Rollout, Approximate Policy Iteration, and Distributed Reinforcement Learning

by

Dimitri P. Bertsekas

Chapter 2
Rollout and Policy Improvement

These notes represent “work in progress,” and will be periodically updated. They more than likely contain errors (hopefully not serious ones). Furthermore, the references to the literature are incomplete. Your comments and suggestions to the author at dbertsek@asu.edu are welcome.

February 23, 2020
# Rollout and Policy Improvement

<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1. Approximation in Value and Policy Space</td>
</tr>
<tr>
<td>2.1.1. Approximation in Value Space - One-Step and Multistep Lookahead</td>
</tr>
<tr>
<td>2.1.2. Approximation in Policy Space</td>
</tr>
<tr>
<td>2.1.3. Combined Approximation in Value and Policy</td>
</tr>
<tr>
<td>2.2. General Issues of Approximation in Value Space</td>
</tr>
<tr>
<td>2.2.1. Model-Based and Model-Free Implementations</td>
</tr>
<tr>
<td>2.2.2. Off-Line and On-Line Implementations</td>
</tr>
<tr>
<td>2.2.3. Methods for Cost-to-Go Approximation</td>
</tr>
<tr>
<td>2.2.4. Methods for Simplification of the Lookahead Minimization</td>
</tr>
<tr>
<td>2.2.5. Simplification of the Lookahead Minimization by Q-Factor Approximation</td>
</tr>
<tr>
<td>2.3. Rollout and the Policy Improvement Principle</td>
</tr>
<tr>
<td>2.3.1. On-Line Rollout for Deterministic Discrete Optimization</td>
</tr>
<tr>
<td>2.3.2. The Fortified Rollout Algorithm</td>
</tr>
<tr>
<td>2.3.3. Truncated Rollout Algorithms</td>
</tr>
<tr>
<td>2.4. Stochastic Rollout and Monte Carlo Tree Search</td>
</tr>
<tr>
<td>2.4.1. Simulation-Based Implementation of the Rollout Algorithm</td>
</tr>
<tr>
<td>Section</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>2.4.2. Variance Reduction in Rollout - Comparing Advantages</td>
</tr>
<tr>
<td>2.4.3. Monte Carlo Tree Search</td>
</tr>
<tr>
<td>2.4.4. Randomized Policy Improvement by Monte Carlo Tree Search</td>
</tr>
<tr>
<td>2.4.5. Rollout Parallelization</td>
</tr>
<tr>
<td>2.5. On-Line Rollout for Deterministic Infinite-Spaces Problems -</td>
</tr>
<tr>
<td>2.5.1. Model Predictive Control</td>
</tr>
<tr>
<td>2.6. Multiagent Rollout</td>
</tr>
<tr>
<td>2.6.1. Multiagent Parallelization</td>
</tr>
<tr>
<td>2.7. Constrained Rollout for Deterministic Optimization</td>
</tr>
<tr>
<td>2.7.1. Constrained Multiagent Rollout</td>
</tr>
<tr>
<td>2.8. Discrete and Combinatorial Optimization Applications</td>
</tr>
<tr>
<td>2.8.1. Multidimensional Assignment</td>
</tr>
<tr>
<td>2.9. Rollout for Minimax Control</td>
</tr>
<tr>
<td>2.10. Notes and Sources</td>
</tr>
</tbody>
</table>
As we noted in Chapter 1, the exact solution of optimal control problems by DP is often impossible. To a great extent, the reason lies in what Bellman has called the “curse of dimensionality.” This refers to a rapid increase of the required computation and memory storage as the size of the problem increases. Moreover, there are many circumstances where the structure of the given problem is known well in advance, but some of the problem data, such as various system parameters, may be unknown until shortly before control is needed, thus seriously constraining the amount of time available for the DP computation. These difficulties motivate suboptimal control schemes that strike a reasonable balance between convenient implementation and adequate performance.

In this chapter, we first summarize in Section 2.1 the principal approximation approaches for the finite horizon deterministic and stochastic DP problems of Chapter 1. We then focus on the key ideas of one-step and multistep lookahead, and policy improvement by rollout, as well as various possibilities for their implementation. Many of the finite horizon methods will be adapted to infinite horizon DP later, together with some additional methods that are specific to the infinite horizon context.

2.1 APPROXIMATION IN VALUE AND POLICY SPACE

There are two general types of approximation in DP-based suboptimal control. The first is approximation in value space, where we aim to approximate the optimal cost function or the cost function of a given policy, often using some process based on data collection. The second is approximation in policy space, where we select a policy from a suitable class of policies based on some criterion; the selection process often uses data, optimization, and neural network approximations.† In some settings the value space and policy space approximation approaches may be combined. In this section we provide a broad overview of the main ideas.

2.1.1 Approximation in Value Space - One-Step and Multistep Lookahead

Let us consider the finite horizon stochastic DP problem of Section 1.2. In an important form of approximation in value space, we approximate the

† The principal role of neural networks within the context of this monograph is to provide the means for approximating various target functions from input-output data. This includes cost functions and Q-factors of given policies, optimal cost-to-go functions and Q-factors, as well as given policies from data. In this chapter, we will discuss the general contexts for approximation. The training methods for constructing the approximations themselves from data are mostly based on optimization and regression, and will be discussed in Chapter 3.
Figure 2.1.1 Schematic illustration of one-step lookahead and the associated three types of approximations. At each state $x_k$, it uses the control obtained from the minimization

$$\hat{\mu}_k(x_k) \in \arg\min_{u_k \in U_k(x_k)} E \left\{ g_k(x_k, u_k, w_k) + \hat{J}_{k+1}(x_{k+1}) \right\},$$

or equivalently in terms of Q-factors,

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

This defines the suboptimal policy $\hat{\pi} = \{\hat{\mu}_0, \ldots, \hat{\mu}_{N-1}\}$, referred to as the one-step lookahead policy based on $\hat{J}_{k+1}$, $k = 0, \ldots, N - 1$. There are three potential areas of approximation here, which can be considered independently of each other: cost-to-go approximation, expected value approximation, and minimization approximation.

Optimal cost-to-go functions $J^*_k$ with some other functions $\hat{J}_k$. We then replace $J^*_k$ in the DP equation with $\hat{J}_{k+1}$. In particular, at state $x_k$, we use the control obtained from the minimization

$$\hat{\mu}_k(x_k) \in \arg\min_{u_k \in U_k(x_k)} E \left\{ g_k(x_k, u_k, w_k) + \hat{J}_{k+1}(x_{k+1}) \right\}.$$  \hspace{1cm} (2.1)

This process defines a suboptimal policy $\hat{\pi} = \{\hat{\mu}_0, \ldots, \hat{\mu}_{N-1}\}$ and is called one-step lookahead; see Fig. 2.1.1. There are several possibilities for selecting or computing the functions $\hat{J}_{k+1}$, some of which are discussed in what follows. In some schemes the expected value and minimization operations may also be carried out approximately; see Fig. 2.1.1.

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

$$\hat{Q}_k(x_k, u_k) = E \left\{ g_k(x_k, u_k, w_k) + \hat{J}_{k+1}(x_{k+1}) \right\};$$

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

$$\hat{\mu}_k(x_k) \in \arg\min_{u_k \in U_k(x_k)} \hat{Q}_k(x_k, u_k),$$ \hspace{1cm} (2.2)
Sec. 2.1 Approximation in Value and Policy Space

First $\ell$ Steps “Future”

At $x_k$, $u_k, \mu_{k+1}, \ldots, \mu_{k+\ell-1}$, $E\left\{ g_k(x_k, u_k, w_k) + \sum_{m=k+1}^{k+\ell-1} g_m(x_m, \mu_m(x_m), w_m) + \tilde{J}_{k+\ell}(x_{k+\ell}) \right\}$

Figure 2.1.2 Schematic illustration of $\ell$-step lookahead with approximation in value space. At each state $x_k$ solves an $\ell$-stage DP problem to obtain a sequence $\{\tilde{u}_k, \tilde{\mu}_{k+1}, \ldots, \tilde{\mu}_{k+\ell-1}\}$, and then uses the first control $\tilde{u}_k$ in this sequence. It involves the same three approximations as one-step lookahead: cost-to-go approximation, expected value approximation, and minimization approximation. The minimization of the expected value is more time consuming, but the cost-to-go approximation after $\ell$ need not be chosen as accurately/carefully as one-step lookahead.

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

Approximation in Value Space - Multistep Lookahead

An important extension of one-step lookahead is multistep lookahead (also referred to as $\ell$-step lookahead), whereby at state $x_k$ we minimize the cost of the first $\ell > 1$ stages with the future costs approximated by a function $\tilde{J}_{k+\ell}$. For example, in two-step lookahead the function $\tilde{J}_{k+1}$ is given by

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

where $\tilde{J}_{k+2}$ is some approximation of the optimal cost-to-go function $J^*_k$. More generally, at state $x_k$ we solve the $\ell$-stage problem

$$\min_{u_k, \mu_{k+1}, \ldots, \mu_{k+\ell-1}} E\left\{ g_k(x_k, u_k, w_k) + \sum_{i=k+1}^{k+\ell-1} g_i(x_i, \mu_i(x_i), w_i) + \tilde{J}_{k+\ell}(x_{k+\ell}) \right\}$$

to obtain a corresponding optimal sequence $\{\tilde{u}_k, \tilde{\mu}_{k+1}, \ldots, \tilde{\mu}_{k+\ell-1}\}$. We then use the first control $\tilde{u}_k$ in this sequence, we obtain the next state

$$x_{k+1} = f_k(x_k, \tilde{u}_k, w_k),$$

and we repeat the process with $x_k$ replaced by $x_{k+1}$; see Fig. 2.1.2.
Actually, one may view \( \ell \)-step lookahead as the special case of one-step lookahead where the lookahead function is the optimal cost function of an \((\ell - 1)\)-stage DP problem with a terminal cost \( \tilde{J}_{k+\ell}(x_{k+\ell}) \) on the state \( x_{k+\ell} \) obtained after \( \ell - 1 \) stages. However, it is often important to view \( \ell \)-step lookahead separately, in order to address special implementation issues that do not arise in the context of one-step lookahead.

The motivation for \( \ell \)-step lookahead is that by increasing the value of \( \ell \), one may require a less accurate approximation \( \tilde{J}_{k+\ell} \) to obtain good performance. Otherwise expressed, for the same quality of cost function approximation, better performance may be obtained as \( \ell \) becomes larger. This makes intuitive sense, since in this case, the cost of more stages is treated exactly, i.e., with optimization. Moreover, after many stages, due to randomness, discounting, or other factors, the cost of the remaining stages may become negligible or may not depend much on the choice of the control \( u_k \) at time \( k \). Indeed this expectation is typically realized in practice, although one can construct artificial examples when this is not so (see [Ber19a], Section 2.2.1).

Note that in a deterministic setting, the lookahead problems are also deterministic, and may be addressed by efficient shortest path methods. This makes deterministic problems particularly good candidates for the use of many-step lookahead. Generally, the implementation of \( \ell \)-step lookahead can be prohibitively time-consuming for stochastic problems, because it requires at each step the solution of a stochastic DP problem with an \( \ell \)-step horizon. As a practical guideline, one should at least try to use the largest value of \( \ell \) for which the computational overhead for solving the \( \ell \)-step lookahead minimization problem on-line is acceptable.

In our discussion of approximation in value space of the present section, we will focus primarily on one-step lookahead. Usually, there are straightforward extensions of the main ideas to the multistep context.

### Approximation in Value Space - Infinite Horizon

In this chapter we will focus exclusively on finite horizon problems. However, we should note that approximation in value space, with both one-step and multistep lookahead is conceptually very similar in infinite horizon problems. There are again three potential areas of approximation: cost-to-go approximation, expected value approximation, and minimization approximation; see Fig. 2.1.3.

A major advantage of the infinite horizon context is that only one approximate cost function \( \tilde{J} \) is needed, rather than the \( N \) functions \( \tilde{J}_1, \ldots, \tilde{J}_N \) of the \( N \)-step horizon case. Moreover, we will see that for infinite horizon problems, there are additional important algorithms that are amenable to approximation in value space. Approximate policy iteration, Q-learning, temporal difference methods, and their variants are some of these. For this reason, in the infinite horizon case, there is a richer set of algorithmic op-
2.1 Approximation in Value and Policy Space

Sec. 2.1 Approximation in Value and Policy Space

Figure 2.1.3 Schematic illustration of approximation in value space with one-step lookahead for infinite horizon problems, and the associated three approximations.

tions for approximation in value space, despite the fact that the associated mathematical theory is more complex.

It is also important to note that most of the algorithmic ideas of approximation in value space can be adapted from the finite horizon to the infinite horizon context. This is convenient for us, as it will allow us to develop the approximation methodology within the conceptually simpler finite horizon context, and postpone to Chapter 4 the discussion of infinite horizon algorithms.

2.1.2 Approximation in Policy Space

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

\[
\hat{\mu}_k(x_k, r_k), \quad k = 0, \ldots, N - 1,
\]

where \( r_k \) is a parameter, and then estimate the parameters \( r_k \) using some type of optimization.

Neural networks, described in Chapter 3, are often used to generate the parametric class of policies, in which case \( r_k \) is the vector of weights/parameters of the neural network. In Chapter 3, we will also discuss methods for obtaining the training data required for obtaining the parameters \( r_k \), and we will consider several other classes of approximation architectures. An important advantage of approximation in policy space is that once the parametrized policy is obtained, the computation of controls

\[
u_k = \hat{\mu}_k(x_k, r_k), \quad k = 0, \ldots, N - 1,
\]
during on-line operation of the system is often much easier compared with the lookahead minimization (2.1).
Uncertainty System Environment Cost Control Current State

Controller

\[ u_k = \tilde{\mu}_k(x_k, r_k) \]

System

State \( x_k \)

Training Data

**Figure 2.1.1** Schematic illustration of parametric approximation in policy space.

A policy

\[ \tilde{\mu}_k(x_k, r_k), \quad k = 0, 1, \ldots, N - 1, \]

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

In Section 3.4, we will discuss in some detail the approximation of policies for the case where the number of controls available at \( x_k \) is finite. In this case, we will see that \( \tilde{\mu}_k(x_k, r_k) \) is computed as a randomized policy, i.e., a set of probabilities of applying each of the available controls at \( x_k \) (a parametrized policy may be computed in randomized form for reasons of algorithmic/training convenience; in practice it is typically implemented by applying at state \( x_k \) the control of maximum probability). The methods to obtain the parameter \( r_k \) are similar to classification methods used in pattern recognition.

There are also alternative optimization-based approaches, where the main idea is that once we use a vector \((r_0, r_1, \ldots, r_{N-1})\) to parametrize the policies \( \pi \), the expected cost \( J_\pi(x_0) \) is also parametrized, and can be viewed as a function of \((r_0, r_1, \ldots, r_{N-1})\). We can then optimize this cost by using a gradient-like or random search method. This is a widely used approach for optimization in policy space, which, however, we will not discuss much in this book (for details and many references to the literature, see the RL book [Ber19a], Section 5.7). Actually, this type of approach is used most often in an infinite horizon context, where the policies of interest are stationary, so a single function \( \mu \) needs to be approximated rather than the \( N \) functions \( \mu_0, \ldots, \mu_{N-1} \).

In this chapter we discuss primarily approximation in value space, although some of the ideas are also relevant to approximation in policy space. We focus on finite horizon problems, postponing the discussion of infinite horizon problems for Chapter 4 and later. Many of the methods of the present chapter and Chapter 3 also apply with small modifications to infinite horizon problems.
2.1.3 Combined Approximation in Value and Policy Space

In this section, we discuss various ways to combine approximation in value and in policy space. In particular, we first describe how approximation in policy space can be built starting from approximation in value space. We then discuss a reverse process, namely how we can start from some policy, such as a nearly optimal policy, and construct an approximation in value or Q-factor space, which in turn can be used to construct a new policy through one-step or multistep lookahead. This is the rollout approach, which we will discuss at length in this chapter. Finally, we show how to combine the two types of approximation in a perpetual cycle of repeated approximations in value and policy space. This involves the use of approximation architectures, such as neural networks, which will be discussed in Chapter 3.

Approximation in Policy Space on Top of Approximation in Value Space

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

\[
\min_{r_k} \sum_{s=1}^{q} \| u^s_k - \hat{\mu}_k(x^s_k, r_k) \|^2
\]

(2.3)

(possibly with added regularization).† In particular, we may determine \(u^s_k\) using a human or a software “expert” that can choose “near-optimal”

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

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

\[
u_k^s \in \arg \min_{u \in U_k(x_k^s)} E\left\{ g_k(x_k^s, u, w_k) + \tilde{J}_{k+1}(f_k(x_k^s, u, w_k)) \right\},
\]

(2.4)

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

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

(2.5)

In this case, we collect the sample state-control pairs \((x_k^s, u_k^s)\), \(s = 1, \ldots, q\), by using approximation in value space through Eq. (2.4) or Q-factor approximation through Eq. (2.5), and then apply approximation in policy space through Eq. (2.3) (i.e., approximation in policy space is built on top of approximation in value space).

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

An important approach for approximation in value space is to use one-step or multistep lookahead with cost function approximation \( \tilde{J}_k(x_k) \) equal to the tail problem cost \( J_{k,\pi}(x_k) \) starting from \( x_k \) and using some known policy \( \pi = \{\mu_0, \ldots, \mu_{N-1}\} \), i.e., in the case of one-step lookahead, use the control

\[
\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} E\left\{ g_k(x_k, u_k, w_k) + J_{k+1,\pi}(f_k(x_k, u_k, w_k)) \right\}.
\]

Equivalently we can use one-step lookahead with the Q-factors \( Q_{k,\pi}(x_k, u_k) \) of the policy, i.e., use the control

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

This has the advantage of simplifying the one-step lookahead minimization. In practice, for computational expediency, an approximation to \( J_{k,\pi}(x_k) \) or \( Q_{k,\pi}(x_k, u_k) \) is frequently used instead, as we will discuss shortly.

Using the cost function or the Q-factors of a policy as a basis for approximation in value space constitutes the *rollout algorithm*, which is the principal subject of this chapter. Since the values \( J_{k,\pi}(x_k) \), \( k = 1, \ldots, N \), are typically not available analytically, it is necessary to compute them as
Figure 2.1.5 Schematic illustration of approximation in value space using a policy \( \pi = \{\mu_0, \ldots, \mu_{N-1}\} \). We use one-step or multistep lookahead where \( J_{k+1}(x_{k+1}) \) is equal to the tail problem cost \( J_{k+1,\pi}(x_{k+1}) \) starting from \( x_{k+1} \) and using policy \( \pi \). To economize in computation cost, we may use truncated rollout, whereby we perform the simulation with \( \pi \) for a limited number of stages starting from each possible next state \( x_{k+1} \), and either neglect the costs of the remaining stages or add some heuristic cost approximation at the end to compensate for these costs.

needed by some form of simulation. In particular, for a deterministic finite horizon problem, we may compute \( J_{k,\pi}(x_k) \) by accumulating the stage costs along the (unique) trajectory that starts at \( x_k \) and uses \( \pi \) to the end of the horizon. For a stochastic problem it is necessary to obtain \( J_{k,\pi}(x_k) \) by Monte Carlo simulation, i.e., generate a number of random trajectories starting from \( x_k \) and using \( \pi \) up to the end of the horizon, and then average the corresponding random trajectory costs.

Thus, starting with a policy \( \pi \), which we will call the base policy, we may use one-step or multistep lookahead with cost approximations \( J_{k,\pi} \), and obtain a new policy \( \tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\} \), which we call the rollout policy. One of the fundamental facts in DP is that the rollout policy has a policy improvement property: \( \tilde{\pi} \) has no worse cost than \( \pi \), i.e., for all \( x_k \) and \( k \),

\[
J_{k,\tilde{\pi}}(x_k) \leq J_{k,\pi}(x_k).
\] (2.6)

A computationally expedient alternative to the rollout process just described is to perform the simulation with \( \pi \) for a limited number of stages, and either neglect the costs of the remaining stages or use some heuristic cost approximation \( \tilde{J}_k \) at the end to compensate for these costs;
Figure 2.1.6 Schematic illustration of sequential approximation in value and policy space (or perpetual rollout). It produces a sequence of policies and cost function approximations. Each generated policy is viewed as the rollout policy with the preceding policy viewed as the base policy. The rollout policy is then approximated in policy space, and viewed as the base policy for the next iteration. The approximation in value space may involve either costs or Q-factors of the base policy.

see Fig. 2.1.5. We will call this process truncated rollout. One possibility is to collect many pairs of state and base policy costs \((x_k, \mu_k(x_k))\) from which to obtain cost function approximations \(J_k\), for each of the stages \(k = 1, \ldots, N\). The functions \(J_k\) approximate the policy cost functions \(J_{k,\pi}\), thus yielding an approximate rollout policy \(\tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}\) through one-step or multistep lookahead. This policy satisfies the cost improvement property (2.6) in an approximate sense (within some error bound, which is small if \(\tilde{J}_k\) is close to \(J_{k,\pi}\); see Chapter 4 for a more precise statement).

Perpetual Rollout and Approximate Policy Iteration

The truncated rollout process starts with a base policy and a terminal cost approximation, possibly obtained by approximation in value space (cf. Fig. 2.1.5), and can generate state-control samples of the rollout policy. Once this is done, the rollout policy may be implemented by approximation in policy space built on top of approximation in value space, as discussed earlier in this section. Thus the rollout process can be repeated in perpetuity, so we can obtain a sequence of policies (through approximation in policy space) and corresponding sequence of cost approximations (through approximation in value space); see Fig. 2.1.6. When neural networks are used, the approximations in value and policy space are commonly referred to as the value network and the policy network, respectively. For example, AlphaGo and AlphaZero use both value and policy networks for approximation in value space and policy space. Note that the value and policy networks must be constructed off-line (before the control process begins), since their training involves a lot of data collection and computation.

