(x_k, u_k)) \].

The problem then becomes to find a control sequence that minimizes the terminal cost \( G(y_N) \) subject to the constraint \( y_N \in C \). This is a problem to which the standard form of DP applies:

\[ J_k^*(y_k) = \min_{u_k \in U_k(y_k)} J_{k+1}^*(y_k, u_k, f_k(x_k, u_k)), \quad k = 0, \ldots, N - 1, \]

where

\[ J_N^*(x_N) = g_N(x_N), \]

and for \( k = 0, \ldots, N - 1 \), the constraint set \( U_k(y_k) \) is the subset of controls for which it is possible to attain feasibility. Thus \( U_k(y_k) \) is the set of \( u_k \) such that there exist \( u_{k+1}, \ldots, u_{N-1} \) and corresponding \( x_{k+1}, \ldots, x_N \),
which together with \(y_k\), satisfy

\[
(y_k, u_k, x_{k+1}, u_{k+1}, \ldots, x_{N-1}, u_{N-1}, x_N) \in C.
\]

The reformulation to an unconstrained problem just described is typically impractical, because the associated computation can be overwhelming. However, it provides guidance for structuring a constrained rollout algorithm, which we describe next. Moreover, it allows the interpretation of this constrained rollout algorithm in terms of the Newton step, which is the central theme of this monograph.

**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

\[
x_{t+1} = f_t(x_t, u_t), \quad t = k, \ldots, N - 1.
\]

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

\[
y_{k+1} = (y_k, u_k, f_k(x_k, u_k)).
\]

(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 \((y_k, u_k, R(y_{k+1}))\),

which is denoted by \(T_k(y_k, u_k)\):

\[
T_k(y_k, u_k) = (y_k, u_k, R(y_{k+1})).
\]  

(8.30)

This process is illustrated in Fig. 8.17. 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 \(x_{k+1}\).
The objective is to produce a feasible final complete trajectory $\tilde{U}$ and selects from $u$ controls that is no larger than the cost of $R$.

Our rollout algorithm starts from a given initial state $\tilde{y}$, where $\tilde{y}$, $\tilde{x}$, ... , $\tilde{x}_{k-1}$, $\tilde{x}_k$.

A complete trajectory $T_k(\tilde{y}, u_k)$ is the last state component of $\tilde{y}$, of the form $N$ for which $k$ $y_k = (\tilde{y}_k, u_k, x_{k+1})$, and form the complete trajectory $T_k(\tilde{y}, u_k)$.

Then the rollout algorithm:

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

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

A complete trajectory $T_k(\tilde{y}, u_k)$ of the form (8.30) is generally feasible for only the subset $U_k(\tilde{y})$ of controls $u_k$ that maintain feasibility:

$$U_k(\tilde{y}) = \{u_k \mid T_k(\tilde{y}, u_k) \in C\}. \quad (8.31)$$

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

$$\tilde{y}_{k+1} = (\tilde{y}_k, \tilde{u}_k, f_k(\tilde{x}_k, \tilde{u}_k)), \quad k = 0, \ldots, N - 1, \quad (8.32)$$

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 $G(T_k(\tilde{y}, u_k))$ over all $u_k$ for which $T_k(\tilde{y}, u_k)$ is feasible. Thus at stage $k$, the algorithm forms the set $U_k(\tilde{y})$ [cf. Eq. (8.31)] and selects from $U_k(\tilde{y})$ a control $\tilde{u}_k$ that minimizes the cost of the complete trajectory $T_k(\tilde{y}, u_k)$:

$$\tilde{u}_k \in \arg \min_{u_k \in U_k(\tilde{y})} G(T_k(\tilde{y}, u_k)); \quad (8.33)$$

see Fig. 8.17. 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.,

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

\[
\tilde{y}_k = (\tilde{x}_0, \tilde{u}_0, \tilde{x}_1, \ldots, \tilde{u}_{k-1}, \tilde{x}_k) \quad (8.34)
\]

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

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

The algorithm then forms the set of controls

\[
U_k(\tilde{y}_k) = \{ u_k | T_k(\tilde{y}_k, u_k) \in C \}
\]

that is consistent with feasibility [cf. Eq. (8.31)], and chooses a control \( \tilde{u}_k \in U_k(\tilde{y}_k) \) according to the minimization

\[
\tilde{u}_k \in \arg \min_{u_k \in U_k(\tilde{y}_k)} G(T_k(\tilde{y}_k, u_k)), \quad (8.35)
\]

[cf. Eq. (8.33)], where

\[
T_k(\tilde{y}_k, u_k) = (\tilde{y}_k, u_k, R(\tilde{y}_k, u_k, f_k(\tilde{x}_k, u_k)));
\]

[cf. Eq. (8.30)]. Finally, the algorithm sets

\[
\tilde{x}_{k+1} = f_k(\tilde{x}_k, \tilde{u}_k), \quad \tilde{y}_{k+1} = (\tilde{y}_k, \tilde{u}_k, \tilde{x}_{k+1})
\]

[cf. Eq. (8.32)], thus obtaining the partial trajectory \( \tilde{y}_{k+1} \) to start the next stage.