The process just described also applies and indeed becomes simpler for infinite horizon problems. Within this context, it is called approximate policy iteration, and will be discussed later in Chapter 4. An important
type of such a method is known as optimistic approximate policy iteration, and involves algorithms such as Q-learning and policy evaluation by temporal differences. Together with their variations, which depend on the details of the data collection, and the amount of data used for the value and policy space approximations, these algorithms underly a very large part of the RL methodology.

2.2 GENERAL ISSUES OF APPROXIMATION IN VALUE SPACE

In this section we discuss some of the generic implementation aspects of approximation in value space. In particular, we address how to overcome the lack of a mathematical model through the use of a computer simulator, we discuss issues relating to the mixture of on-line and off-line computation that is appropriate for given types of problems, and also consider methods to simplify the one-step lookahead minimization.

2.2.1 Model-Based and Model-Free Implementations

Generally, a finite horizon DP problem is defined by the state, control, and disturbance spaces, the functions $f_k$ and $g_k$, the control constraint sets $U_k(x_k)$, and the probability distributions of the disturbances. We refer to these as the mathematical model of the problem. Of course there is only one mathematical model for a given problem, but there may be several different implementations of a given method for solving the problem exactly or approximately. Some of these implementations may rely exclusively on analytical calculations using the mathematical model, but others may additionally or exclusively rely on Monte Carlo simulation.

In this book, we will adhere to an unambiguous technical definition of the term “model-free” (also used in the RL book [Ber19a]). In particular, we will refer to a method as being model-free if the calculation of all the expected values in Eq. (2.1), and other related expressions, is done with Monte Carlo simulation, and the conditional probability distribution of $w_k$, given $(x_k, u_k)$, is either not analytically available, or is not used for reasons of computational expediency.†

Note that for deterministic problems there is no expected value to compute, so methods for these problems come under the model-based category, even if values of the functions $g_k$ and $f_k$ become available through complicated computer calculations. Still however, Monte Carlo simulation may enter the solution process of a deterministic problem for a variety of reasons. For example the games of chess and Go are perfectly deterministic, but the AlphaGo and AlphaZero programs (Silver et al. [SHM16], [SHS17])

† An example is when Monte Carlo integration is used to estimate integrals that are analytically available but are hard to compute.
use randomized policies and rely heavily on Monte Carlo tree search techniques, which use sampling and will be discussed in Section 2.4.3. Our view of a model-free method aims to avoid ambiguities in cases where the mathematical model is available to use for closed form calculations, but Monte Carlo simulation is used anyway for reasons of convenience.

2.2.2 Off-Line and On-Line Implementations

In value space approximation, an important consideration is whether the functions $\tilde{J}_{k+1}$ and the corresponding suboptimal policy $\{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$ of Eq. (2.1) are computed off-line (i.e., before the control process begins, and for all $x_k$ and $k$), or on-line (i.e., after the control process begins, when needed, and for just the states $x_k$ to be encountered).

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

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

(ii) On-line methods, where most of the computation is performed just after the current state $x_k$ becomes known, the values $\tilde{J}_{k+1}(x_{k+1})$ are computed only at the relevant next states $x_{k+1}$, and are used to compute the control to be applied via Eq. (2.1). In contrast with the off-line approximation methods, these methods are well-suited for on-line replanning, whereby the problem data may change over time. Depending on the problem, such changes may include unanticipated demands for services, emergency situations, forecasts relating to future values of the disturbances, etc. A similar situation where on-line methods may have an advantage is when the initial state and other problem data may become known just before the control process begins.

Examples of typically off-line schemes are the neural network and other parametric approximations that we will discuss in Chapter 3. Examples of typically on-line schemes are rollout and model predictive control, which we will discuss in this chapter. Of course there are also hybrid methods, where significant computation is done off-line to expedite the on-line computation
2.2 General Issues of Approximation in Value Space

Figure 2.2.1 Schematic illustration of various approaches for approximation in value space with one-step lookahead.

of needed values of $\tilde{J}_{k+1}$. An example is the truncated rollout schemes to be discussed in Section 2.3.2.

2.2.3 Methods for Cost-to-Go Approximation

There are two major issues that arise in the implementation of one-step lookahead; each of the two can be considered separately from the other:

(a) Obtaining $\tilde{J}_k$, i.e., the method to compute the lookahead functions $\tilde{J}_k$ that are involved in the lookahead minimization (2.1). There are quite a few approaches here (see Fig. 2.2.1). Some of them are discussed in this chapter, and more will be discussed in subsequent chapters.

(b) Control selection, i.e., the method to perform the minimization (2.1) and implement the suboptimal policy $\hat{\mu}_k$. Again there are several exact and approximate methods for control selection, some of which will be discussed in this chapter (see Fig. 2.2.1). In particular, we may:

(1) Approximate the expected value appearing in the minimized expression.

(2) Perform the optimization approximately.

In this section we will provide a high level discussion of these issues, focusing for simplicity on just the case of one-step lookahead.

Regarding the computation of $\tilde{J}_k$, we may distinguish between three types of methods:

(a) Problem approximation: Here the functions $\tilde{J}_k$ in Eq. (2.1) are obtained as the optimal or nearly optimal cost functions of a simplified optimization problem, which is more convenient for computation.
Simplifications may include exploiting decomposable structure, ignoring various types of uncertainties, and reducing the size of the state space. We discuss several types of problem approximation approaches in the RL book [Ber19a]. A major type of problem approximation method is aggregation, which is described and analyzed in the books [Ber12], and [Ber19a], and the surveys [Ber18b], [Ber18c]. In this book, our use and discussion of problem approximation will be peripheral to our development, even though it can be combined with the approximation in value space methods that are our main focus.

(b) On-line approximate optimization, such as rollout algorithms and model predictive control, which are discussed in the present chapter. These methods often involve the use of a suboptimal policy or heuristic, which is applied on-line when needed to approximate the true optimal cost-to-go values. The suboptimal policy may be obtained by any other method, e.g., one based on heuristic reasoning, or on a more principled approach.

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

Additional variations of the above methods are obtained when used in combination with approximate minimization over $u_k$ in Eq. (2.1), and also when the expected value over $w_k$ is computed approximately via some type of approximation (cf. Section 2.2.2) or adaptive simulation and Monte Carlo tree search (cf. Section 2.4.2).

In what follows in this section we discuss a variety of ideas relating to approximations in the one-step and multistep lookahead minimization.

### 2.2.4 Methods for Simplification of the Lookahead Minimization

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

$$E\left\{ g_k(x_k, u_k, w_k) + \tilde{J}_{k+1} \left( f_k(x_k, u_k, w_k) \right) \right\}, \quad (2.7)$$

once the cost-to-go approximating functions $\tilde{J}_{k+1}$ or approximate Q-factors $\tilde{Q}_k$ have been selected. In this section, we will assume that we have a mathematical model, i.e., that the functions $g_k$ and $f_k$ are available in essentially closed form, and that the conditional probability distribution of $w_k$, given $(x_k, u_k)$, is also available.
Important issues here are the computation of the expected value (if the problem is stochastic) and the minimization over \( u_k \in U_k(x_k) \) in Eq. (2.7). Both of these operations may involve substantial work, which is of particular concern when the minimization is to be performed on-line.

One possibility is to eliminate the expected value from the expression (2.7) using (assumed) certainty equivalence. Here we choose a typical value \( \tilde{w}_k \) of \( w_k \), and use the control \( \tilde{\mu}_k(x_k) \) that solves the deterministic problem

\[
\min_{u_k \in U_k(x_k)} \left[ g_k(x_k, u_k, \tilde{w}_k) + \tilde{J}_{k+1} \left( f_k(x_k, u_k, \tilde{w}_k) \right) \right].
\]

(2.8)

The approach of turning a stochastic problem into a deterministic one by replacing uncertain quantities with single typical values highlights the possibility that \( \tilde{J}_{k+1} \) may itself be obtained by using deterministic methods. This approach and its variations are discussed in detail in the RL book [Ber19a] (Section 2.3). For example, an extension of certainty equivalence is a simplification of the probability distribution of \( w_k \) (such as concentrating its probability mass to a few representative values), so that the computation of the expected value in the one-step lookahead expression (2.7) is simplified. These possibilities also admit straightforward extension to the case of multistep lookahead.

Let us now consider the issue of algorithmic minimization over \( U_k(x_k) \) in Eqs. (2.7) and (2.8). If \( U_k(x_k) \) is a finite set, the minimization can be done by brute force, through exhaustive computation and comparison of the relevant cost expressions. This of course can be very time consuming, particularly for multistep lookahead, but parallel computation can be used with great effect for this purpose [as well as for the calculation of the expected value in the expression (2.7)]. In particular, the minimization over the different controls \( u_k \in U_k(x_k) \) are uncoupled and may be done in parallel. Note, however, that when Monte Carlo simulation is used to evaluate the expectation over \( w_k \), it may be important to use the same samples of \( w_k \) for all controls for purposes of variance reduction. This issue is discussed in Section 2.4.2.

For some problems, integer programming techniques may also be used in place of brute force minimization over \( U_k(x_k) \). Moreover, for deterministic problems with multistep lookahead, sophisticated exact or approximate shortest path methods may be considered; several methods of this type are available, such as label correcting methods, \( A^* \) methods, and their variants (see Section 2.9, as well as the author’s textbooks [Ber98] and [Ber17] for detailed accounts, which are consistent with the context of this chapter).

When the control constraint set is infinite, it may be replaced by a finite set through discretization. However, a more efficient alternative may be to use continuous space nonlinear programming techniques. This possibility can be attractive for deterministic problems, which lend themselves better to continuous space optimization; an example is the model predictive control context (see Section 2.5).
For stochastic problems with continuous control spaces and either one-step or multistep lookahead, the methodology of stochastic programming may be useful. This methodology bears a close connection with linear and nonlinear programming methods. We refer to the textbook [Ber17] for a discussion of its application to the approximate DP context, and references to the literature.

One more possibility to simplify or bypass the one-step lookahead minimization (2.7) is to construct a Q-factor approximation, which is based on approximate Q-factor samples. This is also suitable for model-free policy implementation and is described next.

2.2.5 Simplification of the Lookahead Minimization by Q-Factor Approximation

We now discuss how to simplify the minimization over $U_k(x_k)$ of the approximate Q-factors

$$E\left\{g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}(f_k(x_k, u_k, w_k))\right\},$$

in the computation of the one-step lookahead control. We assume that:

(a) Either the system equation $x_{k+1} = f_k(x_k, u_k, w_k)$, is known or there is a computer program/simulator that for any given state $x_k$ and control $u_k \in U_k(x_k)$, simulates sample probabilistic transitions to a successor state $x_{k+1}$, and generates the corresponding transition costs.

(b) The cost function approximation $\tilde{J}_{k+1}$ is available in some way. In particular, $\tilde{J}_{k+1}$ may be obtained by solving a simpler problem for which a model is available, or it may be obtained by some other method, based for example on simulation with a given policy and/or neural network training.

Given a state $x_k$, we may compute the Q-factors (2.9) for all the pairs $(x_k, u_k), u_k \in U_k(x_k)$, and then select the minimizing control. However, in many cases this can be very time-consuming for on-line computation purposes. To deal with this difficulty, a common approach is to introduce a parametric family of Q-factor functions,

$$\tilde{Q}_k(x_k, u_k, r_k),$$

where $r_k$ is the parameter vector and use a least squares fit/regression to approximate the expected value that is minimized in Eq. (2.7). The steps are as follows:
Summary of Q-Factor Approximation Based on Cost Function Approximation

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

(a) Use the system equation or a simulator to collect a large number of “representative” quadruplets $(x_s^k, u_s^k, x_{k+1}^s, g_s^k)$, and corresponding Q-factors

$$\beta_s^k = g_s^k + \hat{J}_{k+1}(x_{k+1}^s), \quad s = 1, \ldots, q.$$ \hspace{1cm} (2.10)

Here $x_{k+1}^s$ is the next state

$$x_{k+1}^s = f_k(x_s^k, u_s^k, w_s^k)$$

that corresponds to some sample disturbance $w_s^k$. This disturbance also determines the one-stage-cost sample

$$g_s^k = g_k(x_s^k, u_s^k, w_s^k),$$

see Fig. 2.2.2).

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

$$\bar{r}_k \in \arg\min_{r_k} \sum_{s=1}^q (\hat{Q}_k(x_s^k, u_s^k, r_k) - \beta_s^k)^2.$$ \hspace{1cm} (2.11)

(c) Use the policy

$$\hat{\mu}_k(x_k) \in \arg\min_{u_k \in U_k(x_k)} \hat{Q}_k(x_k, u_k, \bar{r}_k).$$ \hspace{1cm} (2.12)

Note some important points about the preceding procedure:

(1) It is model-free in the sense that it is based on Monte Carlo simulation. Moreover, it does not need the functions $f_k$ and $g_k$, and the probability distribution of $w_k$ to generate the policy $\hat{\mu}_k$ through the least squares regression (2.11) and the Q-factor minimization (2.12). Using the simulator to collect the samples (2.10) and the cost function approximation $\hat{J}_{k+1}$ suffices.

(2) Two approximations are potentially required: One to compute $\hat{J}_{k+1}$, which is needed for the samples $\beta_s^k$ [cf. Eq. (2.10)], and another to compute $\hat{Q}_k$ through the regression (2.11). The approximation method-
Figure 2.2.2 Schematic illustration of Q-factor approximation, assuming approximate cost functions $\tilde{J}_{k+1}$ are known. The input to the system model or simulator are sample state-control pairs $(x_s^k, u_s^k)$, and the outputs are a next state sample $x_{s+1}^k$ and cost sample $g_s^k$. These correspond to a disturbance $w_s^k$ according to

$$x_{s+1}^k = f_k(x_s^k, u_s^k, w_s^k), \quad g_s^k = g_k(x_s^k, u_s^k, w_s^k).$$

The sample Q-factors $\beta_s^k$ are generated according to Eq. (2.10), and are used in the least squares regression (2.11) to yield a parametric Q-factor approximation $\tilde{Q}_k$ and the policy implementation (2.12).

ods to obtain $\tilde{J}_{k+1}$ and $\hat{Q}_k$ may be unrelated.

3. The policy $\hat{\mu}_k$ obtained through the minimization (2.12) is not the same as the one obtained through the minimization (2.7). There are two reasons for this. One is the approximation error introduced by the Q-factor architecture $\tilde{Q}_k$, and the other is the simulation error introduced by the finite-sample regression (2.11). We have to accept these sources of error as the price to pay for the convenience of not requiring a mathematical model for policy implementation.

Let us also mention a regularized variant of the least squares minimization in Eq. (2.11), where a positive multiple of the squared deviation $\|r - \hat{r}\|^2$ of $r$ from some initial guess $\hat{r}$ is added to the least squares objective. In some cases, a nonquadratic minimization may be used in place of Eq. (2.11) to determine $\tilde{r}_k$, but in this book we will focus on least squares exclusively. We refer to Chapter 3 for further discussion.

2.3 ROLLOUT AND THE POLICY IMPROVEMENT PRINCIPLE

The principal aim of rollout is policy improvement, i.e., start with a suboptimal/heuristic policy, called the base policy (or sometimes, the default policy, or the base heuristic), and produce an improved policy by limited lookahead minimization with use of the heuristic at the end. This latter policy is called the rollout policy, and the fact that it is indeed “improved” will be established, under various conditions; see what follows in this section, and also in Section 4.3, in the context of infinite horizon problems.

Truncated Rollout

In its pure $\ell$-step lookahead form, rollout can be defined very simply: it...
Sec. 2.3 Rollout and the Policy Improvement Principle

Figure 2.3.1 The structure of truncated rollout with $\ell$-step lookahead at state $x_k$. It involves:

(a) Multistep lookahead over $\ell$ stages (possibly $\ell = 1$).
(b) Running a heuristic algorithm/base policy from state $x_{k+\ell}$ for a number of steps, say $m$.
(c) Adding a cost function approximation at the end of the $\ell + m$ steps.

The approximate cost $\tilde{J}_{k+\ell}(x_{k+\ell})$ is the sum of the base policy costs of the $m$ stages plus the terminal cost function approximation from state $x_{k+\ell+m}$. The minimization produces the rollout control $\tilde{u}_k$, and the rollout policy $\tilde{\mu}_k(x_k) = \tilde{u}_k$.

is approximation in value space with the approximate cost-to-go values $\tilde{J}_{k+\ell}(x_{k+\ell})$ calculated by running the base policy, starting from each possible next state $x_{k+\ell}$. There is also a truncated variant of rollout for problems involving a long horizon, where the base policy is used for a limited number of steps, and some cost function approximation is added at the end to take into account the cost of the remaining steps (see Fig. 2.3.1). This cost function approximation is often simple (for example equal to zero), particularly when the base policy is run for a fairly large number of steps (or to the end of the horizon). It may also be quite complex, and may be obtained with a sophisticated off-line training process, involving for example a neural network (see Chapter 3).

The choice of the base policy is of course important for the performance of the rollout approach. However, experimental evidence has shown that surprisingly good rollout performance may be attained even with a relatively poor base heuristic, particularly when long multistep lookahead is used. A plausible explanation for this is that addition of the same constant to the values of $\tilde{J}_{k+\ell}(x_{k+\ell})$ for all relevant states $x_{k+\ell}$ leaves the rollout policy unaffected (see [Ber19a], Section 2.1.6, for further discussion of this point). For example, even if the base policy is grossly suboptimal and/or the terminal cost function approximation is poor, the rollout policy will perform well provided that the differences

$$\tilde{J}_{k+\ell}(x_{k+\ell}) - J^*_k(x_{k+\ell})$$

do not vary much over the set of relevant states $x_{k+\ell}$. Moreover, since the base policy is run to the end of the horizon, the rollout algorithm makes
control choices with “forward vision” and considers the states that might occur far into the future. This is not necessarily true for the base policy.

Generally speaking, there is no restriction on the base policy. It may be obtained in a variety of ways, including sophisticated off-line methods that are based on approximation in value or policy space. What is important in the context of rollout is that the cost-to-go values of the base policy starting from any state can be calculated in some way, possibly including simulation.

Rollout with Value and/or Policy Approximation

Implementation of the rollout policy on-line requires a lot of base policy evaluations, i.e., the evaluation of costs-to-go of the base policy starting from all possible next states $x_{k+1}$. This is a potential computational bottleneck, particularly for stochastic problems, so we may consider an approximation-based alternative, which is less computationally demanding. This alternative is based on off-line approximation in value space and/or approximation in policy space, using parametric architectures. We discussed this possibility in Section 2.1.3, in the context of approximation in value space built on top of approximation in policy space. We will provide some additional discussion here.

We may consider two approaches:

(a) **Off-line approximation of the Q-factors of the base policy**: Given the base policy $\pi$, it is possible to use it to calculate cost samples of the policy’s Q-factors for a collection of state-control pairs $(x^*_s, u^*_s)$, $s = 1, \ldots, q$. This can be done by simulation: for each $(x^*_s, u^*_s)$, we generate a random disturbance $w^*_s$, transition to the next state

$$x^*_{k+1} = f_k(x^*_s, u^*_s, w^*_s),$$

and corresponding transition cost

$$g_k(x^*_s, u^*_s, w^*_s),$$

and then use the base policy to calculate a sample of the transition costs of the remaining stages, starting from $x^*_{k+1}$. We can then use all these Q-factor samples and a parametric family of Q-factor functions,

$$\tilde{Q}_k(x_k, u_k, r_k),$$

where $r_k$ is the parameter vector, together with a least squares minimization or regression, to approximate the true Q-factor expression of the base policy,

$$Q_{k,\pi}(x_k, u_k) = E\left\{g_k(x_k, u_k, w_k) + J_{k+1,\pi}(f_k(x_k, u_k, w_k))\right\};$$
Sec. 2.3 Rollout and the Policy Improvement Principle

cf. Section 2.2.5 and Eq. (2.11). The parametric Q-factor function thus obtained can be used for approximate implementation of the rollout policy; cf. Eq. (2.12). A variation of this is a truncated rollout scheme, where a parametric approximation of the cost function of the base policy is used as terminal cost function approximation; cf. Section 2.1.3.

(b) Off-line approximation of the rollout policy: This alternative approach is based on off-line approximation in policy space. Here we compute a large number of sample state-rollout control pairs \((x_k^s, u_k^s)\), \(s = 1, \ldots, q\), i.e., for each \(s\), \(u_k^s\) is the rollout control at state \(x_k^s\). We can then introduce a parametric family of policies \(\tilde{\mu}_k(x_k, r_k)\), where \(r_k\) is a parameter vector, and choose the parameter \(r_k\) by solving the least squares/regression problem

\[
\min_{r_k} \sum_{s=1}^{q} \| u_k^s - \tilde{\mu}_k(x_k^s, r_k) \|^2
\]  

[cf. Eq. (2.3)]. This is approximation in policy space on top of approximation in value space, which we discussed in Section 2.2.4. It will yield an approximate implementation of the rollout policy.

It is also possible to use approximation in both value and policy space for parametric approximation of the rollout policy. First obtain parametric approximations \(\tilde{Q}_k(x_k, u_k, r_k)\) of the Q-factors of the base policy [as in (a) above], and then use them to obtain an approximation of the rollout policy [as in (b) above]. In particular, we generate many state-control samples \((x_k^s, u_k^s)\), \(s = 1, \ldots, q\), where \(u_k^s\) is the (approximate) rollout control obtained from the minimization

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

The samples \((x_k^s, u_k^s)\) are then used for approximation in policy space as described in (b) above; cf. Eq. (2.13).

Policy Iteration and Perpetual Rollout

As we have noted in Section 2.1.3, there is an important connection of rollout with the method of policy iteration for infinite horizon problems, which we will discuss in Chapter 4. We will see there that the pure form of the rollout policy (no truncation and no terminal cost function approximation) can be viewed as the result of a single policy iteration, starting from the base policy. Reversely, policy iteration can be viewed as a perpetual rollout algorithm, whereby a sequence of policies is generated, each one being the rollout policy obtained by using the preceding one as a base policy.
A major difficulty for implementing perpetual rollout is the need to compute multiple successive policies and/or evaluate (at least approximately their cost function values). This necessitates continuing/perpetual evaluations of cost functions and/or policies, typically using approximation architectures and training methods, which we described briefly earlier (cf. Fig. 2.1.6), and we will discuss further in subsequent chapters.

In this section we will describe rollout for finite-state deterministic problems. We will extend our discussion to stochastic problems in Section 2.4. Then, in Section 2.5 and later sections in this chapter we will discuss rollout for special types of problems, including infinite control spaces problems, model predictive control, multiagent, constrained DP, and minimax problems. We will revisit rollout schemes in Chapter 4, in the context of infinite horizon problems, where a variety of additional perpetual rollout schemes are possible under the general methodology of approximate policy iteration.

### 2.3.1 On-Line Rollout for Deterministic Discrete Optimization

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 $\hat{J}_{k+\ell}(x_{k+\ell})$. Discrete deterministic problems, including a very broad class of challenging combinatorial problems, such as scheduling, routing, assignment, etc, can be formulated as DP problems, as discussed in Section 1.3.1, and are particularly well-suited for rollout because they involve a finite number of control choices at each state.

In this section, we will develop the theory of rollout for deterministic problems, including the central issue of policy improvement. We will also illustrate several variants of the method with examples, and we will discuss issues of efficient implementation. We will return to deterministic problems in Sections 2.7 and 2.8, to discuss additional methods and 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 is also finite); cf. Section 1.3.1. 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

---

† 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).
sequence of states \( \{x_{k+1}, \ldots, x_N\} \) and controls \( \{u_{k+1}, \ldots, u_{N-1}\} \) such that
\[
x_{i+1} = f_i(x_i, u_i), \quad i = k, \ldots, N - 1.
\]

The rollout algorithm then applies the control that minimizes over \( u_k \in U_k(x_k) \) the tail cost expression for stages \( k \) to \( N \):
\[
g_k(x_k, u_k) + g_{k+1}(x_{k+1}, u_{k+1}) + \cdots + g_{N-1}(x_{N-1}, u_{N-1}) + g_N(x_N). \quad (2.14)
\]

Equivalently, and more succinctly, the rollout algorithm applies at state \( x_k \) the control \( \hat{\mu}_k(x_k) \) given by the minimization
\[
\hat{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k), \quad (2.15)
\]

where \( \tilde{Q}_k(x_k, u_k) \) is the approximate Q-factor defined by
\[
\tilde{Q}_k(x_k, u_k) = g_k(x_k, u_k) + H_{k+1}(f_k(x_k, u_k)), \quad (2.16)
\]

with \( H_{k+1}(x_{k+1}) \) denoting the cost of the base heuristic starting from state \( x_{k+1} \) [i.e., \( H_{k+1}(x_{k+1}) \) is the sum of all the terms in Eq. (2.14), except the first]; see Fig. 2.3.2. The rollout process defines a suboptimal policy \( \tilde{\pi} = \{\hat{\mu}_0, \ldots, \hat{\mu}_{N-1}\} \), referred to as the rollout policy, where for each \( x_k \) and \( k \), \( \hat{\mu}_k(x_k) \) is the control produced by the Q-factor minimization (2.15).

Note that the rollout algorithm requires running the base heuristic for a number of times that is bounded by \( Nm \), where \( m \) is an upper bound on the number of control choices available at each state. Thus if \( m \) is small relative to \( N \), the rollout algorithm requires computation equal to a small multiple of \( N \) times the computation time for a single application of the base heuristic. Similarly, if \( m \) 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 \( N \).
Figure 2.3.3 Schematic illustration of rollout with the nearest neighbor heuristic for the traveling salesman problem. The initial state $x_0$ consists of a single city. The final state $x_N$ is a complete tour of $N$ cities, containing each city exactly once.

Example 2.3.1 (Traveling Salesman Problem)

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

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 $c_{k+1} \neq c_0, c_1, \ldots, c_k$, 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 total cost

$$g(c_0, c_1) + \cdots + g(c_{N-2}, c_{N-1}) + g(c_{N-1}, c_0). \quad (2.17)$$

We can formulate the traveling salesman problem as a DP problem as we discussed in Example 1.3.1. We choose a starting city, say $c_0$, as the initial state $x_0$. Each state $x_k$ corresponds to a partial tour $(c_0, c_1, \ldots, c_k)$ consisting of distinct cities. The states $x_{k+1}$, next to $x_k$, are sequences of the form $(c_0, c_1, \ldots, c_k, c_{k+1})$ 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. (2.17).

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. 2.3.3.

Here, each application of the heuristic at stage requires computation proportional to $(N-k)!$, and is bounded by a number that is proportional to $N^2$. Since the number of control choices at any state is bounded by $N$, the corresponding computation for the rollout solution is bounded by a number that is proportional to $N^3$.

Cost Improvement with a Rollout Algorithm - Sequential Consistency

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

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

**Definition 2.3.1:** We say that the base heuristic is sequentially consistent if it has the property that when it generates the sequence

$$\{x_k,u_k,x_{k+1},u_{k+1},\ldots,x_N\}$$

starting from state $x_k$, it also generates the sequence
starting from state $x_{k+1}$.

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

As an example, the reader may verify that the nearest neighbor heuristic described in the traveling salesman Example 2.3.1 is sequentially consistent. Similar examples include the use of many types of greedy heuristics (Section 6.4 of the book [Ber17] 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.

Note that under sequential consistency the heuristic defines a legitimate DP policy. This is the policy that moves from $x_k$ to the state $x_{k+1}$ that lies on the path $\{x_k, x_{k+1}, \ldots, x_N\}$ generated by the base heuristic.

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

**Proposition 2.3.1: (Cost Improvement Under Sequential Consistency)** Consider the rollout policy $\tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$ obtained with a sequentially consistent base heuristic, and let $J_{k,\tilde{\pi}}(x_k)$ denote the cost obtained with $\tilde{\pi}$ starting from $x_k$ at time $k$. Then

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

(2.18)

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

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

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

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

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

$$= H_k(x_k),$$

(2.19)
where:
(a) The first equality is the DP equation for the rollout policy \( \tilde{\pi} \).
(b) The first inequality holds by the induction hypothesis.
(c) The second equality holds by the definition of the rollout algorithm.
(d) The third equality is the DP equation for the policy that corresponds to the base heuristic (this is the step where we need sequential consistency).

This completes the induction proof of the cost improvement property (2.18). Q.E.D.

**Sequential Improvement**

We will next show that the rollout policy has no worse performance than its base heuristic under a condition that is weaker than sequential consistency. Let us recall that the rollout algorithm \( \tilde{\pi} = \{ \tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1} \} \) is defined by the minimization
\[
\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k),
\]
where \( \tilde{Q}_k(x_k, u_k) \) is the approximate Q-factor defined by
\[
\tilde{Q}_k(x_k, u_k) = g_k(x_k, u_k) + H_{k+1}(f_k(x_k, u_k)),
\]
[cf. Eqs. (2.16)], and \( H_{k+1}(f_k(x_k, u_k)) \) denotes the cost of the base heuristic starting from state \( f_k(x_k, u_k) \).

**Definition 2.3.2:** We say that the base heuristic is *sequentially improving* if for all \( x_k \) and \( k \), we have
\[
\min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k) \leq H_k(x_k).
\]

In words, the sequential improvement property (2.20) states that
\[
\text{Minimal heuristic Q-factor at } x_k \leq \text{Heuristic cost at } x_k. \tag{2.21}
\]

Note that *when the heuristic is sequentially consistent it is also sequentially improving*. This follows from Eq. (2.21), since for a sequentially consistent heuristic, the heuristic cost at \( x_k \) is equal to the Q-factor of the control \( \pi_k \) that the heuristic applies at \( x_k \),
\[
\tilde{Q}_k(x_k, \pi_k) = g_k(x_k, \pi_k) + H_{k+1}(f_k(x_k, \pi_k)),
\]
which is greater or equal to the minimal Q-factor at \( x_k \). This implies Eq. (2.20). We will show that a sequentially improving heuristic yields policy improvement.

**Proposition 2.3.2: (Cost Improvement Under Sequential Improvement)** Consider the rollout policy \( \tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\} \) obtained with a sequentially improving base heuristic, and let \( J_{k, \tilde{\pi}}(x_k) \) denote the cost obtained with \( \tilde{\pi} \) starting from \( x_k \) at time \( k \). Then

\[
J_{k, \tilde{\pi}}(x_k) \leq H_k(x_k), \quad \text{for all } x_k \text{ and } k, \tag{2.22}
\]

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

**Proof:** Follows from the calculation of Eq. (2.19), by replacing the last two steps (that rely on sequential consistency) with Eq. (2.20). 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 \( x_k \). Empirically, it has been observed that the cost improvement obtained by rollout with a sequentially improving heuristic is typically considerable and often dramatic. Generally, however, it is hard to provide solid theoretical support for this observation. Several case studies indicate consistently good performance of rollout (at least in the pure form described above); see the end of chapter references. The textbook [Ber17], Section 6.4, provides some detailed worked-out examples. The price for the performance improvement is extra computation that is typically equal to the computation time of the base heuristic times a factor that is a low order polynomial of \( N \).

On the other hand the sequential improvement condition may not hold for a given base heuristic. This is not surprising since any heuristic (no matter how inconsistent or silly) is in principle admissible to use as base heuristic. Here is an example:

**Example 2.3.2 (Sequential Improvement Violation)**

Consider the 2-stage problem shown in Fig. 2.3.4, which involves two states at each of stages 1 and 2, and the controls shown. Suppose that the unique optimal trajectory is \((x_0, u_0^*, x_1^*, u_1^*, x_2^*)\), and that the base heuristic produces this optimal trajectory starting at \( x_0 \). The rollout algorithm chooses a control at \( x_0 \) as follows: it runs the base heuristic to construct a trajectory starting from \( x_1^* \) and \( \tilde{x}_1 \), with corresponding costs \( H_1(x_1^*) \) and \( H_1(\tilde{x}_1) \). If

\[
g_0(x_0, u_0^*) + H_1(x_1^*) > g_0(x_0, \tilde{u}_0) + H_1(\tilde{x}_1), \tag{2.23}
\]

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
Sec. 2.3 Rollout and the Policy Improvement Principle

Optimal Trajectory Chosen by Base Heuristic at $x_0$

$x_0^*$

High Cost Transition Chosen by Heuristic at $x_1^*$

Rollout Choice

Figure 2.3.4 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.

$x_1^*$ (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 (2.23) holds then the heuristic is not sequentially improving at $x_0$, i.e., that

$$H_0(x_0) < \min \{ g_0(x_0, u_0^*) + H_1(x_1^*), g_0(x_0, \tilde{u}_0) + H_1(x_1^*) \}.$$  

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 $(x_0, u_0^*, x_1^*, \tilde{u}_1, \tilde{x}_2)$, and

$$g_0(x_0, \tilde{u}_0) + H_1(x_1^*),$$

which is the cost of the trajectory $(x_0, \tilde{u}_0, \tilde{x}_1, \tilde{u}_1, \tilde{x}_2)$.

The preceding example suggests a simple enhancement to the rollout algorithm, which detects when the sequential improvement condition is violated, and takes corrective measures. In particular, it discards the choice of the rollout algorithm if this leads to a trajectory that is inferior to the best trajectory obtained so far; see the discussion of the “fortified” variant of the algorithm in Section 2.3.2.

Using Multiple Heuristics

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

In particular, let us assume that we have $M$ base heuristics, and that the $m$th of these, given a state $x_{k+1}$, produces a trajectory $\tilde{T}^m_{k+1} = \{x_{k+1}, \tilde{u}^m_{k+1}, \ldots, \tilde{x}^m_{N-1}, \tilde{x}^m_N\}$, and corresponding cost $C(\tilde{T}^m_{k+1})$. The superheuristic then produces at $x_{k+1}$ the trajectory $\tilde{T}^m_{k+1}$ for which $C(\tilde{T}^m_{k+1})$ is minimum.

An interesting property, which can be readily verified by using the definitions, is that if all the base heuristics are sequentially improving, the same is true for the superheuristic. Indeed, let us write the sequential improvement condition (2.20) for each of the base heuristics

$$
\min_{u_k \in U_k(x_k)} \hat{Q}^m_k(x_k, u_k) \leq H^m_k(x_k), \quad m = 1, \ldots, M,
$$

for all $x_k$ and $k$, where $\hat{Q}^m_k(x_k, u_k)$ and $H^m_k(x_k)$ are Q-factors and heuristic costs that correspond to the $m$th base heuristic. Then by taking minimum over $m$, we have

$$
\min_{m=1,\ldots,M} \min_{u_k \in U_k(x_k)} \hat{Q}^m_k(x_k, u_k) \leq \min_{m=1,\ldots,M} H^m_k(x_k),
$$

for all $x_k$ and $k$. By interchanging the order of the minimizations of the left side, we then obtain

$$
\min_{u_k \in U_k(x_k)} \min_{m=1,\ldots,M} \hat{Q}^m_k(x_k, u_k) \leq \min_{m=1,\ldots,M} H^m_k(x_k),
$$

which is precisely the sequential improvement condition for the superheuristic.

### 2.3.2 The Fortified Rollout Algorithm

We will describe a variant of the rollout algorithm 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 $x_k$, it stores the trajectory $T_k = \{x_0, u_0, \ldots, u_{k-1}, x_k\}$

† Another 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.
that has been constructed up to stage $k$, called permanent trajectory, and it also stores a tentative best trajectory
\[ \mathcal{T}_k = \{x_k, u_k, x_{k+1}, u_{k+1}, \ldots, x_{N-1}, u_N\} \]
with corresponding cost
\[ C(\mathcal{T}_k) = g_k(x_k, u_k) + g_{k+1}(x_{k+1}, u_{k+1}) + \cdots + g_{N-1}(x_{N-1}, u_{N-1}) + g_N(x_N). \]

The tentative best trajectory is such that $P_k \cup \mathcal{T}_k$ is the best end-to-end trajectory computed up to stage $k$ of the algorithm. Initially, $\mathcal{T}_0$ 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 $\mathcal{T}_k$, and use $\mathcal{T}_k$ instead.\†

In particular, upon reaching state $x_k$, we run the rollout algorithm as earlier, i.e., for every $u_k \in U_k(x_k)$ and next state $x_{k+1} = f_k(x_k, u_k)$, we run the base heuristic from $x_{k+1}$, and find the control $\tilde{u}_k$ that gives the best trajectory, denoted
\[ \tilde{T}_k = \{x_k, \tilde{u}_k, \tilde{x}_{k+1}, \tilde{u}_{k+1}, \ldots, \tilde{u}_{N-1}, \tilde{x}_N\} \]
with corresponding cost
\[ C(\tilde{T}_k) = g_k(x_k, \tilde{u}_k) + g_{k+1}(\tilde{x}_{k+1}, \tilde{u}_{k+1}) + \cdots + g_{N-1}(\tilde{x}_{N-1}, \tilde{u}_{N-1}) + g_N(\tilde{x}_N). \]

Whereas the ordinary rollout algorithm would choose control $\tilde{u}_k$ and move to $\tilde{x}_{k+1}$, the fortified algorithm compares $C(\mathcal{T}_k)$ and $C(\tilde{T}_k)$, and depending on which of the two is smaller, chooses $\tilde{u}_k$ or $u_k$ and moves to $\tilde{x}_{k+1}$ or to $x_{k+1}$, respectively. In particular, if $C(\mathcal{T}_k) \leq C(\tilde{T}_k)$ the algorithm sets the next state and corresponding tentative best trajectory to
\[ x_{k+1} = x_{k+1}, \quad \mathcal{T}_{k+1} = \{x_{k+1}, u_{k+1}, \ldots, u_{N-1}, x_N\}, \]
and if $C(\mathcal{T}_k) > C(\tilde{T}_k)$ it sets the next state and corresponding tentative best trajectory to
\[ x_{k+1} = \tilde{x}_{k+1}, \quad \tilde{T}_{k+1} = \{\tilde{x}_{k+1}, \tilde{u}_{k+1}, \ldots, \tilde{u}_{N-1}, \tilde{x}_N\}. \]

In other words the fortified rollout at $x_k$ follows the current tentative 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

\† 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 tedious and will not be given; we refer to Bertsekas, Tsitsiklis, and Wu [BTW97], and [Ber17], Section 6.4.2.
Figure 2.3.5 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 
\[
\mathcal{T}_k = \{x_k, u_k, x_{k+1}, u_{k+1}, \ldots, u_{N-1}, x_N\}
\]
such that \( \mathcal{P}_k \cup \mathcal{T}_k \) is the best end-to-end trajectory computed so far. We now run the rollout algorithm at \( x_k \), i.e., we find the control \( \tilde{u}_k \) that minimizes over \( u_k \) the sum of \( g_k(x_k, u_k) \) plus the heuristic cost from the state \( x_{k+1} = f_k(x_k, u_k) \), and the corresponding trajectory 
\[
\tilde{T}_k = \{x_k, \tilde{u}_k, \tilde{x}_{k+1}, u_{k+1}, \ldots, u_{N-1}, x_N\}.
\]
If the cost of the end-to-end trajectory \( \mathcal{P}_k \cup \tilde{T}_k \) is lower than the cost of \( \mathcal{P}_k \cup \mathcal{T}_k \), we add \( (\tilde{u}_k, \tilde{x}_{k+1}) \) to the permanent trajectory and set the tentative best trajectory to 
\[
\mathcal{T}_{k+1} = \{x_{k+1}, u_{k+1}, \ldots, u_{N-1}, x_N\}.
\]
Otherwise we add \( (u_k, x_{k+1}) \) to the permanent trajectory and set the tentative best trajectory to 
\[
\mathcal{T}_{k+1} = \{x_{k+1}, u_{k+1}, \ldots, u_{N-1}, x_N\}.
\]

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

every state the trajectory that consists of the union of the permanent and the tentative best trajectories, 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.
Example 2.3.3

Let us consider the application of the fortified rollout algorithm to the problem of Example 2.3.2. The fortified rollout algorithm stores as initial tentative best trajectory the unique 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 permanent trajectory to \((x_0, u_0^*, x_1^*)\) and the tentative best trajectory to \((x_1^*, u_1^*, x_2^*)\).

We finally note that the fortified rollout algorithm can be used in a different setting to restore and maintain the cost improvement property. Suppose in particular that the rollout minimization at each step is performed with approximations. For example the control \(u_k\) may have multiple independently constrained components, i.e.,

\[
u_k = (u_k^1, \ldots, u_k^m), \quad U_k(x_k) \in U_1^1(x_k) \times \cdots \times U_m^m(x_k),
\]

and 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, 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.

2.3.3 Truncated Rollout Algorithms

We may incorporate multistep lookahead into the rollout framework. This may be essential in problems where at a given state, the effect of the first control may be small relative to the effect of the remaining controls.

To describe two-step lookahead for deterministic problems, in its most straightforward implementation, suppose that after \(k\) steps we have reached state \(x_k\). We then consider the set of all two-step-ahead states \(x_{k+2}\), we run the base heuristic starting from each of them, and compute the two-stage cost to get from \(x_k\) to \(x_{k+2}\), plus the cost of the base heuristic from

\[
\uparrow \quad \text{Such a situation arises in the context of multiagent rollout for deterministic discrete optimization problems; see Sections 2.6-2.8.}
\]
Figure 2.3.6 Illustration of truncated rollout with two-step lookahead and a terminal cost approximation \( \tilde{J} \). The base heuristic is used for a limited number of steps and the terminal cost is added to compensate for the remaining steps.

We select the state, say \( \tilde{x}_{k+2} \), that is associated with minimum cost, compute the controls \( \tilde{u}_k \) and \( \tilde{u}_{k+1} \) that lead from \( x_k \) to \( \tilde{x}_{k+2} \), and choose \( \tilde{u}_k \) as the next rollout control and \( x_{k+1} = f_k(x_k, \tilde{u}_k) \) as the next state; see Fig. 2.3.6. 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. Moreover, the \( \ell \)-step lookahead minimization may involve approximations aimed at simplifying the associated computations.

An important variation for problems with a large number of stages, 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. 2.3.6. One possibility that works well for many problems is to simply set the terminal cost approximation to zero. Alternatively, the terminal cost function approximation may be obtained by problem approximation or by running another less expensive heuristic or by using some sophisticated off-line training process that may involve an approximation architecture such as a neural network (see Chapter 3).

Among other variations of deterministic multistep rollout, let us mention a fortified version, which maintains a tentative best trajectory, along
Sec. 2.4 Stochastic Rollout and Monte Carlo Tree Search

the lines described earlier for the one-step lookahead case. In still another version of \(\ell\)-step lookahead rollout, we may consider “pruning” the lookahead tree. 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. 2.3.7. This may be viewed as selective depth lookahead, and aims to limit the number of times that the base heuristic is applied, which can become overwhelming as the length of lookahead is increased. We will encounter again the idea of selective depth lookahead in the context of stochastic rollout and Monte Carlo tree search (see the next section), where in addition to the length of lookahead, the accuracy of the simulation to evaluate the cost of the base heuristic is adapted to the results of earlier computations.

Finally, let us mention a variant of rollout that maintains multiple trajectories, extending from a given state \(x_k\) to possibly multiple next states \(x_{k+1}\). These states are the ones considered “most promising” based on the current results of the base heuristic (like being “\(\epsilon\)-best”), but may be discarded later based on subsequent computations. Such extended forms of rollout are restricted to deterministic problems, and tend to be problem-dependent.