Note that \( T_k(\tilde{y}_k, u_k) \) is not guaranteed to be feasible for any given \( u_k \) (i.e., it may not belong to \( C \)), but we will assume that the constraint set \( U_k(\tilde{y}_k) \) of the problem (8.33) 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 given initial state \( \tilde{y}_0 \) is feasible, i.e., \( R(\tilde{y}_0) \in C \).

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 for our earlier traveling salesman Example 8.5 is illustrated in Fig. 8.18. 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 that the minimum cost tour, ABDCA, is 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 (8.35) 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

\[
G(T_{k+1}(\tilde{y}_{k+1}, \tilde{u}_{k+1})) \leq G(T_k(\tilde{y}_k, \tilde{u}_k)), \quad k = 0, 1, \ldots, N - 1,
\]

while at the first step of the algorithm we have

\[
G(T_0(\tilde{y}_0, \tilde{u}_0)) \leq G(R(\tilde{y}_0)).
\]

It will then follow that the cost improvement property

\[
G(\tilde{y}_N) \leq G(R(\tilde{y}_0))
\]

holds.

**Definition 8.3:** 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 \( y_{k+1} = (y_k, u_k, x_{k+1}) \).
Appendix: Finite Horizon Deterministic Problems - Discrete Optimization

Figure 8.18 The constrained traveling salesman problem; cf. Example 8.5, 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.
As we have noted in the context of unconstrained rollout, greedy heuristics tend to be sequentially consistent. Also any policy \[ \mu_k(y_k), k = 0, 1, \ldots, N - 1 \] for the DP problem of minimizing the terminal cost \( G(y_N) \) subject to the system equation
\[
y_{k+1} = \left( y_k, u_k, f_k(x_k, u_k) \right)
\]
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 8.5. From Fig. 8.18, 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
\[
R(y_k) = (x_k, u_{k+1}, \ldots, u_{N-1}, x_N),
\]
we have
\[
y_k \cup R(y_k) = (x_0, u_0, \ldots, u_{k-1}, x_k, u_{k+1}, \ldots, u_{N-1}, x_N).
\]

**Definition 8.4:** 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 \( U_k(y_k) \) is nonempty, and we have
\[
G(y_k \cup R(y_k)) \geq \min_{u_k \in U_k(y_k)} G(T_k(y_k, u_k)). \tag{8.36}
\]

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 \( U_k(y_k) \) is nonempty. The reason is that starting from \( (y_k, u_k, f_k(x_k, u_k)) \) 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 \( U_k(y_k) \) 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
\[
\{ T_k(y_k, u_k) \mid u_k \in U_k(y_k) \}.
\]
Proposition 8.3: (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

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

where

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

cf. Eq. (8.30). 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

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

[cf. Eq. (8.36)], 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

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

[cf. Eq. (8.36)], 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 $G(T_k(\tilde{y}_k, \tilde{u}_k)) \geq G(T_{k+1}(\tilde{y}_{k+1}, \tilde{u}_{k+1}))$ for all $k$. Q.E.D.

Proposition 8.3 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 (8.36) 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 2.3.2).

In the case of the traveling salesman Example 8.5, it can be verified that the base heuristic specified in Fig. 8.18 is sequentially improving. However, if the travel cost \( D \rightarrow A \) 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. (8.31)] 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

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

(8.37)

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

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

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:

\[
\hat{U}_k(\tilde{y}_k) = \left\{ u_k \in U_k(\tilde{y}_k) \mid G(T_k(\tilde{y}_k, u_k)) \leq G(\hat{T}_k) \right\}.
\]

There are two cases to consider at state \( k \):
Appendix: Finite Horizon Deterministic Problems - Discrete Optimization

(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

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

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

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

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. (8.37)], 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. 8.3, 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 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. (8.34), 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.

**Constrained Multiagent Rollout**

Let us consider a special structure of the control space, where the control \( u_k \) consists of \( m \) components, \( u_k = (u^1_k, \ldots, u^m_k) \), each belonging to a corresponding set \( U^\ell_k(x_k) \), \( \ell = 1, \ldots, m \). Thus the control space at stage \( k \) is the Cartesian product

\[
U_k(x_k) = U^1_k(x_k) \times \cdots \times U^m_k(x_k).
\]

We refer to this as the *multiagent case*, motivated by the special case where each component \( u^\ell_k, \ell = 1, \ldots, m \), is chosen by a separate agent \( \ell \) at stage \( k \).

Similar to the stochastic unconstrained case of Section 3.2 (Fig. 3.17), 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 \( u^1_k, u^2_k, \ldots, u^m_k \), and between \( x_k \) and the next state \( x_{k+1} = f_k(x_k, u_k) \), we introduce artificial intermediate “states”

\[
(x_k, u^1_k), (x_k, u^1_k, u^2_k), \ldots, (x_k, u^1_k, \ldots, u^{m-1}_k),
\]

and corresponding transitions. The choice of the last control component \( u^m_k \) at “state” \( (x_k, u^1_k, \ldots, u^{m-1}_k) \) 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 \( u^1_k, \ldots, u^m_k \), 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 \( U^\ell_k(x_k), \ell = 1, \ldots, m, \) the computation required for the \( m \) single control component minimizations is of order \( O(nm) \) per stage. By contrast the standard rollout minimization (8.35) involves the computation and comparison of as many as \( n^m \) terms \( G(T_k(\hat{y}_k, u_k)) \) per stage.