2.4 STOCHASTIC ROLLOUT AND MONTE CARLO TREE SEARCH

We will now discuss the extension of the rollout algorithm to stochastic finite-state DP problems. We will restrict ourselves to the case where the base heuristic is a policy \(\pi = \{\mu_0, \ldots, \mu_{N-1}\}\) (i.e., is sequentially consistent, in the context of deterministic problems). It is possible to consider more general rollout algorithms that involve base heuristics with a sequential improvement property, but we will not pursue this idea, as it does not seem to have been applied so far in interesting stochastic contexts.

We first note that the cost improvement property that we showed for deterministic problems under the sequential consistency condition carries through for stochastic problems. In particular, let us denote by \(J_{k,\pi}(x_k)\) the cost corresponding to starting the base policy at state \(x_k\), and by \(J_{k,\tilde{\pi}}(x_k)\) the cost corresponding to starting the rollout algorithm at state \(x_k\). We claim that

\[
J_{k,\tilde{\pi}}(x_k) \leq J_{k,\pi}(x_k), \quad \text{for all } x_k \text{ and } k.
\]

We prove this inequality by induction similar to the deterministic case [cf. Eq. (2.19)]. Clearly it holds for \(k = N\), since \(J_{N,\tilde{\pi}} = J_{N,\pi} = g_N\). Assuming that it holds for index \(k + 1\), we have for all \(x_k\),

\[
J_{k,\tilde{\pi}}(x_k) = E\left\{g_k(x_k, \hat{\mu}_k(x_k), w_k) + J_{k+1,\tilde{\pi}}\left(f_k(x_k, \hat{\mu}_k(x_k), w_k)\right)\right\}
\]
Selective Depth Tree Projections of Leafs of the Tree

Current State $x_0$

$\text{Base Heuristic}$

$\text{States } x_N$

**Figure 2.3.7** 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 ($x_{k+1}^1$, $x_{k+1}^2$, $x_{k+1}^3$, $x_{k+1}^4$ 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 $x_{k+1}^i$ that corresponds to smallest overall cost is chosen by the selective depth lookahead rollout algorithm. For problems with a large number of stages, we can also truncate the rollout trajectories and add a terminal cost function approximation as compensation for the resulting error; cf. Fig. 2.3.6.

$$
\leq E \left\{ g_k(x_k, \tilde{\mu}_k(x_k), w_k) + J_{k+1, \pi}(f_k(x_k, \tilde{\mu}_k(x_k), w_k)) \right\}
= \min_{u_k \in U_k(x_k)} E \left\{ g_k(x_k, u_k, w_k) + J_{k+1, \pi}(f_k(x_k, u_k, w_k)) \right\}
\leq E \left\{ g_k(x_k, \mu_k(x_k), w_k) + J_{k+1, \pi}(f_k(x_k, \mu_k(x_k), w_k)) \right\}
= J_{k, \pi}(x_k),
$$

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.
The third equality is the DP equation for the policy \( \pi \) that corresponds to the base policy.

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

Similar to deterministic problems, it has been observed empirically that for stochastic problems the rollout policy not only does not deteriorate the performance of the base policy, but also typically produces substantial cost improvement; see the case studies referenced at the end of the chapter.

**Example 2.4.1 (Stochastic Scheduling - The Quiz Problem)**

This is a well-known problem, which together with its extensions illustrates a broad class of stochastic scheduling problems. In its simplest form, a person is given a list of \( N \) questions and can answer these questions in any order he/she chooses. Question \( i \) will be answered correctly with probability \( p_i \), and the person will then receive a reward \( v_i \). At the first incorrect answer, the quiz terminates and the person is allowed to keep his or her previous rewards. The problem is to choose the ordering of questions so as to maximize the total expected reward.

There are \( N \) stages corresponding to the \( N \) questions. At each stage the state is the set of questions answered so far plus a binary variable that indicates that either all the questions asked so far have been answered correctly, or whether some question was answered incorrectly, in which case the system has moved to a cost-free termination state. In its simplest form, the problem admits an optimal policy that can be easily implemented. Assuming that at stage \( k \), we have answered correctly all the \( k \) questions asked so far, we order the complementary set of questions in decreasing order of the “index of preference” \( p_i v_i / (1 - p_i) \), and we answer next the question that yields maximum index value. This is known as the *index policy*. The proof of the optimality of this policy is fairly simple. It is based on finite horizon stochastic DP algorithm and can be found for example in the textbook [Ber17].

Unfortunately, however, with only minor changes in the structure of the problem, the index policy need not be optimal. Examples of difficult variations of the problem may involve one or more of the following characteristics:

(a) A limit on the maximum number of questions that can be answered, which is smaller than the number of questions \( N \). To see that the index policy is not optimal anymore, consider the case where there are two questions, only one of which may be answered. Then it is optimal to answer the question that offers the maximum expected reward \( p_i v_i \) rather than the maximum index \( p_i v_i / (1 - p_i) \).

(b) A time window for each question, which constrains the set of time slots when each question may be answered. Time windows may also be combined with the option to refuse answering a question at a given period, when either no question is available during the period, or answering any one of the available questions involves excessive risk.

(c) Precedence constraints, whereby the set of questions that can be answered in a given time slot depends on the immediately preceding question, and possibly on some earlier answered questions.
(d) Sequence-dependent rewards, whereby the reward from answering correctly a given question depends on the immediately preceding question, and possibly on some earlier answered questions.

Nonetheless, even when the index policy is not optimal, it can be used as a base policy for the rollout algorithm. The reason is that at a given state, the index policy together with its expected reward can be easily calculated. In particular, each feasible question order \((i_1, \ldots, i_N)\) has expected reward equal to

\[
p_{i_1}(v_{i_1} + p_{i_2}(v_{i_2} + p_{i_3}(\cdots + p_{i_N}v_{i_N})\cdots)),
\]

Thus the rollout algorithm that uses the index heuristic as a base policy operates as follows: at a state where all questions asked so far have been answered correctly, we consider the set of questions \(J\) that are eligible to be answered next. For each question \(j \in J\), we consider a sequence of questions that starts with \(j\) and continues with the remaining questions chosen according to the index rule. We compute the expected reward of the sequence, denoted \(R(j)\), using the above formula. Then among the questions \(j \in J\), we choose to answer next the one with maximal \(R(j)\). A computational study of rollout algorithms for challenging instances of the quiz problem and related scheduling problems, using a few different types of base policies, is given in the paper [BeC99].

### 2.4.1 Simulation-Based Implementation of the Rollout Algorithm

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

\[
x_{i+1} = f_i(x_i, \mu_i(x_i), w_i), \quad i = k + 1, \ldots, N - 1,
\]

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

\[
x_{k+1} = f_k(x_k, u_k, w_k),
\]

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

\[
Q_{k, \pi}(x_k, u_k) = E\left\{g_k(x_k, u_k, w_k) + J_{k+1, \pi}(f_k(x_k, u_k, w_k))\right\},
\]

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

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

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

(2.24)

Example 2.4.2 (Backgammon)

The first impressive application of rollout was given for the ancient two-player game of backgammon, in the paper by Tesauro and Galperin [TeG96]; see Fig. 2.4.1. They implemented a rollout algorithm, which attained a level of play that was better than all computer backgammon programs, and eventually better than the best humans. Tesauro had proposed earlier the use of one-step and two-step lookahead with lookahead cost function approximation provided by a neural network, resulting in a backgammon program called TD-Gammon [Tes89a], [Tes89b], [Tes92], [Tes94], [Tes95], [Tes02]. TD-Gammon was trained with an approximate policy iteration method of the type to be discussed in Chapter 4, and was used as the base policy (for each of the two players) to simulate game trajectories. The rollout algorithm also involved truncation of long game trajectories, using a terminal cost function approximation based on TD-Gammon’s position evaluation. Game trajectories are of course random, since they involve the use of dice at each player’s turn. Thus the scores of many trajectories have to be generated and Monte Carlo averaged to assess the probability of a win from a given position.

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

Most of the currently existing computer backgammon programs descend from TD-Gammon. Rollout-based backgammon programs are the most powerful in terms of performance, consistent with the principle that a rollout algorithm performs better than its base heuristic. However, they are too time-consuming for real-time play, because of the extensive on-line simulation
requirement at each move (the situation in backgammon is exacerbated by its high branching factor, i.e., for a given position, the number of possible successor positions is quite large, as compared for example with chess). They have been used in a limited diagnostic way to assess the quality of neural network-based programs (many articles and empirical works on computer backgammon are posted on-line; see e.g., http://www.bkgm.com/articles/page07.html).

2.4.2 Variance Reduction in Rollout - Comparing Advantages

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

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

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

An alternative approach is to approximate the Q-factor difference 
\(\tilde{Q}_{k,\pi}(x_k, u_k) - \tilde{Q}_{k,\pi}(x_k, \tilde{u}_k)\) by sampling the difference 
\[ C_k(x_k, u_k, w_k) - C_k(x_k, \tilde{u}_k, w_k), \quad (2.25) \]
where \(w_k = (w_k, w_{k+1}, \ldots, w_{N-1})\) is the same disturbance sequence for the two controls \(u_k\) and \(\tilde{u}_k\), and
\[
C_k(x_k, u_k, w_k) = g_N(x_N) + g_k(x_k, u_k, w_k) + \sum_{i=k+1}^{N-1} g_i(x_i, \mu_i(x_i), w_i),
\]
with \(\{\mu_{k+1}, \ldots, \mu_{N-1}\}\) being the tail portion of the base policy. For this to make sense, it is necessary that the probability distribution of each disturbance \(w_i\) does not depend on \(x_i\) and \(u_i\).

The approximation that is based on the difference (2.25), which involves a common disturbance \(w_k\) for \(u_k\) and \(\tilde{u}_k\), may be far more accurate than the one obtained by differencing samples of \(C_k(x_k, u_k, w_k)\) and \(C_k(x_k, \tilde{u}_k, w_k)\), which involve two different disturbances \(w_k\) and \(\tilde{w}_k\). Indeed, by introducing the zero mean sample errors 
\[ D_k(x_k, u_k, w_k) = C_k(x_k, u_k, w_k) - \tilde{Q}_{k,\pi}(x_k, u_k), \]
it can be seen that the variance of the error in estimating \(\tilde{Q}_{k,\pi}(x_k, u_k) - \tilde{Q}_{k,\pi}(x_k, \tilde{u}_k)\) with the former method will be no larger than with the latter method if and only if 
\[
E_{w_k, \tilde{w}_k} \left\{ \left| D_k(x_k, u_k, w_k) - D_k(x_k, \tilde{u}_k, \tilde{w}_k) \right|^2 \right\} 
\geq E_{w_k} \left\{ \left| D_k(x_k, u_k, w_k) - D_k(x_k, \tilde{u}_k, w_k) \right|^2 \right\}.
\]
By expanding the quadratic forms and using the fact \(E\{D_k(x_k, u_k, w_k)\} = 0\), we see that this condition is equivalent to 
\[ E\{D_k(x_k, u_k, w_k)D_k(x_k, \tilde{u}_k, w_k)\} \geq 0; \quad (2.26) \]
i.e., the errors \(D_k(x_k, u_k, w_k)\) and \(D_k(x_k, \tilde{u}_k, w_k)\) being nonnegatively correlated. A little thought should convince the reader that this property is likely to hold in many types of problems.

Roughly speaking, the relation (2.26) holds if changes in the value of \(u_k\) (at the first stage) have little effect on the value of the error \(D_k(x_k, u_k, w_k)\).
relative to the effect induced by the randomness of $w_k$. To see this, suppose that there exists a scalar $\gamma < 1$ such that, for all $x_k$, $u_k$, and $\hat{u}_k$, there holds

$$E \left\{ |D_k(x_k, u_k, w_k) - D_k(x_k, \hat{u}_k, w_k)|^2 \right\} \leq \gamma E \left\{ |D_k(x_k, u_k, w_k)|^2 \right\}. \tag{2.27}$$

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

$$D_k(x_k, u_k, w_k)D_k(x_k, \hat{u}_k, w_k)$$

$$\geq |D_k(x_k, u_k, w_k)|^2$$

$$- |D_k(x_k, u_k, w_k)| \cdot |D_k(x_k, \hat{u}_k, w_k) - D_k(x_k, u_k, w_k)|,$$

from which we obtain

$$E \{ D_k(x_k, u_k, w_k)D_k(x_k, \hat{u}_k, w_k) \}$$

$$\geq E \left\{ |D_k(x_k, u_k, w_k)|^2 \right\}$$

$$- E \left\{ |D_k(x_k, u_k, w_k)| \cdot |D_k(x_k, \hat{u}_k, w_k) - D_k(x_k, u_k, w_k)| \right\}$$

$$\geq E \left\{ |D_k(x_k, u_k, w_k)|^2 \right\} - \frac{1}{2} E \left\{ |D_k(x_k, u_k, w_k)|^2 \right\}$$

$$- \frac{1}{2} E \left\{ |D_k(x_k, \hat{u}_k, w_k) - D_k(x_k, u_k, w_k)|^2 \right\}$$

$$\geq \frac{1 - \gamma}{2} E \left\{ |D_k(x_k, u_k, w_k)|^2 \right\},$$

where for the second inequality we use the generic relation $-|a| \cdot |b| \geq -\frac{1}{4}(a^2 + b^2)$ for two scalars $a$ and $b$, and for the third inequality we use Eq. (2.27).

Thus, under the assumption (2.27), the condition (2.26) holds and guarantees that by averaging cost difference samples rather than differencing (independently obtained) averages of cost samples, the simulation error variance does not increase.

Let us finally note the potential benefit of using Q-factor differences in contexts other than rollout. In particular when approximating Q-factors $Q_{k, \pi}(x_k, u_k)$ using parametric architectures (Section 3.4 in the next chapter), it may be important to approximate and compare instead the differences

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

The function $A_{k, \pi}(x_k, u_k)$ is also known as the advantage of the pair $(x_k, u_k)$, and can serve just as well as $Q_{k, \pi}(x_k, u_k)$ for the purpose of comparing controls, but may work better in the presence of approximation errors. The use of advantages will be discussed further in Chapter 3.
2.4.3 Monte Carlo Tree Search

In the rollout implementation just described, we implicitly assumed that once we reach state $x_k$, we generate the same large number of trajectories starting from each pair $(x_k, u_k)$, with $u_k \in U(x_k)$, to the end of the horizon. The drawback of this is threefold:

(a) The trajectories may be too long because the horizon length $N$ is large (or infinite, in an infinite horizon context).

(b) Some of the controls $u_k$ may be clearly inferior to others, and may not be worth as much sampling effort.

(c) Some of the controls $u_k$ that appear to be promising, may be worth exploring better through multistep lookahead.

This has motivated lookahead variants, generally referred to as Monte Carlo tree search (MCTS for short), which aim to trade off computational economy with a hopefully small risk of degradation in performance. Such variants involve, among others, early discarding of controls deemed to be inferior based on the results of preliminary calculations, and simulation that is limited in scope (either because of a reduced number of simulation samples, or because of a shortened horizon of simulation, or both).

In particular, a simple remedy for (a) above is to use rollout trajectories of reasonably limited length, with some terminal cost approximation at the end (in an extreme case, the rollout may be skipped altogether for some states, i.e., rollout trajectories have zero length). The terminal cost function may be very simple (such as zero) or may be obtained through some auxiliary calculation. In fact the base policy used for rollout may be used to construct the terminal cost function approximation, as noted for the rollout-based backgammon algorithm of Example 2.4.2. In particular, an approximation to the cost function of the base policy may be obtained by training some approximation architecture, such as a neural network (see Chapter 3), and may be used as a terminal cost function.

A simple but less straightforward remedy for (b) is to use some heuristic or statistical test to discard some of the controls $u_k$, as soon as this is suggested by the early results of simulation. Similarly, to implement (c) one may use some heuristic to increase the length of lookahead selectively for some of the controls $u_k$. This is similar to the selective depth lookahead procedure for deterministic rollout that we illustrated in Fig. 2.3.7.

The MCTS approach can be based on sophisticated procedures for implementing and combining the ideas just described. The implementation is often adapted to the problem at hand, but the general idea is to use the interim results of the computation and statistical tests to focus the simulation effort along the most promising directions. Thus to implement MCTS one needs to maintain a lookahead tree, which is expanded as the relevant Q-factors are evaluated by simulation, and which balances the competing desires of exploitation and exploration (generate and evaluate
controls that seem most promising in terms of performance versus assessing
the potential of inadequately explored controls). Ideas that were developed
in the context of multiarmed bandit problems have played an important
role in the construction of this type of MCTS procedures (see the end-of-
chapter references).

Example 2.4.3 (Statistical Tests for Adaptive Sampling with
a One-Step Lookahead)

Let us consider a typical one-step lookahead selection strategy that is based
on adaptive sampling. We are at a state $x_k$ and we try to find a control $u_k$
that minimizes an approximate Q-factor

$$
\tilde{Q}_k(x_k, u_k) = E \left\{ q_k(x_k, u_k, w_k) + \tilde{j}_{k+1}(f_k(x_k, u_k, w_k)) \right\}
$$

over $u_k \in U_k(x_k)$, with $\tilde{Q}_k(x_k, u_k)$ computed by averaging samples of the
expression within braces. We assume that $U_k(x_k)$ contains $m$ elements, which
for simplicity are denoted $1, \ldots, m$. At the $\ell$th sampling period, knowing the
outcomes of the preceding sampling periods, we select one of the $m$ controls,
say $i_\ell$, and we draw a sample of $\tilde{Q}_k(x_k, i_\ell)$, whose value is denoted by $S_{i_\ell}$.
Thus after the $n$th sampling period we have an estimate $Q_{i,n}$ of the Q-factor
of each control $i = 1, \ldots, m$ that has been sampled at least once, given by

$$
Q_{i,n} = \frac{\sum_{\ell=1}^{n} \delta(i_\ell = i) S_{i_\ell}}{\sum_{\ell=1}^{n} \delta(i_\ell = i)}
$$

where

$$
\delta(i_\ell = i) = \begin{cases} 
1 & \text{if } i_\ell = i, \\
0 & \text{if } i_\ell \neq i.
\end{cases}
$$

Thus $Q_{i,n}$ is the empirical mean of the Q-factor of control $i$ (total sample
value divided by total number of samples), assuming that $i$ has been sampled
at least once.

After $n$ samples have been collected, with each control sampled at least
once, we may declare the control $i$ that minimizes $Q_{i,n}$ as the “best” one,
i.e., the one that truly minimizes the Q-factor $Q_k(x_k, i)$. However, there is
a positive probability that there is an error: the selected control may not
minimize the true Q-factor. In adaptive sampling, roughly speaking, we want
to design the sample selection strategy and the criterion to stop the sampling,
in a way that keeps the probability of error small (by allocating some sampling
effort to all controls), and the number of samples limited (by not wasting
samples on controls $i$ that appear inferior based on their empirical mean
$Q_{i,n}$).

Intuitively, a good sampling policy will balance at time $n$ the desires of
exploitation and exploration (i.e., sample controls that seem most promising,
in the sense that they have a small empirical mean $Q_{i,n}$, versus assessing the
potential of inadequately explored controls, those $i$ that have been sampled
a small number of times). Thus it makes sense to sample next the control $i$
that minimizes the sum

$$
T_{i,n} + R_{i,n}
$$
Sec. 2.4 Stochastic Rollout and Monte Carlo Tree Search

Figure 2.4.2 Illustration of one-step lookahead MCTS at a state $x_k$. The Q-factor sampled next corresponds to the control $i$ with minimum sum of exploitation index (here taken to be the running average $Q_{i,n}$) and exploration index ($R_{i,n}$, possibly given by the UCB rule).

of two indexes: an exploitation index $T_{i,n}$ and an exploration index $R_{i,n}$. Usually the exploitation index is chosen to be the empirical mean $Q_{i,n}$. The exploration index is based on a confidence interval formula and depends on the sample count

$$s_i = \sum_{\ell=1}^{n} \delta(i_\ell = i)$$

of control $i$. A frequently suggested choice is the UCB rule (upper confidence bound), which sets

$$R_{i,n} = -c \sqrt{\frac{\log n}{s_i}},$$

where $c$ is a positive constant that is selected empirically (some analysis suggests values near $c = \sqrt{2}$, assuming that $Q_{i,n}$ is normalized to take values in the range $[-1,0]$). The UCB rule, first proposed in the paper by Auer, Cesa-Bianchi, and Fischer [ACF02], has been extensively discussed in the literature both for one-step and for multistep lookahead [where it is called UCT (UCB applied to trees; see Kocsis and Szepesvari [KoS06])].†

Its justification is based on probabilistic analyses that relate to the multiarmed bandit problem, and is beyond our scope. Alternatives to the UCT formula have been suggested, and in fact in the AlphaZero program, the exploitation term has a different form from the one above, and depends on the depth of lookahead (see Silver et al. [SHS17]).

Sampling policies for MCTS with multistep lookahead are based on similar sampling ideas to the case of one-step lookahead. A simulated trajectory is run from a node $i$ of the lookahead tree that minimizes the sum $T_{i,n} + R_{i,n}$ of an exploitation index and an exploration index. There

† The paper [ACF02] refers to the rule given here as UCB1 and credits its motivation to the paper by Agrawal [Agr95].
are several schemes of this type, but the details are beyond our scope (see
the end-of-chapter references).

A major success has been the use of MCTS in two-player game con-
texts, such as the AlphaGo computer program (Silver et al. [SHM16]),
which performs better than the best humans in the game of Go. This pro-
gram integrates several of the techniques discussed in this chapter, and in
Chapters 3 and 4, including MCTS and rollout using a base policy that is
trained off-line using a deep neural network. We will discuss neural net-
work training techniques in Chapter 3. The AlphaZero program, which
has performed spectacularly well against humans and other programs in
the games of Go and chess (Silver et al. [SHS17]), bears some similarity
with AlphaGo, and critically relies on MCTS, but does not use rollout.

2.4.4 Randomized Policy Improvement by Monte Carlo Tree
Search

We have described rollout and MCTS as schemes for policy improvement:
start with a base policy, and compute an improved policy based on the
results of one-step lookahead or multistep lookahead followed by simulation
with the base policy. We have implicitly assumed that both the base policy
and the rollout policy are deterministic in the sense that they map each
state $x_k$ into a unique control $\tilde{\mu}_k(x_k)$ [cf. Eq. (2.24)]. In some contexts,
success has been achieved with randomized policies, which map a state $x_k$
to a probability distribution over the set of controls $U_k(x_k)$, rather than
mapping onto a single control. In particular, the AlphaGo and AlphaZe-
ro programs use MCTS to generate and use for training purposes random-
ized policies, which specify at each board position the probabilities with which
the various moves are selected.

A randomized policy can be used as a base policy in a rollout context
in exactly the same way as a deterministic policy: for a given state $x_k$, we
just generate sample trajectories and associated sample Q-factors, starting
from each leaf-state of the lookahead tree that is rooted at $x_k$. We then
average the corresponding Q-factor samples. However, the rollout/improved
policy, as we have described it, is a deterministic policy, i.e., it applies at $x_k$
the control $\tilde{\mu}_k(x_k)$ that is “best” according to the results of the rollout [cf.
Eq. (2.24)]. Still, however, if we wish to generate an improved policy that
is randomized, we can simply change the probabilities of different controls
in the direction of the deterministic rollout policy. This can be done by
increasing by some amount the probability of the “best” control $\tilde{\mu}_k(x_k)$
from its base policy level, while proportionally decreasing the probabilities
of the other controls.

The use of MCTS provides a related method to “improve” a ran-
domized policy. In the process of adaptive simulation that is used in MCTS,
we generate frequency counts of the different controls, i.e., the proportion
of rollout trajectories associated with each $u_k \in U_k(x_k)$. We can then ob-
tain the rollout randomized policy by moving the probabilities of the base policy in the direction suggested by the frequency counts, i.e., increase the probability of high-count controls and reduce the probability of the others. This type of policy improvement is reminiscent of gradient-type methods, and has been successful in some contexts; see the end-of-chapter references for such policy improvement implementations in AlphaGo, AlphaZero, and other application contexts.

2.4.5 Rollout Parallelization

Rollout and related algorithms such as approximate policy iteration (to be discussed in Chapter 4) can benefit greatly from the availability of parallel computation. There are several different ways to use parallelization to relieve the substantial computational burden of rollout. We provide an overview of some of the possibilities below, and defer a more detailed discussion of some of them for later.

(a) *Q-factor parallelization*: At state \( x_k \), rollout does a separate Q-factor calculation for each control \( u_k \in U_k(x_k) \). These calculations are decoupled and can be executed in parallel.

(b) *Monte Carlo parallelization*: Each of the Q-factor calculations in (a) above involves a Monte Carlo simulation when the problem is stochastic. Monte Carlo simulation requires many samples, which can be collected in parallel.

(c) *Multiagent parallelization*: In many problems, the control space has a special structure, whereby the control \( u_k \) consists of \( m \) components, \( u_k = (u_k^1, \ldots, u_k^m) \), with a separable control constraint \( u_k^\ell \in U_k^\ell(x_k) \), \( \ell = 1, \ldots, m \). Among others, this arises in applications involving distributed decision making by \( m \) agents, each choosing a single control component. Then the lookahead minimization at state \( x_k \) involves the computation of as many as \( n^m \) Q-factors, where \( n \) is the maximum number of elements of the sets \( U_k^\ell(x_k) \). This can be overwhelming even for moderate values of \( m \). In Section 2.6 and later, we will consider a problem reformulation, whereby control space complexity is traded off with state space complexity, by “unfolding” the control \( u_k \) into its \( m \) components, which are applied separately by the \( m \) agents. With such a scheme, the number of Q-factors computed at state \( x_k \) is reduced to \( nm \). Then in Section 2.6.1, we will discuss the possibility of some of the agents computing their Q-factor in parallel, based on suitable schemes of limited inter-agent communication.

(d) *Multiprocessor parallelization*: This relates to using rollout in conjunction with approximation in policy space and value space, as in the perpetual rollout schemes of Section 2.1.3. In particular, it is possible to partition the state space into subsets, and executing sepe-
arate rollout calculations on each subset. These calculations can be done in parallel. We will discuss the use of partitioning in the context of approximation architectures in Section 3.1.1, and we will consider distributed asynchronous versions of perpetual rollout and policy iteration in Chapter 4, in the context of infinite horizon problems. We should also add that state space partitioning can be used in obvious ways to parallelize exact finite horizon DP calculations, as well as in the context of infinite horizon methods such as value iteration (see Chapter 4).

2.5 ON-LINE ROLLOUT FOR DETERMINISTIC INFINITE-SPACES PROBLEMS - OPTIMIZATION HEURISTICS

We have considered so far discrete-spaces applications of rollout, where there is a finite number of relevant Q-factors at each state $x_k$, which are evaluated by simulation and compared by exhaustive comparison. When the control constraint set is infinite, to implement this approach the constraint set must first be discretized, which is often inconvenient and ineffective. In this section we will discuss an alternative approach for deterministic problems that can deal with an infinite number of controls and Q-factors at $x_k$ without discretization. The idea is to use a base heuristic that involves a continuous optimization, and to rely on a nonlinear programming method to solve the corresponding lookahead optimization problem.

To get a sense of the basic idea, consider the one-step lookahead rollout minimization

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

(2.28)

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

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

(2.29)

with $H_{k+1}(x_{k+1})$ being the cost of the base heuristic starting from state $x_{k+1}$ [cf. Eq. (2.16)]. Suppose that we have a differentiable closed-form expression for $H_{k+1}$, and the functions $g_k$ and $f_k$ are known and are differentiable with respect to $u_k$. Then the Q-factor $\tilde{Q}_k(x_k, u_k)$ of Eq. (2.29) is also differentiable with respect to $u_k$, and its minimization (2.28) may be addressed with one of the many gradient-based methods that are available for differentiable unconstrained and constrained optimization.

The preceding approach requires that the heuristic cost $H_{k+1}(x_{k+1})$ be available in closed form, which is highly restrictive, but this difficulty can be circumvented by using a base heuristic that is itself based on multistep optimization. In particular, suppose that $H_{k+1}(x_{k+1})$ is the optimal cost of some $(\ell - 1)$-stage deterministic optimal control problem that is related
to the original problem. Then the rollout algorithm (2.28)-(2.29) can be implemented by solving the \( \ell \)-stage deterministic optimal control problem, which seamlessly concatenates the first stage minimization over \( u_k \in U_k(x_k) \), seamlessly forms an \( \ell \)-stage continuous spaces optimal control/nonlinear programming problem that starts at state \( x_k \).

Figure 2.5.1 Schematic illustration of rollout for a deterministic problem with infinite control spaces. The base heuristic is to solve an \((\ell - 1)\)-stage deterministic optimal control problem, which together with the \( k \)th stage minimization over \( u_k \in U_k(x_k) \), seamlessly forms an \( \ell \)-stage continuous spaces optimal control/nonlinear programming problem that starts at state \( x_k \).

2.5.1 Model Predictive Control

We will consider a classical control problem, where the objective is to keep the state of a deterministic system close to the origin of the state space or close to a given trajectory. This problem has a long history, and has been addressed by a variety of methods. Starting in the late 50s and

\[ \text{Note, however, that for this to be possible, it is necessary to have a mathematical model of the system; a simulator is not sufficient. Another difficulty occurs when the control space is the union of a discrete set and a continuous set. Then it may be necessary to use some type of mixed integer programming technique to solve the \( \ell \)-stage problem. Alternatively, it may be possible to handle the discrete part by brute force enumeration, followed by continuous optimization. This last possibility is also amenable to parallelization.} \]
Figure 2.5.2 Illustration of constrained motion of a robot from point A to point B. There are state (position/velocity) constraints, and control (acceleration) constraints. When there are mobile obstacles, the state constraints may change unpredictably, necessitating on-line replanning.

early 60s, approaches based on state variable system representations and optimal control became popular. The linear-quadratic approach whereby the system is represented by a linear model, the cost is quadratic in the state and the control, and there are no state and control constraints was developed during this period, and is still used extensively. Unfortunately, however, linear-quadratic models are often not satisfactory. There are two main reasons for this:

(a) The system may be nonlinear, and it may be inappropriate to use for control purposes a model that is linearized around the desired point or trajectory.

(b) There may be control and/or state constraints, which are not handled adequately through quadratic penalty terms in the cost function. For example, the motion of a robot may be constrained by the presence of obstacles and hardware limitations (see Fig. 2.5.2). The solution obtained from a linear-quadric model may not be suitable for such a problem, because quadratic penalties treat constraints “softly” and may produce trajectories that violate the constraints.

These inadequacies of the linear-quadratic formulation have motivated a methodology, called model predictive control (MPC for short), which combines elements of several ideas that we have discussed so far,
such as multistep lookahead, rollout with infinite control spaces, and certainty equivalence. Aside from resolving the difficulty with infinitely many Q-factors at $x_k$, while dealing adequately with state and control constraints, MPC is well-suited for on-line replanning, like all rollout methods.

We will focus primarily on the most common form of MPC, where the system is either deterministic, or else it is stochastic, but it is replaced with a deterministic version by using typical values in place of the uncertain quantities, or state estimates in place of exact state values, similar to the certainty equivalent control approach. Moreover we will consider the case where the objective is to keep the state close to the origin (or more generally some point of interest, called the set point); this is called the regulation problem. Similar approaches have been developed for the problem of maintaining the state of a nonstationary system along a given state trajectory, and also, with appropriate modifications, to control problems involving disturbances. In particular, in some cases, the trajectory is treated like a sequence of set points, and the subsequently described algorithm is applied repeatedly.

We will consider a deterministic system

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

whose state $x_k$ and control $u_k$ are vectors that consist of a finite number of scalar components. The cost per stage is assumed nonnegative

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

(e.g., a quadratic cost). We impose state and control constraints

$$x_k \in X_k, \quad u_k \in U_k(x_k), \quad k = 0, 1, \ldots .$$

We also assume that the system can be kept at the origin at zero cost, i.e.,

$$f_k(0, \overline{u}_k) = 0, \quad g_k(0, \overline{u}_k) = 0 \quad \text{for some control } \overline{u}_k \in U_k(0).$$

For a given initial state $x_0 \in X_0$, we want to obtain a sequence $\{u_0, u_1, \ldots \}$ such that the states and controls of the system satisfy the state and control constraints with small total cost. In our subsequent development of MPC, we will assume for simplicity an infinite number of stages and no terminal cost, although our discussion can be modified to deal with finite horizon cases.

The MPC Algorithm

Let us describe the MPC algorithm for the deterministic problem just described. At the current state $x_k$: 
(a) MPC solves an $\ell$-step lookahead version of the problem, which requires that $x_{k+\ell} = 0$.

(b) If $\{\tilde{u}_k, \ldots, \tilde{u}_{k+\ell-1}\}$ is the optimal control sequence of this problem, MPC applies $\tilde{u}_k$ and discards the other controls $\tilde{u}_{k+1}, \ldots, \tilde{u}_{k+\ell-1}$.

(c) At the next stage, MPC repeats this process, once the next state $x_{k+1}$ is revealed.

In particular, at the typical stage $k$ and state $x_k \in X_k$, the MPC algorithm solves an $\ell$-stage optimal control problem involving the same cost function and the requirement $x_{k+\ell} = 0$. This is the problem

$$\min_{u_i, i = k, \ldots, k+\ell-1} \sum_{i=k}^{k+\ell-1} g_i(x_i, u_i),$$

subject to the system equation constraints

$$x_{i+1} = f_i(x_i, u_i), \quad i = k, \ldots, k + \ell - 1,$$

the state and control constraints

$$x_i \in X_i, \quad u_i \in U_i(x_i), \quad i = k, \ldots, k + \ell - 1,$$

and the terminal state constraint

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

Let $\{\tilde{u}_k, \tilde{u}_{k+1}, \ldots, \tilde{u}_{k+\ell-1}\}$ be a corresponding optimal control sequence. The MPC algorithm applies at stage $k$ the first component $\tilde{u}_k$ of this sequence, and discards the remaining components; see Fig. 2.5.3.$\dagger$

To guarantee that there is an integer $\ell$ such that the preceding MPC algorithm is feasible, we assume the following.

**Constrained Controllability Condition**

There exists an integer $\ell > 1$ such that for every initial state $x_k \in X_k$, we can find a sequence of controls $u_k, \ldots, u_{k+\ell-1}$ that drive to 0 the state $x_{k+\ell}$ of the system at time $k + \ell$, while satisfying all the intermediate state and control constraints

$$u_k \in U_k(x_k), \quad x_{k+1} \in X_{k+1}, \ldots,$$

$$x_{k+\ell-1} \in X_{k+\ell-1}, \quad u_{k+\ell-1} \in U_{k+\ell-1}(x_{k+\ell-1}).$$

$\dagger$ In the case, where we want the system to follow a given nominal trajectory, rather than stay close to the origin, we should modify the MPC optimization to impose as a terminal constraint that the state $x_{k+\ell}$ should be a point on the nominal trajectory (instead of $x_{k+\ell} = 0$). We should also change the cost function to reflect a penalty for deviating from the given trajectory.
Finding an integer \( \ell \) that satisfies the constrained controllability condition is an important issue. Generally the constrained controllability condition tends to be satisfied if the control constraints are not too stringent, and the state constraints do not allow a large deviation from the origin. In this case not only can we implement MPC, but also the resulting closed-loop system will tend to be stable; see the following discussion of stability. Note that the actual state trajectory produced by MPC may never reach the origin (see the subsequent Example 2.5.1). This is because we use only the first control \( \tilde{u}_k \) of the \( k \)th stage sequence \( \{\tilde{u}_k, \tilde{u}_{k+1}, \ldots, \tilde{u}_{k+\ell-1}\} \), which aims at \( x_{k+\ell} = 0 \). At the next stage \( k + 1 \) the control generated by MPC may be different than \( \tilde{u}_{k+1} \), because it will aim one step further to the terminal condition \( x_{k+\ell+1} = 0 \).

To make the connection of MPC with rollout, we note that the one-step lookahead function \( \tilde{J} \) implicitly used by MPC [cf. Eq. (2.30)] is the cost-to-go function of a certain base heuristic. This is the heuristic that drives to 0 the state after \( \ell - 1 \) stages (not \( \ell \) stages) and keeps the state at 0 thereafter, while observing the state and control constraints, and minimizing the associated \( (\ell - 1) \)-stages cost, in the spirit of our earlier discussion; cf. Fig. 2.5.1.

\[\dagger\] For a more detailed discussion of the constrained controllability condition, including ways to enforce it using reachability methods, we refer to the book [Ber19a] and to the MPC literature.
Sequential Improvement Property and Stability Analysis

It turns out that the base heuristic just described is sequentially improving, so MPC has a cost improvement property, of the type discussed in Section 2.4.1. To see this, let us denote by \( \hat{J}_k(x_k) \) the optimal cost of the \( \ell \)-stage problem solved by MPC when at a state \( x_k \in X_k \). Let also \( H_k(x_k) \) and \( H_{k+1}(x_{k+1}) \) be the optimal heuristic costs of the corresponding \( (\ell - 1) \)-stage optimization problems that start at \( x_k \) and \( x_{k+1} \), and drive the states \( x_{k+\ell-1} \) and \( x_{k+\ell} \), respectively, to 0. Thus, by the principle of optimality, we have the DP equation

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

Since having one less stage at our disposal to drive the state to 0 cannot decrease the optimal cost, we have

\[
\hat{J}_k(x_k) \leq H_k(x_k).
\]

By combining the preceding two relations, we obtain

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

which is the sequential improvement condition for the base heuristic [cf. Eq. (2.20)].

Often the primary objective in MPC, aside from fulfilling the state and control constraints, is to obtain a stable closed-loop system, i.e., a system that naturally tends to stay close to the origin. This is typically expressed adequately by the requirement of a finite cost over an infinite number of stages:

\[
\sum_{k=0}^{\infty} g_k(x_k, u_k) < \infty,
\]

where \( \{x_0, u_0, x_1, u_1, \ldots\} \) is the state and control sequence generated by MPC.

We will now show that the stability condition (2.32) is satisfied by the MPC algorithm. Indeed, from the sequential improvement condition (2.31), we have

\[
g_k(x_k, u_k) + H_{k+1}(x_{k+1}) \leq H_k(x_k), \quad k = 0, 1, \ldots
\]

Adding this relation for all \( k \) in a range \([0, K]\), where \( K = 0, 1, \ldots \), we obtain

\[
H_{K+1}(x_{K+1}) + \sum_{k=0}^{K} g_k(x_k, u_k) \leq H_0(x_0).
\]

\[†\] Note that the base heuristic is not sequentially consistent, as it fails to satisfy the definition given in Section 2.4.1 (see the subsequent Example 2.5.1).
Since (in view of the nonnegativity of the cost per stage) we have
\[ 0 \leq H_{K+1}(x_{K+1}), \]
it follows that
\[ \sum_{k=0}^{K} g_k(x_k, u_k) \leq H_0(x_0), \quad K = 0, 1, \ldots, \]
and taking the limit as \( K \to \infty \), we obtain
\[ \sum_{k=0}^{\infty} g_k(x_k, u_k) \leq H_0(x_0) < \infty, \quad (2.33) \]
[\( H_0(x_0) \) is finite because the transfer from \( x_0 \) to \( x_\ell = 0 \) is feasible by the constrained controllability condition]. We have thus verified the stability condition (2.32).

**Example 2.5.1**

Consider a scalar linear system and a quadratic cost
\[ x_{k+1} = x_k + u_k, \quad g_k(x_k, u_k) = x_k^2 + u_k^2, \]
where the state and control constraints are
\[ x_k \in X_k = \{ x \mid |x| \leq 1.5 \}, \quad u_k \in U_k(x_k) = \{ u \mid |u| \leq 1 \}. \]

We apply the MPC algorithm with \( \ell = 2 \). For this value of \( \ell \), the constrained controllability assumption is satisfied, since the 2-step sequence of controls
\[ u_0 = -\text{sgn}(x_0), \quad u_1 = -x_1 = -x_0 + \text{sgn}(x_0) \]
drives the state \( x_2 \) to 0, for any \( x_0 \) with \( |x_0| \leq 1.5 \).

At state \( x_k \in X_k \), MPC minimizes the two-stage cost
\[ x_k^2 + u_k^2 + (x_k + u_k)^2 + u_{k+1}^2, \]
subject to the control constraints
\[ |u_k| \leq 1, \quad |u_{k+1}| \leq 1, \]
and the state constraints
\[ |x_{k+1}| \leq 1.5, \quad x_{k+2} = x_k + u_k + u_{k+1} = 0. \]
This is a quadratic programming problem, which can be solved with available software, and in this case analytically, because of its simplicity. In particular, it can be verified that the minimization yields

\[
\tilde{u}_k = -\frac{2}{3}x_k, \quad \tilde{u}_{k+1} = -(x_k + \tilde{u}_k).
\]

Thus the MPC algorithm selects \(\tilde{u}_k = -\frac{2}{3}x_k\), which results in the closed-loop system

\[
x_{k+1} = \frac{1}{3}x_k, \quad k = 0, 1, \ldots.
\]

Note that while this closed-loop system is stable, its state is never driven to 0 if started from \(x_0 \neq 0\). Moreover, it is easily verified that the base heuristic is not sequentially consistent. For example, starting from \(x_k = 1\), the base heuristic generates the sequence

\[
\left\{ x_k = 1, u_k = -\frac{2}{3}x_k, x_{k+1} = \frac{1}{3}x_k, u_{k+1} = -\frac{1}{3}x_{k+2} = 0, u_{k+2} = 0, \ldots \right\},
\]

while starting from the next state \(x_{k+1} = \frac{1}{3}\) it generates the sequence

\[
\left\{ x_{k+1} = \frac{1}{3}, u_{k+1} = -\frac{2}{9}x_{k+2} = \frac{1}{9}x_{k+2} = -\frac{1}{9}x_{k+3} = 0, u_{k+3} = 0, u_{k+3} = 0, \ldots \right\},
\]

so the sequential consistency condition of Section 2.4.1 is violated.

Regarding the choice of the horizon length \(\ell\) for the MPC calculations, note that if the constrained controllability assumption is satisfied for some value of \(\ell\), it is also satisfied for all larger values of \(\ell\). This argues for a large value of \(\ell\). On the other hand, the optimal control problem solved at each stage by MPC becomes larger and hence more difficult as \(\ell\) increases. Thus, the horizon length is usually chosen on the basis of some experimentation: first ensure that \(\ell\) is large enough for the constrained controllability assumption to hold, and then by further experimentation to ensure overall satisfactory performance.

The MPC scheme that we have described is just the starting point for a broad methodology with many variations, which often relate to the suboptimal control methods that we have discussed so far in this chapter. For example, in the problem solved by MPC at each stage, instead of the requirement of driving the system state to 0 in \(\ell\) steps, one may use a large penalty for the state being nonzero after \(\ell\) steps. Then, the preceding stability analysis goes through, as long as the terminal penalty is chosen so that the sequential improvement condition (2.31) is satisfied and Eq. (2.33) can be shown. We can view the terminal penalty as a terminal cost function approximation in the context of rollout.

Another possibility is to generate terminal cost function values by simulation of some policy. In still another variant, instead of aiming to drive the state to 0 after \(\ell\) steps, one aims to reach a sufficiently small neighborhood of the origin, within which a stabilizing controller, designed by other methods, may be used.
2.6 MULTIAGENT ROLLOUT

We will now consider a special structure of the control space, whereby the control $u_k$ consists of $m$ components, $u_k = (u_{1k}, \ldots, u_{mk})$, with a separable control constraint structure $u_{\ell k} \in U_{\ell k}(x_k)$, $\ell = 1, \ldots, m$. Thus the control constraint set is the Cartesian product

$$U_k(x_k) = U_{1k}(x_k) \times \cdots \times U_{mk}(x_k). \quad (2.34)$$

We will view each component $u_{\ell k}$, $\ell = 1, \ldots, m$, as being chosen at stage $k$ by a separate “agent” (a decision making entity), from within a given set $U_{\ell k}(x_k)$. This brings up a connection with applications involving distributed decision making by multiple agents with a suitable structure of communication and coordination between the agents. Then the one-step lookahead minimization

$$\tilde{u}_k = \arg \min_{u_k \in U_k(x_k)} E \left\{ g_k(x_k, u_k, w_k) + J_{k+1, \pi} (f_k(x_k, u_k, w_k)) \right\}, \quad (2.35)$$

involves as many as $n^m$ Q-factors, where $n$ is the maximum number of elements of the sets $U_{\ell k}(x_k)$ [so that $n^m$ is an upper bound to the number of controls in $U_k(x_k)$, in view of its Cartesian product structure (2.34)]. Thus the rollout algorithm requires an exponential [order $O(n^m)$] number of base policy cost computations per stage, which can be overwhelming even for moderate values of $m$.

This potentially large computational overhead motivates us to construct an alternative more efficient rollout algorithm that still achieves the cost improvement property

$$J_{k, \hat{\pi}}(x_k) \leq J_{k, \pi}(x_k), \quad \forall \ x_k, k, \quad (2.36)$$

where $J_{k, \hat{\pi}}(x_k)$, $k = 0, \ldots, N$, is the cost-to-go of the rollout policy starting from state $x_k$. Indeed we will use the special multiagent constraint structure (2.34) to construct an algorithm that maintains the cost improvement property at smaller computational cost, namely requiring order $O(nm)$ base policy cost computations per stage.

A key idea here is that the computational requirements of the rollout one-step minimization (2.35) 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 $m$ individual component decisions, thereby reducing the complexity of the control space while increasing the complexity...
of the state space. The potential advantage is that the extra state space complexity does not affect the computational requirements of some RL algorithms, including rollout.

To this end, we introduce a modified but equivalent problem, involving one-at-a-time agent control selection. At the generic state $x_k$, 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 $x_{k+1} = f_k(x_k, u_k, w_k)$, we introduce artificial intermediate "states" $(x_k, u^1_k), (x_k, u^2_k), \ldots, (x_k, u^{m-1}_k)$, 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 to the next state $x_{k+1} = f_k(x_k, u_k, w_k)$ according to the system equation, while incurring cost $g_k(x_k, u_k, w_k)$; see Fig. 2.6.1.

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 $(\mu^1_k(x_k), \ldots, \mu^m_k(x_k))$ of the original problem, including a base policy in the context of rollout, is admissible for the reformulated problem, and has the same cost function for the original as well as the reformulated problem.†

The motivation for the reformulated problem is that the control space

† Note that formally the set of policies of the reformulated problem contains the set of policies of the original problem, but not reversely. Still, however, it can be seen that the optimal cost function of the reformulated and the original problems are equal, since the multi-transition structure of the reformulated problem cannot be exploited to reduce the cost function beyond what can be achieved with a single-transition structure.
is simplified at the expense of introducing \( m - 1 \) additional layers of states, and corresponding \( m - 1 \) cost-to-go functions

\[
J_k^1(x_k, u_k^1), J_k^2(x_k, u_k^1, u_k^2), \ldots, J_k^{m-1}(x_k, u_k^1, \ldots, u_k^{m-1}),
\]

in addition to \( J_k(x_k) \). The increase in size of the state space does not adversely affect the operation of rollout. Moreover, in a different context it can be dealt with by using function approximation, i.e., with the introduction of cost-to-go approximations

\[
\tilde{J}_k^1(x_k, u_k^1, r_k^1), \tilde{J}_k^2(x_k, u_k^1, u_k^2, r_k^2), \ldots, \tilde{J}_k^{m-1}(x_k, u_k^1, \ldots, u_k^{m-1}, r_k^{m-1}),
\]

in addition to \( \tilde{J}_k(x_k, r_k) \), where \( r_k, r_k^1, \ldots, r_k^{m-1} \) are parameters of corresponding approximation architectures such as neural networks; see Chapter 3.

**One-Component-at-a-Time Rollout**

Consider now the standard rollout algorithm applied to the reformulated problem of Fig. 2.6.1, with a given base policy \( \pi = \{\mu_0, \ldots, \mu_{N-1}\} \), which is also a policy of the original problem [so that \( \mu_k = (\mu_k^1, \ldots, \mu_k^m) \), with each \( \mu_k^\ell \), \( \ell = 1, \ldots, m \), being a function of just \( x_k \)]. The algorithm involves a minimization over only one control component at the states \( x_k \) and at the intermediate states

\[
(x_k, u_k^1), (x_k, u_k^1, u_k^2), \ldots, (x_k, u_k^1, \ldots, u_k^{m-1}).
\]

In particular, for each stage \( k \), the algorithm requires a sequence of \( m \) minimizations, once over each of the agent controls \( u_k^1, \ldots, u_k^m \), with the past controls determined by the rollout policy, and the future controls determined by the base policy. Assuming a maximum of \( n \) elements in the constraint sets \( U_k^\ell(x_k) \), the computation required at each stage \( k \) is of order \( O(n) \) for each of the “states”

\[
x_k, (x_k, u_k^1), \ldots, (x_k, u_k^1, \ldots, u_k^{m-1}),
\]

for a total of order \( O(nm) \) computation.

To elaborate, at \( (x_k, u_k^1, \ldots, u_k^{\ell-1}) \) with \( \ell \leq m \), and for each of the controls \( u_k^\ell \), we generate by simulation a number of system trajectories up to stage \( N \), with all future controls determined by the base policy. We average the costs of these trajectories, thereby obtaining the \( Q \)-factor corresponding to \( (x_k, u_k^1, \ldots, u_k^{\ell-1}, u_k^\ell) \). We then select the control \( u_k^\ell \) that corresponds to the minimal \( Q \)-factor, with the controls \( u_k^1, \ldots, u_k^{\ell-1} \) held fixed at the values computed earlier.

Prerequisite assumptions for the preceding algorithm to work in an on-line multiagent setting are:
Figure 2.6.2 Illustration of the two-spiders and two-flies problem. The spiders move along integer points of a line. The two flies stay still at some integer locations. The optimal policy is to move the two spiders towards different flies, the ones that are initially closest to them.

Multiagent rollout with the given base policy starts with spider 1 at location $n$, and calculates the two Q-factors of moving to locations $n-1$ and $n+1$, assuming that the remaining moves of the two spiders will be made using the go-towards-the-nearest-fly base policy. The Q-factor of going to $n-1$ is smallest because it saves in unnecessary moves of spider 1 towards fly 2, so spider 1 will move towards fly 1. The trajectory generated by multiagent rollout is to move spiders 1 and 2 towards flies 1 and 2, respectively, then spider 2 first captures fly 2, and then spider 1 captures fly 1.

(a) All agents have access to the current state $x_k$.
(b) There is an order in which agents compute and apply their local controls.
(c) There is intercommunication between agents, so agent $\ell$ knows the local controls $u_{k}^{1}, \ldots, u_{k}^{\ell-1}$ computed by the predecessor agents $1, \ldots, \ell-1$ in the given order.

Note that the rollout policy obtained from the reformulated problem may be different from the rollout policy obtained from the original problem. However, the former rollout algorithm is far more efficient than the latter in terms of required computation.

We provide two deterministic examples that illustrate the algorithm.

Example 2.6.1 (Spiders and Flies)

This is an extension of the spider-and-fly problem of Example 1.2.1. We have two spiders moving along integer locations on a straight line. There are also two flies, and for simplicity we will assume that the flies’ positions are fixed at some integer locations, although the problem is qualitatively similar when the flies move randomly. The spiders have the option of moving either left or right by one unit; see Fig. 2.6.2. The objective is to minimize the time to capture both flies. The problem has essentially a finite horizon since the spiders can force the capture of the flies within a known number of steps.

Here the optimal policy is to move the two spiders towards different flies, the ones that are initially closest to them (with ties broken arbitrarily). The minimal time to capture is the sum of the initial distances of the two spider-fly pairs.

Let us apply multiagent rollout with the base policy that directs each spider to move one unit towards the closest fly position (and in case of a tie, move towards the fly that lies to the right). The base policy is poor because it...
unnecessarily moves both spiders in the same direction, when in fact only one is needed to capture the fly. This limitation is due to the lack of coordination between the spiders: each acts selfishly, ignoring the presence of the other. We will see that rollout restores a significant degree of coordination between the spiders through an optimization that takes into account the long-term consequences of the spider moves.

According to the multiagent rollout mechanism, the spiders choose their moves one-at-a-time, optimizing over the two Q-factors corresponding to the right and left moves, while assuming that future moves will be chosen according to the base policy. Let us consider a stage, where the two flies are alive while the spiders are at different locations as in Fig. 2.6.2. Then the rollout algorithm will start with spider 1 and calculate two Q-factors corresponding to the right and left moves, while using the base heuristic to obtain the next move of spider 2, and the remaining moves of the two spiders. Depending on the values of the two Q-factors, spider 1 will move to the right or to the left, and it can be seen that it will choose to move away from spider 2 even if doing so increases its distance to its closest fly contrary to what the base heuristic will do; see Fig. 2.6.2. Then spider 2 will act similarly and the process will continue. Intuitively, spider 1 moves away from spider 2 and fly 2, because it recognizes that spider 2 will capture earlier fly 2, so it might as well move towards the other fly.

Thus the multiagent rollout algorithm induces implicit move coordination, i.e., each spider moves in a way that takes into account future moves of the other spider. In fact it can be verified that the algorithm will produce an optimal sequence of moves starting from any initial state. It can also be seen that ordinary rollout (both flies move at once) will also produce an optimal move sequence.

This example illustrates how a poor base heuristic can produce an excellent rollout solution, something that can be observed in many other problems. Intuitively, the key fact is that rollout relies on “intrinsic vision” for better performance, in the sense in can benefit from control calculations that reach far into future stages.

**Example 2.6.2 (Multi-Vehicle Routing)**

Consider $n$ vehicles that move along the arcs of a given graph. Some of the nodes of the graph include a task to be performed by the vehicles. Each task will be performed only once, immediately after a vehicle reaches the corresponding node for the first time. We assume a horizon that is large enough to allow every task to be performed. The problem is to find a route for each vehicle so that, roughly speaking, the tasks are collectively performed by the vehicles in minimum time. To express this objective, we assume that for each stage there is a cost of one unit for each task that is pending (was never performed) at the end of stage, so the total cost is the number of stages that a task is pending, summed over the set of tasks.

For a large number of vehicles and a complicated graph, this is a non-trivial combinatorial problem. It can be approached by DP, like any discrete deterministic optimization problem. In particular, we can view as state at
Figure 2.6.3 An instance of the vehicle routing problem of Example 2.6.2, and the one-vehicle-at-a-time approach. The two vehicles aim to collectively perform the two tasks as fast as possible. Here, we should avoid sending both vehicles to node 4, towards the task at node 7; sending only vehicle 2 towards that task, while sending vehicle 1 towards the task at node 9 is clearly optimal. However, the base heuristic has "limited vision" and does not perceive this. By contrast the rollout algorithm looks beyond the first move and avoids this inefficiency: it examines both moves of vehicle 1 to nodes 3 and 4, uses the base heuristic to explore the corresponding trajectories to the end of the horizon, and discovers that vehicle 2 can reach quickly node 7, and that it is best to send vehicle 1 towards node 9.

In particular, the one-vehicle-at-a-time rollout algorithm will operate as follows: given the starting position pair (1, 2) of the vehicles and the current pending tasks at nodes 7 and 9, we first consider the two possible moves of vehicle 1 (to nodes 3 and 4). We then select the move of vehicle 2 according to the base heuristic, which is to move to node 4 along the path to the closest pending task. Then select the next move of vehicle 1 on the path to the closest pending task, then do the same for vehicle 2, etc. The Q-factors for vehicle 1 are equal to 5 (for going to node 3) and 7 (for going to node 4), so the rollout algorithm moves vehicle 1 to node 3. The rollout algorithm then repeats with vehicle 2 [with the first move of vehicle 1 fixed to 1 → 3], thus selecting its move to node 4. It then continues with vehicle 1 at the next stage, etc, until all the tasks have been performed.

Eventually the rollout finds the optimal solution (move vehicle 1 to node 9 in three moves and move vehicle 2 to node 7 in two moves), which has a total cost of 5. By contrast it can be seen that the base heuristic at the initial state will initially move both vehicles to node 4 (towards the closest pending task), and generate a trajectory that moves vehicle 1 along the path 1 → 4 → 7 and vehicle 2 along the path 4 → 1 → 3 → 6 → 9, while incurring a total cost of 7.

At a given stage the n-tuple of current positions of the vehicles together with the list of pending tasks. Unfortunately, however, the number of these states can be enormous (it increases exponentially with the number of nodes and the number of vehicles). Moreover the number of joint move choices by the vehicles also increases exponentially with the number of vehicles.

We are thus motivated to use a one-vehicle-at-a-time rollout approach. In particular, we consider a base heuristic that moves each vehicle towards the pending task that is closest (in terms of number of moves needed to reach
The vehicles make their selection one-at-a-time in the order 1, . . . , n, and take into account the tasks that have already been performed; see the illustration of Fig. 2.6.3. In contrast to the all-vehicles-at-once rollout, the one-vehicle-at-a-time rollout considers a polynomial (in $n$) number of moves and corresponding shortest path problems at each stage. In the example of Fig. 2.6.3, the one-vehicle-at-a-time rollout finds the optimal solution, while the base heuristic starting from the initial state does not.

The Cost Improvement Property

Generally, it is unclear how the two rollout policies perform relative to each other in terms of attained cost.† On the other hand, both rollout policies perform no worse than the base policy, since the performance of the base policy is identical for both the reformulated problem and for the original problem. This cost improvement property can also be shown analytically by induction, by modifying the standard rollout cost improvement proof; cf. Section 2.4.

**Proposition 2.6.1: (Cost Improvement for Multiagent Rollout)** The rollout policy $\hat{\pi} = \{\hat{\mu}_0, \ldots, \hat{\mu}_{N-1}\}$ obtained by multiagent rollout satisfies

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

(2.37)

where $\pi$ is the base policy.

**Proof:** We will show the inequality (2.37) by induction, but for simplicity, we will give the proof for the case of just two agents, i.e., $m = 2$. Clearly the inequality holds for $k = N$, since $J_{N,\hat{\pi}} = J_{N,\pi} = g_N$. Assuming that it holds for index $k + 1$, we have for all $x_k$,

$$J_{k,\hat{\pi}}(x_k) = E\left\{g_k(x_k, \hat{\mu}_1^k(x_k), \hat{\mu}_2^k(x_k), w_k)\right\}$$

† For an example where the standard rollout algorithm works better, consider a single-stage problem, where the objective is simply to minimize $g_0(u_0^1, \ldots, u_0^m)$. Let $\bar{u}_0 = (\bar{u}_0^0, \ldots, \bar{u}_0^m)$ be the control applied by the base policy, and assume that $\bar{u}_0$ is not optimal. Suppose that starting at $\bar{u}_0$, the cost cannot be improved by varying any single control component. Then the one-component-at-a-time rollout algorithm stays at the suboptimal $\bar{u}_0$, while the standard rollout algorithm finds an optimal control. This is the classical type of situation where the coordinate descent method fails to make progress in the context of $m$-dimensional optimization. In conclusion, for one-stage problems, the standard rollout algorithm will perform as well or better that the one-component-at-a-time rollout algorithm. However, one can construct two-stage examples where the reverse is true.
\begin{align*}
&+ J_{k+1, \pi} \left( f_k(x_k, \mu_{1}^k(x_k), \mu_{2}^k(x_k), w_k) \right) \\
\leq E \{ g_k(x_k, \mu_{1}^k(x_k), \mu_{2}^k(x_k), w_k) \\
&+ J_{k+1, \pi} \left( f_k(x_k, \mu_{1}^k(x_k), \mu_{2}^k(x_k), w_k) \right) \} \\
= \min_{u_{2}^k \in U_{2}^k(x_k)} E \{ g_k(x_k, \mu_{1}^k(x_k), u_{2}^k, w_k) \\
&+ J_{k+1, \pi} \left( f_k(x_k, \mu_{1}^k(x_k), u_{2}^k, w_k) \right) \} \\
\leq E \{ g_k(x_k, \mu_{1}^k(x_k), \mu_{2}^k(x_k), w_k) \\
&+ J_{k+1, \pi} \left( f_k(x_k, \mu_{1}^k(x_k), \mu_{2}^k(x_k), w_k) \right) \} \\
= \min_{u_{1}^k \in U_{1}^k(x_k)} E \{ g_k(x_k, u_{1}^k, \mu_{2}^k(x_k), w_k) \\
&+ J_{k+1, \pi} \left( f_k(x_k, u_{1}^k, \mu_{2}^k(x_k), w_k) \right) \} \\
\leq E \{ g_k(x_k, \mu_{1}^k(x_k), \mu_{2}^k(x_k), w_k) \\
&+ J_{k+1, \pi} \left( f_k(x_k, \mu_{1}^k(x_k), \mu_{2}^k(x_k), w_k) \right) \} \\
= J_{k, \pi}(x_k),
\end{align*}

where:

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

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

(c) The second equality holds by the definition of the rollout algorithm as it pertains to agent 2.

(d) The third equality holds by the definition of the rollout algorithm as it pertains to agent 1.

(e) The fourth equality is the DP equation for the base policy \( \pi \).

The induction proof of the cost improvement property (2.37) is thus complete for the case \( m = 2 \). The proof for an arbitrary number of agents \( m \) is entirely similar. \textbf{Q.E.D.}

**Multiagent Rollout Variants**

It is worth noting a few variants of the one-component-at-a-time rollout algorithm.

(a) Instead of selecting the agent controls in a fixed order, it is possible to change the order at each stage \( k \) (the preceding cost improvement
Sec. 2.6 Multiagent Rollout

proof goes through again by induction). In fact it is possible to optimize over multiple orders at the same stage.

(b) We may use rollout variants involving multistep lookahead, truncated rollout, and terminal cost function approximation, as described earlier. Of course, in such variants the cost improvement property need not hold.

(c) The multiagent rollout algorithm can be simply modified to apply to infinite horizon problems. In this context, we may also consider policy iteration methods, which may be viewed as repeated rollout. These methods may involve one-component-at-a-time policy improvement, and possibly approximation in value and in policy space of intermediate-generated policies (see Chapters 3 and 4, and the RL book [Ber19a], Section 5.7.3).

Moreover, when the problem is deterministic there are additional possible variants of the multiagent rollout algorithm. In particular, for deterministic problems, one may use a more general base policy, i.e., a heuristic that is not defined by an underlying policy; cf. Section 2.3.1. In this case, if the sequential improvement assumption for the modified problem of Fig. 2.6.1 is not satisfied, then the cost improvement property may not hold. However, cost improvement may be restored by introducing fortification, as discussed in Section 2.3.2. We will next discuss a related possibility in a context where distributed computation is attractive to deal with the case of a large number of agents.

2.6.1 Multiagent Parallelization

We will now consider multiagent rollout algorithms that are distributed and asynchronous in the sense that the agents may compute their rollout controls in parallel or in some irregular order rather than in sequence. An example of such an algorithm is obtained when at a given stage, agent \( \ell \) computes the rollout control \( \tilde{u}_k^\ell \) before knowing the rollout controls of some of the agents 1, \ldots, \( \ell - 1 \), and uses the controls \( \mu^{k}_{1}(x_k), \ldots, \mu^{k-1}_{\ell}(x_k) \) of the base policy in their place. Note that there is two-fold computational savings from such an algorithm, first by reducing the number of Q-factors to compute at state \( x_k \) from \( nm \) to \( nm \), and then by reducing the time to compute some of these Q-factors in parallel.

Unfortunately, however, while such an algorithm is likely to work well for many problems, it may not possess the cost improvement property. To illustrate this we construct a simple example involving a single state, two agents, and two controls per agent, where the second agent does not take into account the control applied by the first agent, and as a result the rollout policy performs worse than the base policy.
Example 2.6.3 (Cost Deterioration in the Absence of Adequate Agent Coordination)

Consider a deterministic problem with a single state and two agents \((m = 2)\). Thus the state does not change and the costs of different stages are decoupled (the problem is essentially static). Each of the two agents has two controls: \(u^1_k \in \{0, 1\}\) and \(u^2_k \in \{0, 1\}\). The cost per stage \(g_k\) is

\[
g_k = \begin{cases} 
0 & \text{if } u^1_k \neq u^2_k, \\
1 & \text{if } u^1_k = u^2_k = 0, \\
2 & \text{if } u^1_k = u^2_k = 1. 
\end{cases}
\]

Suppose that the base policy applies \(u^1_k = u^2_k = 0\). Then it can be seen that when executing rollout, the first agent applies \(u^1_k = 1\), and in the absence of knowledge of this choice, the second agent also applies \(u^2_k = 1\) (thinking that the first agent will use the base policy control \(u^1_k = 0\)). Thus the cost of the rollout policy is 2 per stage, while the cost of the base policy is 1 per stage. By contrast the rollout algorithm that takes into account the first agent’s control when selecting the second agent’s control applies \(u^1_k = 1\) and \(u^2_k = 0\), thus resulting in a rollout policy with the optimal cost of 0 per stage.

The difficulty here is inadequate coordination between the two agents. In particular, each agent uses rollout to compute the local control, each thinking that the other will use the base policy control. If instead the two agents were to coordinate their control choices, they would have applied an optimal policy.

The simplicity of the preceding example raises serious questions as to whether the cost improvement property \((2.37)\) can be easily maintained by a distributed rollout algorithm where the agents do not know the controls applied by the preceding agents in the given order of local control selection, and use instead the controls of the base policy. Still, however, such an algorithm is computationally attractive in view of its potential for efficient distributed implementation, and may be worth considering in a practical setting. A noteworthy property of this algorithm is that if the base policy is optimal, the same is true of the rollout policy. This suggests that if the base policy is nearly optimal, the same is true of the rollout policy.

We may also speculate that if the agents are naturally “weakly coupled” in the sense that their choice of control has little impact on the desirability of controls of other agents, then a more flexible inter-agent communication pattern may be sufficient for cost improvement. In particular, one may divide the agents into “coupled” groups, and require coordination of control selection only within each group, while the computation of different groups may proceed in parallel. For example, in applications where the agents’ locations are distributed within some geographical area, it may make sense to form agent groups on the basis of geographic proximity, i.e., one may require that agents that are geographically near each other (and hence are more coupled) coordinate their control selections, while agents
Figure 2.6.4 Illustration of the 2-dimensional spiders-and-fly problem. The state is the vector of distances between spiders and fly. At each time period, each spider moves to a neighboring location or stays where it is. The spiders make moves with perfect knowledge of the locations or each other and of the fly. The fly moves randomly, regardless of the position of the spiders.

that are geographically far apart (and hence are less coupled) forego any coordination.

Example 2.6.4 (Spiders-and-Fly)

Let us consider a multiple spider version of the spider-and-fly problem of Example 1.2.1. Here there are \( m \) spiders and one fly moving on a 2-dimensional grid. During each time period the fly moves to a some other position according to a given state-dependent probability distribution. Each spider learns the current state (the vector of spiders and fly locations) at the beginning of each time period, and either moves to a neighboring location or stays where it is. Thus each spider \( i \) has a many as five choices at each time period. The control vector is \( u = (u^1, \ldots, u^m) \), where \( u^i \) is the choice of the \( i \)th spider, so there are about \( 5^m \) possible values of \( u \). However, if we view this as a multiagent problem, as per the reformulation of Fig. 2.6.1, the size of the control space is reduced to \( \leq 5 \) moves per spider.

To apply multiagent rollout, we need a base policy. A simple possibility is to use the policy that directs each spider to move on the path of minimum distance to the current fly position. According to the multiagent rollout mechanism, the spiders choose their moves in a given order, taking into account the current state, and assuming that future moves will be chosen according to the base policy. This is a tractable computation, particularly if the rollout with
the base policy is truncated after some stage, and the cost of the remaining stages is approximated using a certainty equivalence approximation in order to reduce the cost of the Monte Carlo simulation. The problem can be made more complicated by introducing terrain obstacles or multiple flies.

We may also consider distributed rollout schemes, where the coordination between the flies is imperfect, as per our earlier discussion. In an example of such a scheme, the spiders that are close to the fly may coordinate their moves, while spiders that are far from the fly simply move “towards” the fly without any coordination with other flies.

We finally note that for deterministic multiagent problems with uncoordinated agent computations, we can restore the cost improvement property through the use of fortification. This possibility was described at the end of Section 2.3.2. Indeed, by maintaining a tentative best trajectory, the resulting fortified multiagent rollout algorithm will produce a solution that is no worse than the one produced by the base heuristic at the initial state, provided that the latter solution is taken to be the initial best tentative trajectory.

2.7 CONSTRAINED ROLLOUT FOR DETERMINISTIC OPTIMIZATION

In this section we consider an extension of the rollout algorithm that applies to constrained deterministic dynamic programming, including challenging combinatorial optimization problems. Many methods are available of such problems, such as local search and random search methods. When the problem is formulated so that a feasible solution $u$ consists of multiple components, i.e., has the form $u = \{u_0, \ldots, u_{N-1}\}$ for some $N$, rollout differs from these methods in an important way. It builds up a solution component-by-component, first $u_0$, then $u_1$, etc, obtaining a complete solution only upon generation of the last component $u_{N-1}$. By contrast, local and random search methods generate sequences of complete solutions $u_t$, each consisting of all $N$ components. This suggests that rollout may be better suited for problems that have a naturally sequential character.

We will develop a constrained rollout algorithm that relies on a base heuristic (a suboptimal policy), similar to Section 2.3.1. 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.

We will also discuss multiagent problems, where the control at each stage consists of multiple components (one per agent), which are coupled either through the cost function or the constraints or both. Similar to Section 2.6, we will show that the cost improvement property is maintained with an alternative implementation that has greatly reduced computational requirements, and makes possible the use of rollout in problems
with many agents. We will demonstrate this alternative algorithm in Section 2.8, where we will apply it to some problems that involve both a spatial and a temporal structure.

We consider a deterministic optimal control problem with the state $x_k$ taking values in some (possibly infinite) set and the control $u_k$ taking values in some finite set, and $f_k$ is some function. 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. We distinguish a complete trajectory from a partial trajectory, which is defined to be a subset of a complete trajectory, consisting of a subsequence of time-contiguous states and controls.

Our problem is stated succinctly as

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

where $G$ is a given real-valued cost function and $C$ is a given constraint set of trajectories.†

An example is the standard problem formulation for deterministic DP: an additive cost

$$G(x_0, u_0, x_1, u_1, \ldots, u_{N-1}, x_N) = g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k),$$

where the controls satisfy the time-uncoupled constraints

$$u_k \in U_k(x_k), \quad k = 0, 1, \ldots, N - 1,$$

(so here $C$ is the set of trajectories that are generated by the system equation with controls satisfying the above constraints). Our aim, however, is to use rollout to address problems involving far more complicated constraints, as well as the multiagent control structures where $u_k$ consists of multiple components; cf. Section 2.6.

† Actually it is not essential that we know the form of the function $G$. For the constrained rollout algorithms of this section, it is sufficient to have access to a human or software expert that can compare any two trajectories $T_1$ and $T_2$, without assigning numerical values to them. It is essential, however, that the expert’s rankings should have a transitivity property: if $T$ is ranked better than $T'$ and $T'$ is ranked better than $T''$, then $T$ is ranked better than $T''$. Of course, the expert should also be able to determine whether a given trajectory $T$ satisfies the constraint $T \in C$. 

Trajectory constraints can arise in a number of ways. For example there may be coupling of the controls of different stages such as

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

(2.43)

where \( g^m_k, k = 0, 1, \ldots, N, \) and \( b^m, m = 1, \ldots, M, \) are given functions and scalars, respectively.

For another constrained DP problem with a multiagent character, consider the multi-vehicle routing Example 2.6.2, and assume that some of the vehicles have capacity constraints, and cannot perform more than a maximum number of tasks, and/or some of the tasks are constrained to be performed within specified time windows. This is a far more complicated problem, although it does admit suboptimal solution with single vehicle heuristics that have a shortest path character (see Example 2.7.1 later in this section).

Generally, constrained DP problems can be transformed to unconstrained DP problems. The idea is to redefine the state at stage \( k \) to be the partial trajectory

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

which evolves according to a redefined system equation:

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

(2.44)

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.

Unfortunately, with the DP reformulation just described, the exact solution of the problem is typically impractical because the associated computation can be overwhelming. It is much greater than the computation for the corresponding additive cost/time-uncoupled control constraints problem (2.41)-(2.42), where the constraint \( T \in C \) is absent. Experience with constrained DP problems suggests that the use of an approximate solution approach is essentially unavoidable. This is the motivation for the methodology of this section. However, the DP formulation based on the redefined system (2.44) provides guidance for structuring the constrained rollout that we describe next.

**Using a Base Heuristic for Constrained Rollout**

We will now describe our constrained rollout algorithm. We assume the availability of a base heuristic, which for any given partial trajectory

\[ y_k = (x_0, u_0, x_1, \ldots, u_{k-1}, x_k), \]
Sec. 2.7 Constrained Rollout for Deterministic Optimization

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 for every \( t = k, \ldots, N-1 \) the system equation
\[ x_{t+1} = f_t(x_t, u_t). \]

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 \( T_k(y_k, u_k) \):
\[ T_k(y_k, u_k) = (y_k, u_k, R(y_{k+1})). \quad (2.45) \]

Note that the partial trajectory \( R(y_{k+1}) \) produced by the base heuristic depends on the entire partial trajectory \( y_{k+1} \).

A complete trajectory \( T_k(y_k, u_k) \) of the form (2.45) is generally feasible for only a subset of controls \( u_k \), which we denote by \( U_k(y_k) \):
\[ U_k(y_k) = \{ u_k \mid T_k(y_k, u_k) \in C \}. \quad (2.46) \]

Our rollout algorithm starts from a given initial state \( \tilde{y}_0 = \tilde{x}_0 \), and generates successive partial trajectories \( \tilde{y}_1, \ldots, \tilde{y}_N \), of the form
\[ \tilde{y}_{k+1} = (\tilde{y}_k, \tilde{u}_k, f_k(\tilde{x}_k, \tilde{u}_k)), \quad k = 0, \ldots, N - 1, \quad (2.47) \]
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 \( H(T_k(\tilde{y}_k, u_k)) \) over all \( u_k \) for which \( T_k(\tilde{y}_k, u_k) \) is feasible. Thus at stage \( k \), the algorithm forms the set \( U_k(\tilde{y}_k) \) and selects from \( U_k(\tilde{y}_k) \) a control \( \tilde{u}_k \) that minimizes the cost of the complete trajectory \( T_k(\tilde{y}_k, \tilde{u}_k) \):
\[ \tilde{u}_k \in \arg\min_{u_k \in U_k(\tilde{y}_k)} G(T_k(\tilde{y}_k, u_k)); \quad (2.48) \]

see Fig. 2.7.1. 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.,
\[ G(\tilde{y}_N) \leq G(R(\tilde{y}_0)). \quad (2.49) \]
Figure 2.7.1. Illustration of the rollout algorithm. At stage $k$, and given the current partial trajectory $\tilde{y}_k = (\tilde{x}_0, \tilde{u}_0, \tilde{x}_1, \ldots, \tilde{u}_{k-1}, \tilde{x}_k)$, which starts at $\tilde{x}_0$ and ends at $\tilde{x}_k$, it considers all possible next states $x_{k+1} = f_k(\tilde{x}_k, \tilde{u}_k)$, and runs the base heuristic starting at $y_{k+1} = (\tilde{y}_k, \tilde{u}_k, x_{k+1})$. It then:

(a) Finds $\tilde{u}_k$, the control that minimizes over all $u_k$ the cost $G(T_k(\tilde{y}_k, u_k))$ over all $u_k$ that yield a feasible complete trajectory $T_k(\tilde{y}_k, u_k) \in C$.

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

**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

$$\tilde{y}_k = (\tilde{x}_0, \tilde{u}_0, \tilde{x}_1, \ldots, \tilde{u}_{k-1}, \tilde{x}_k) \quad (2.50)$$

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

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

that is consistent with feasibility [cf. Eq. (2.46)], and chooses a control $\tilde{u}_k \in U_k(\tilde{y}_k)$ according to the minimization

$$\tilde{u}_k \in \arg \min_{u_k \in U_k(\tilde{y}_k)} G(T_k(\tilde{y}_k, u_k)), \quad (2.51)$$
[cf. Eq. (2.45)]. Finally, the algorithm sets

\[
\begin{align*}
  \tilde{x}_{k+1} &= f_k(\tilde{x}_k, \tilde{u}_k), \\
  \tilde{y}_{k+1} &= (\tilde{y}_k, \tilde{u}_k, \tilde{x}_{k+1})
\end{align*}
\]  (2.52)

[cf. Eq. (2.47)], thus obtaining the partial trajectory 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 initially our assumptions will guarantee that the constraint set \(U_k(\tilde{y}_k)\) of the problem (2.48) 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 or minimax problems, since checking feasibility of a complete trajectory is typically difficult in stochastic and minimax contexts.

**The Cost Improvement Property**

We will now introduce sequential consistency and sequential improvement conditions guaranteeing that the control set \(U_k(\tilde{y}_k)\) in the minimization (2.51) 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

\[
G(T_{k+1}(\tilde{y}_{k+1}, \tilde{u}_{k+1})) \leq G(T_k(\tilde{y}_k, \tilde{u}_k)), \quad k = 0, 1, \ldots, N - 1.
\]

Since

\[
\tilde{y}_N = T_{N-1}(\tilde{y}_{N-1}, \tilde{u}_{N-1}), \quad R(\tilde{y}_0) = T_0(\tilde{y}_0, \tilde{u}_0),
\]

it will then follow that the cost improvement condition \(G(\tilde{y}_N) \leq G(R(\tilde{y}_0))\) [cf. Eq. (2.49)] holds.
Definition 2.7.1: We say that the base heuristic is sequentially consistent if whenever it generates a partial trajectory 
\[ (x_k, u_k, x_{k+1}, u_{k+1}, \ldots, u_{N-1}, x_N), \]
starting from a partial trajectory \( y_k \), it also generates the partial trajectory 
\[ (x_{k+1}, u_{k+1}, x_{k+2}, u_{k+2}, \ldots, u_{N-1}, x_N), \]
starting from the partial trajectory \( y_{k+1} = (y_k, u_k, x_{k+1}) \).

As we have noted in the context of unconstrained rollout, greedy heuristics tend to be sequentially consistent. Also any policy [a sequence of feedback control functions \( \mu_k(y_k), k = 0, 1, \ldots, N-1 \)] for the DP problem of minimizing the terminal cost \( G(y_N) \) subject to the system equation
\[ y_{k+1} = (y_k, u_k, f_k(x_k, u_k)) \]
and the feasibility constraint \( y_N \in C \) [cf. Eq. (2.44)] can be seen to be sequentially consistent.

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 2.2: We say that the base heuristic is sequentially improving if for every \( k \) 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
\[ G(y_k \cup R(y_k)) \geq \min_{u_k \in U_k(y_k)} G(T_k(y_k, u_k)). \] (2.53)

Our main result is contained in the following proposition.

Proposition 2.7.1: (Cost Improvement for Constrained Rollout) Assume that the base heuristic is sequentially improving and generates a feasible complete trajectory starting from the initial state \( \hat{y}_0 = \hat{x}_0 \), i.e., \( R(\hat{y}_0) \in C \). Then for each \( k \), the set \( U_k(\hat{y}_k) \) is nonempty, and we have
Sec. 2.7 Constrained Rollout for Deterministic Optimization

\[
G(R(\tilde{y}_0)) \geq G(T_0(\tilde{y}_0, \tilde{u}_0)) \\
\geq G(T_1(\tilde{y}_1, \tilde{u}_1)) \\
\geq \cdots \\
\geq G(T_{N-1}(\tilde{y}_{N-1}, \tilde{u}_{N-1})) \\
= G(\tilde{y}_N),
\]

where

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

cf. Eq. (2.45). 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:** Let the complete trajectory generated by the base heuristic starting from \( \tilde{y}_0 \) have the form

\[
R(\tilde{y}_0) = (\tilde{x}_0, u'_0, x'_1, u'_1, \ldots, u'_{N-1}, x'_N),
\]

and note that since \( R(\tilde{y}_0) \in C \) by assumption, we have \( u'_0 \in U_0(\tilde{y}_0) \). Thus, the set \( U_0(\tilde{y}_0) \) is nonempty. Also, we have

\[
G(R(\tilde{y}_0)) \geq G(T_0(\tilde{y}_0, \tilde{u}_0)),
\]

by the sequential improvement assumption [cf. Eq. (2.53)].

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 \( T_0(\tilde{y}_0, \tilde{u}_0) \). In particular, let \( T_0(\tilde{y}_0, \tilde{u}_0) \) have the form

\[
T_0(\tilde{y}_0, \tilde{u}_0) = (\tilde{x}_0, \tilde{u}_0, \tilde{x}_1, u'_1, x'_2, \ldots, u'_{N-1}, x'_N).
\]

Since \( T_0(\tilde{y}_0, \tilde{u}_0) \) is feasible, we have \( u'_1 \in U_1(\tilde{y}_1) \) by the definition of sequential improvement, so that \( U_1(\tilde{y}_1) \) is nonempty. Also, we have

\[
T_0(\tilde{y}_0, \tilde{u}_0) = \tilde{y}_1 \cup R(\tilde{y}_1),
\]

so that

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

by the definition of sequential improvement. 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 2.7.1 establishes the fundamental cost improvement property for constrained rollout under the sequential improvement condition. On the other hand it is easy to construct examples where the sequential improvement condition (2.53) 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 (see [Ber19a], Example 2.4.2).

Note that 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 \( \{ T_k(y_k, u_k) \mid u_k \in U_k(y_k) \} \).† For another case where the base heuristic is sequentially improving, let the constraint set \( C \) consist of all trajectories \( T = (x_0, u_0, x_1, u_1, \ldots, u_{N-1}, x_N) \) such that

\[
 g_N^m(x_N) + \sum_{k=0}^{N-1} g_k^m(x_k, u_k) \leq b^m, \quad m = 1, \ldots, M, \tag{2.54}
\]

[cf. Eq. (2.43)]. Assume that the partial trajectory \( R(y_k) \) generated by the base heuristic depends only on the last state component \( x_k \) of \( y_k \), so we can write \( R(x_k) \) in place of \( R(y_k) \). Let \( \tilde{C}^m(x_k) \) be the value of the \( m \)th constraint function corresponding to the partial trajectory \( R(x_k) = (x_k, u_k, x_{k+1}, u_{k+1}, \ldots, u_{N-1}, x_N) \), generated by the base heuristic starting from \( x_k \), i.e.,

\[
 \tilde{C}^m(x_k) = g_N^m(x_N) + \sum_{t=k}^{N-1} g_t^m(x_t, u_t), \quad m = 1, \ldots, M.
\]

Then, it can be seen that the base heuristic is sequentially improving if in addition to Eq. (2.53), the partial trajectory \( R(x_k) \) satisfies

\[
 g_k^m(x_k, u_k) + \tilde{C}^m(x_{k+1}) \leq \tilde{C}^m(x_k), \quad m = 1, \ldots, M.
\]

The reason is that if a complete trajectory of the form

\[
 (x_0, u_0, x_1, \ldots, u_{k-1}, x_k) \cup R(x_k)
\]

† Indeed the sequential consistency assumption may be replaced by the weaker condition that the algorithm satisfies

\[
 \tilde{y}_k \cup \tilde{R}(\tilde{y}_k) \in \{ T_k(\tilde{y}_k, u_k) \mid u_k \in U_k(\tilde{y}_k) \}.
\]

In fact this is the idea behind the fortified variant of the algorithm to be discussed shortly: essentially, the complete trajectory \( \tilde{y}_k \cup \tilde{R}(\tilde{y}_k) \) is artificially added to the set \( \{ T_k(\tilde{y}_k, u_k) \mid u_k \in U_k(\tilde{y}_k) \} \).
Sec. 2.7 Constrained Rollout for Deterministic Optimization

belongs to \( C \), then we have

\[
\sum_{t=0}^{k-1} g_t^m(x_t, u_t) + \bar{C}^m(x_k) \leq b^m, \quad m = 1, \ldots, M,
\]

and from Eq. (2.54), it follows that

\[
\sum_{t=0}^{k-1} g_t^m(x_t, u_t) + g_k^m(x_k, u_k) + \bar{C}^m(x_{k+1}) \leq b^m, \quad m = 1, \ldots, M.
\]

This implies that the complete trajectory

\[
(x_0, u_0, x_1, \ldots, u_{k-1}, x_k, u_k, x_{k+1}) \cup R(x_{k+1})
\]

belongs to \( C \), thereby verifying the feasibility portion of the definition of sequential improvement.

The Fortified Rollout Algorithm

We will now discuss some variations and extensions of the constrained rollout algorithm. Let us consider the case where the sequential improvement assumption is not satisfied. Then it may happen that given the current partial trajectory \( \tilde{y}_k \), the set of controls \( U_k(\tilde{y}_k) \) that corresponds to feasible trajectories \( T_k(\tilde{y}_k, u_k) \) [cf. Eq. (2.46)] 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 in Section 2.3.2. 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 \( \hat{x}_k \), i.e., \( \hat{T}_k \) has the form

\[
\hat{T}_k = (\hat{x}_0, \hat{u}_0, \hat{x}_1, \ldots, \hat{u}_{k-1}, \hat{x}_k, \overline{x}_k, \overline{x}_{k+1}, \ldots, \overline{x}_{N-1}, \overline{x}_N), \quad (2.55)
\]

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

\[
\overline{x}_{k+1} = f_k(\hat{x}_k, \overline{x}_k), \quad \overline{x}_{t+1} = f_i(\overline{x}_t, \overline{x}_t), \quad t = k + 1, \ldots, N - 1.
\]

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 \( k \):

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. (2.55)], 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 \( \tilde{T}_0 \), 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 \( \tilde{T}_0 \) as the tentative best trajectory. Similarly, the new tentative best trajectory \( \hat{T}_k \) may be subsequently replaced by another feasible trajectory with smaller cost, etc.

Note that if the base heuristic is sequentially improving, and the fortified rollout algorithm will generate the same complete trajectory as the (nonfortified) rollout algorithm given earlier, with the tentative best trajectory \( \hat{T}_{k+1} \) being equal to the complete trajectory \( T_k(\tilde{y}_k, \tilde{u}_k) \) for all \( k \). The reason is that if the base heuristic is sequentially improving the controls \( \tilde{u}_k \) generated by the nonfortified algorithm belong to the set \( \hat{U}_k(\tilde{y}_k) \) [by Prop. 2.7.1, case (1) above will hold].

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

It is possible to improve the performance of the rollout algorithm at the expense of maintaining more than one partial trajectory. In particular, instead of the partial trajectory \( \tilde{y}_k \) of Eq. (2.50), 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 the rollout algorithm of Section 2, 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 return to 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. By contrast, the constrained rollout algorithm can be applied on-line, provided the computation relating to feasibility can be dealt with sufficiently fast.

2.7.1 Constrained Multiagent Rollout

Let us assume a special structure of the control space, where the control \( u_k \) consists of \( m \) components, \( u_k = (u_{k1}, \ldots, u_{km}) \), each belonging to a corresponding set \( U_{k\ell}(x_k) \), \( \ell = 1, \ldots, m \). Thus the control space at stage \( k \) is the Cartesian product

\[
U_k(x_k) = U_{k1}(x_k) \times \cdots \times U_{km}(x_k).
\]  

(2.56)

Similar to Section 2.6, we refer to this as the multiagent case, motivated by the special case where each component \( u_{k\ell} \), \( \ell = 1, \ldots, m \), is chosen by a separate agent \( \ell \) at stage \( k \). Then the rollout minimization (2.51) involves the computation and comparison of as many as \( n^m \) terms \( G(T_k(\tilde{y}_k, u_k)) \), where \( n \) is the maximum number of elements of the sets \( U_{k\ell}(x_k) \) [so that \( n^m \) is an upper bound to the number of controls in the control space \( U_k \), in view of its Cartesian product structure (2.56)]. Thus the rollout algorithm requires order \( O(n^m) \) applications of the base heuristic per stage.
the state $x_k$ into its $m$ components, and between $x_k$ and the next state $x_{k+1} = f_k(x_k, u_k)$, we introduce artificial intermediate “states” $(x_k, u_k^{1}), (x_k, u_k^{2}), \ldots, (x_k, u_k^{m-1})$, and corresponding transitions. The choice of the last control component $u_k^{m}$ at “state” $(x_k, u_k^{1}, \ldots, u_k^{m-1})$ marks the transition at cost $g_k(x_k, u_k)$ to the next state $x_{k+1} = f_k(x_k, u_k)$ according to the system equation; see Fig. 2.7.2. 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 structures of the two problems are essentially the same.

Consider now the constrained rollout algorithm applied to the reformulated problem of Fig. 2.7.2, with a base heuristic suitably modified so that it completes a partial trajectory of the form

$$(y_k, (x_k, u_k^{1}), (x_k, u_k^{2}), \ldots, (x_k, u_k^{m-1})),$$

$\ell = 1, \ldots, m.$

The algorithm involves a minimization over only one control component at the state $x_k$ and at each of the intermediate states

$$(x_k, u_k^{1}), (x_k, u_k^{2}), \ldots, (x_k, u_k^{m-1}).$$

In particular, for each stage $k$, the algorithm requires a sequence of $m$ minimizations, one over each of the control components $u_k^{1}, \ldots, u_k^{m}$, with the past controls already determined by the rollout algorithm, and the
future controls determined by running the base heuristic. Assuming a maximum of \( n \) elements in the control component spaces \( U_k^\ell, \ell = 1, \ldots, m \), the computation required at each stage \( k \) is of order \( O(n) \) for each of the “states”
\[
x_k, (x_k, u_k^1), \ldots, (x_k, u_k^m), \ldots, (x_N, u_N^1),
\]
for a total of order \( O(nm) \) computation.

To elaborate, for all \( k \) and \( \ell \leq m \), at the current partial trajectory
\[
(\tilde{x}_0, \tilde{u}_0, \ldots, \tilde{x}_k, \tilde{u}_k, \ldots, \tilde{u}_k^{\ell-1}),
\]
and for each of the controls \( u_k^\ell \), we use the base heuristic to generate a complementary partial trajectory
\[
(u_k^{\ell+1}, \ldots, u_k^m, x_k^{k+1}, u_k^{k+1}, \ldots, x_N^{N-1}, u_N^{N-1}, x_N),
\tag{2.57}
\]
up to stage \( N \). We then select the control \( \tilde{u}_k^\ell \) for which the resulting complete trajectory is feasible and has minimum cost. There is also a fortified version of this algorithm, which is similar to the ones described in Sections 2.3.1 and 2.6.

Note that the base heuristic used in the reformulated problem must be capable of generating a complementary partial trajectory of the form (2.57), starting from any partial trajectory of states and controls. Note also that instead of selecting the agent controls in a fixed order, it is possible to change the order at each stage \( k \). In fact it is possible to optimize over multiple orders at the same stage.

**Example 2.7.2 (Constrained Multi-Vehicle Routing)**

There are interesting constrained versions of the multi-vehicle routing Example 2.6.2. For example, the vehicles may have capacity constraints (a limit on how many tasks some vehicles can perform), or may be specialized (some tasks can be performed only by some of the vehicles). There may be constraints on the tasks also, such as time windows (some tasks must be performed within specified time intervals). We can use constrained one-vehicle-at-a-time rollout for such problems, with the same base heuristic as in the unconstrained case of Example 2.6.2. Figure 2.7.3 illustrates this algorithm for the case where one of the vehicles has a capacity constraint.

We finally note that a fortified version of the multiagent rollout algorithm may be useful to restore and maintain the cost improvement property in a distributed asynchronous multiagent context such as the one described in Section 2.6. On the other hand, for constrained problems it may be harder to maintain feasibility when the agents choose their controls without coordination, so success of a distributed multiagent rollout scheme may depend on the structure of the problem at hand.
Figure 2.7.3 An instance of the vehicle routing problem of Example 2.7.2. Here vehicle 1 has limited capacity and can perform only one task. It can be verified that similar to the problem of Fig. 2.6.3, the base heuristic starting from the initial vehicle position, will move both vehicles to node 4, perform the task at node 7 with vehicle 1 in two steps, and then perform the tasks at nodes 9 and 11 with vehicle 2 in five and seven steps, respectively, for a total cost of $2 + 5 + 7 = 14$. However, the rollout algorithm will move vehicle 1 to node 9, and move vehicle 2 to node 7 and then to node 11, for a total cost of $3 + 2 + 5 = 10$, which is also the optimal cost. As in the case of the problem of Fig. 2.6.3, the rollout works better because it corrects the “limited vision” deficiency of the base heuristic through longer exploration.

2.8 DISCRETE AND COMBINATORIAL OPTIMIZATION APPLICATIONS

In this section we will illustrate the application of constrained multiagent rollout within the context of some classical discrete optimization problems. In this regard, it is important to note that the constrained deterministic optimal control problem of the preceding section is very general. In particular, it contains as a special case the fully unstructured discrete optimization problem:

$$
\begin{align*}
\text{minimize} & \quad G(u) \\
\text{subject to} & \quad u \in C,
\end{align*}
$$

(2.58)

where $u$ has $N$ components; i.e., it has the form $u = (u_0, \ldots, u_{N-1})$, where $N$ is a positive integer, $C$ is a finite set of feasible solutions, and $G(u)$ is some cost function.† This is simply the special case of the deterministic optimal control problem where each state $x_k$ can only take a single

† The reverse is also true, namely that any constrained deterministic optimal control problem of the form (2.38)-(2.40), can be converted to the general discrete optimization form (2.58), simply by expressing the states $x_k$ as functions of the preceding controls $u_0, \ldots, u_{k-1}$ through the system equation (2.39), and eliminating them from the cost function expression and the constraints. This ab-
value. Then the state space for each $k$ has a single element, and the system equation $x_{k+1} = f_k(x_k, u_k)$ is trivial and superfluous. Then in effect the partial trajectory $y_k$ is the $k$-tuple $(u_0, \ldots, u_{k-1})$ consisting of the first $k$ components of a solution.

We associate such a $k$-tuple with the $k$th stage of the finite horizon DP problem shown in Fig. 2.8.1. In particular, for $k = 0, \ldots, N - 1$, we view as the states of the $k$th stage all the possible $k$-tuples $(u_0, \ldots, u_{k-1})$. The initial state is some artificial state. From this state we may move to any state $(u_0, \ldots, u_{k-1})$, with $u_0$ belonging to the set

$$U_0 = \{ u_0 \mid \text{there exists a solution of the form } (u_0, u_1, \ldots, u_{N-1}) \in C \}.$$ 

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 $(u_0, \ldots, u_{k-1}, u_k)$, such that $u_k$ belongs to the set

$$U_k(u_0, \ldots, u_{k-1}) = \{ u_k \mid \text{there exists a solution of the form } (u_0, \ldots, u_k, u_{k+1}, \ldots, u_{N-1}) \in C \}.$$ 

These are the set of 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 complete solutions $u = (u_0, \ldots, u_{N-1})$, with cost $G(u)$; see Fig. 2.8.1. All other transitions in this DP problem formulation have cost 0.

Abstraction of the problem may be of value in some contexts because of its inherent simplicity.
Of course here the number of states typically grows exponentially, but we can still apply the constrained rollout algorithm to the preceding DP formulation, using a suitable base heuristic, which will be applied only $N$ times.

An important class of combinatorial problems involve layered graphs with both a temporal and a spatial allocation structure. In this way both the dynamic system character and the multiagent character of our algorithms come into play. We have already discussed briefly a class of discrete optimization problems that involve a graph as well as a multiagent structure, namely the multi-vehicle routing problems of Example 2.6.2, and their extensions involving time windows, vehicle capacity constraints, arc traversal costs, etc. Other challenging combinatorial problems, such as multi-vehicle routing and multi-machine scheduling problems, also involve a spatio-temporal type of structure, and are well suited for the application of our constrained multiagent rollout approach. For more discussion of related combinatorial applications, we refer to Chapter 10 of the author’s network optimization book [Ber98]. Generally, the fine details of such a problem will determine the choice of a suitable base heuristic.

The following is another example of a classical problem with both multiagent structure and spatial allocation character. It can also be viewed as an instance of a 0-1 integer programming problem, and in fact this is the way it is usually addressed in the literature; see e.g., the book [DrH01].

**Example 2.8.1 (Facility Location)**

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

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

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

(2.59)

$$\sum_{i=1}^{M} y_{ik} \leq c_j u_k, \quad k = 0, \ldots, N - 1,$$

(2.60)

together with

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

(2.61)

We wish to minimize the cost

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

(2.62)
subject to the preceding constraints. The essence of the problem is to place
enough facilities at favorable locations to satisfy the clients’ demand at min-
imum cost. This can be a very difficult mixed integer programming problem.

On the other hand, when all the variables $u_k$ are fixed at some 0 or 1 values, the problem belongs to the class of linear transportation problems
(see e.g., [Ber98]), and can be solved by fast polynomial algorithms. Thus the
essential difficulty of the problem is how to select the sequence of variables
$u_k$, $j = 0, \ldots, N - 1$. This can be viewed as a discrete optimization
problem of the type shown in Fig. 2.8.1. In terms of the notation of this figure, the
control components are $u_0, \ldots, u_{N-1}$, where $u_k$ can take the two values 0 or 1.

To address the problem suboptimally by rollout, we must define a base
heuristic at a “state” $(u_0, \ldots, u_{k-1})$, where $u_j = 1$ or $u_j = 0$ specifies that a
facility is or is not placed at location $j$, respectively. A suitable base heuristic
at that state is to place a facility at all of the remaining locations (i.e., $u_j = 1$
for $j = k + 1, \ldots, N - 1$), and its cost is obtained by solving the corresponding
linear transportation problem of minimizing the cost (2.62) subject to the
constraints (2.59)-(2.61), with the variables $u_j$, $j = 0, \ldots, k - 1$, fixed at the
previously chosen values, and the variables $u_j$, $j = k, \ldots, N$, fixed at 1.

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

Figure 2.8.2 Schematic illustration of the facility location problem; cf. Ex-
ample 2.8.1. Clients are matched to facilities, and the location of the facilities
is subject to optimization.
problems, and fix $u_1$, depending on which of the two resulting costs is smallest, etc.

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

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

the rollout algorithm will yield a feasible solution with cost that is no larger than the cost corresponding to the initial application of the base heuristic. In fact it can be verified that the base heuristic here is sequentially improving, so it is not necessary to use the fortified version of the algorithm. Regarding computational costs, the number of transportation problems to be solved is at first count $2N$, but it can be reduced to $N+1$ by exploiting the fact that one of the two transportation problems at each stage after the first has been solved at an earlier stage. It is finally worth noting, for readers that are familiar with the integer programming method of branch-and-bound, that the graph of Fig. 2.8.1 corresponds to the branch-and-bound tree for the problem, so the rollout algorithm amounts to a quick (and imperfect) method to traverse the branch-and-bound tree. This observation may be useful if we wish to use integer programming techniques to add improvements to the rollout algorithm.

We finally note that the rollout algorithm requires the solution of many linear transportation problems, which fairly similar data. It is thus important to use an algorithm that is capable of using effectively the final solution of one transportation problem as a starting point for the solution of the next. The auction algorithm for transportation problems (Bertsekas and Castaño [Ber89]) is particularly well-suited for this purpose.

Figure 2.8.3 Schematic illustration of rollout for the RNA folding problem.
Example 2.8.2 (RNA Folding)

In a classical problem from computational biology, we are given a sequence of nucleotides, represented by circles in Fig. 2.8.3, and we want to “fold” the sequence in an “interesting” way (introduce pairings of nucleotides that result in an “interesting” structure). There are some constraints on which pairings are possible, but we will not go into the details of this. Another constraint is that the pairings should not “cross,” i.e., given a pairing \((i_1, i_2)\) there should be no pairing \((i_3, i_4)\) where either \(i_3 < i_1\) and \(i_1 < i_4 < i_2\), or \(i_1 < i_3 < i_2\) and \(i_2 < i_4\).

We formulate this as a discrete optimization problem involving a pairing decision at each nucleotide in the sequence with at most three choices (open a pairing, close a pairing, do nothing); see Fig. 2.8.3. To apply rollout, we need a base heuristic, which given a partial folding, generates a complete folding (this is the partial folding software shown in Fig. 2.8.3). Two complete foldings can be compared by some other software, called the critic software.

An interesting aspect of this problem is that there is no explicit cost function here (it is internal to the critic software); in fact biologists do not agree on what the appropriate cost function is. Thus by trying different partial folding and critic software, we may obtain multiple solutions, which may be used for further experimental evaluation; we may refer to Pedrielli [PLS20] for further discussion of this rollout approach.

2.8.1 Multidimensional Assignment

We will now focus on multidimensional assignment, a class of combinatorial problems that have both a temporal and a spacial allocation structure. They involve graphs consisting of \(N + 1\) subsets of nodes \((N \geq 2)\), denoted \(N_0, N_1, \ldots, N_N\), and referred to as layers. The arcs of the graphs are directed and are of the form \((i, j)\), where \(i\) is a node in a layer \(N_k\), \(k = 0, 1, \ldots, N - 1\), and \(j\) is a node in the corresponding next layer \(N_{k+1}\). Thus we have a directed graph whose nodes are arranged in \(N + 1\) layers and the arcs connect the nodes of each layer to the nodes in their adjacent layers; see Fig. 2.8.4. Here for simplicity, we assume that each of the layers \(N_k\) contains the same number of nodes, say \(m\), and that there is a unique arc connecting each node in a given layer with each of the nodes of the adjacent layers.

We consider subsets of \(N + 1\) nodes, referred to as groupings, which contain a single node from every layer, and we assume that every grouping is associated with a given cost. A partition of the set of nodes into \(m\) disjoint groupings (so that each node belongs to one and only one grouping) is called an \((N + 1)\)-dimensional assignment. For each grouping, there is an associated cost, which depends on the \(N\)-tuple of arcs that comprise the grouping. The cost of an \((N + 1)\)-dimensional assignment is the sum of the costs of its \(m\) groupings. The problem is to find an \((N + 1)\)-dimensional assignment of minimum cost.
Three-Dimensional Assignment Problem

Figure 2.8.4. Illustration of the graph of an \((N + 1)\)-dimensional assignment problem (here \(N = 5\)). There are \(N + 1\) node layers each consisting of \(m\) nodes (here \(m = 4\)). Each grouping consists of \(N + 1\) nodes, one from each layer, and \(N\) corresponding arcs. An \((N + 1)\)-dimensional assignment consists of \(m\) node-disjoint groupings, where each node belongs to one and only one grouping (illustrated in the figure with thick red lines). For each grouping, there is an associated cost, which depends on the \((N + 1)\)-tuple of arcs that comprise the grouping. The cost of an \((N + 1)\)-dimensional assignment is the sum of the costs of its \(m\) groupings. In the separable case, the cost of a grouping separates into the sum of its \(N\) arc costs, and the problem can be solved by solving \(N\) decoupled 2-dimensional assignment problems.

This is a difficult combinatorial problem with many applications. An important special case arises in the context of multi-target tracking and data association; see Blackman [Bla86], Bar-Shalom and Fortman [BaF88], Bar-Shalom [Bar90], Pattipati, Deb, Bar-Shalom, and Washburn [PDB92], Poore [Poo94], Poore and Robertson [PoR97], Popp, Pattipati, and Bar-Shalom [PPB01], and Choi, Brunet, and How [CBH09].

We note that there are several variants of the multidimensional assignment problem illustrated in Fig. 2.8.4, which are well-suited for the application of constrained rollout. For example, these variants may involve unequal numbers of nodes in each layer, or a sparse structure where some of the possible arcs connecting nodes of adjacent layers are missing. Moreover, there may be cost coupling between collections of groupings that depends on the groupings’ compositions. In this section we will focus on the case where the layers have equal numbers of nodes and where the cost of each grouping depends exclusively on the \(N + 1\) nodes that comprise the grouping. This structure favors the use of base heuristics that rely on solution of 2-dimensional assignment problems.

Three-Dimensional Assignment

To simplify the presentation, we will first focus on the 3-dimensional as-
Sec. 2.8 Discrete and Combinatorial Optimization Applications

3-Dimensional Assignment Problem

![Diagram of a 3-dimensional assignment problem](image)

**Figure 2.8.5.** A 3-dimensional assignment problem consisting of assigning each job $j$ to a machine $\ell$ and to a worker $w$ at cost $a_{j\ell w}$. Each machine assigned to exactly one job and exactly one worker. Here $N = 2$ and $m = 4$.

Assignment special case ($N = 2$), and for descriptive purposes, we will associate the nodes of the three layers with “jobs,” “machines,” and “workers,” respectively. Suppose that the performance of a job $j$ requires a single machine $\ell$ and a single worker $w$ (which cannot be shared by any other job), and that there is a given cost $a_{j\ell w}$ corresponding to the triplet $(j, \ell, w)$. Given a set of $m$ jobs, a set of $m$ machines, and a set of $m$ workers, we want to find a collection of $m$ job-machine-worker triplets that has minimum total cost. This problem is quite challenging, and is well-suited for demonstration of our constrained rollout approach as it has both a temporal and a spacial character.

To transcribe the problem to our deterministic optimal control format, we use the general discrete optimization formulation of Fig. 2.8.1, and assume that there is only one state at each of three stages [the respective collections of jobs (for the first stage), machines (for the second stage), and workers (for the third stage)], and two decisions to make (the assignment of jobs to machines and the assignment of machines to workers); see Fig. 2.8.5. Each of the decisions consists of $m$ components, the $m$ outgoing arcs from the $m$ nodes corresponding to the stage. Thus, the application of the multiagent rollout algorithm of Section 2.7 will involve two stages, a state space consisting of a single element for each state, and a control at each stage that consists of $m$ components (the choice of machine to assign to each job in the first stage, and the choice of worker to assign to each machine in the second stage). These components are computed in sequence according to some predetermined order, which without loss of generality we will assume to be the natural order $1, 2, \ldots, m$.

An important and particularly favorable special case of the problem arises when the costs $a_{j\ell w}$ have the separable form

$$a_{j\ell w} = \beta_{j\ell} + \gamma_{\ell w},$$
where $\beta_{j\ell}$ and $\gamma_{\ell w}$ are given scalars. In this case, there is no coupling between jobs and workers, and the problem can be efficiently (polynomially) solved by solving two decoupled (2-dimensional) assignment problems: one involving the pairing of jobs and machines, with the $\beta_{j\ell}$ as costs, and the other involving the pairing of machines and workers, with the $\gamma_{\ell w}$ as costs. In general, however, the 3-dimensional assignment problem is a difficult integer programming problem, for which there is no known polynomial algorithm.

**The Enforced Separation Heuristic and Three-Dimensional Assignment**

The separable case motivates a simple heuristic approach for the nonseparable 3-dimensional assignment problem, which is well-suited for the use of constrained rollout. The base heuristic is a two-stage procedure that is based on relaxing each the grouping constraints, by first focusing on assigning machines to workers, and then focusing on assigning jobs to machines, after suitably modifying the costs $a_{j\ell w}$ to make them separable. In particular, we first relax the constraints on the jobs by assuming that each machine-worker pair can be assigned to the most favorable job. Mathematically, this relaxed problem takes the polynomially solvable 2-dimensional assignment form

$$
\begin{align*}
\text{minimize} \quad & \sum_{w=1}^{m} \sum_{\ell=1}^{m} c_{\ell w} v_{\ell w} \\
\text{subject to} \quad & \sum_{\ell=1}^{m} v_{\ell w} = 1, \quad \forall \ w = 1, \ldots, m, \\
& \sum_{w=1}^{m} v_{\ell w} = 1, \quad \forall \ \ell = 1, \ldots, m, \\
& v_{\ell w} = 0 \text{ or } 1, \quad \forall \ \ell, \ w = 1, \ldots, m,
\end{align*}
$$

where

$$
c_{\ell w} = \min_{j=1,\ldots,m} a_{j\ell w},
$$

$v_{\ell w}$ are the variables of the problem, and $v_{\ell w} = 1$ indicates that machine $\ell$ is assigned to worker $w$, so the constraints of the above problem enforce the condition that each machine is assigned to one and only one worker.†

† An alternative is to define each cost $c_{\ell w}$ as a “representative” cost $a_{j\ell w}$ (for some specially selected job $j_{\ell w}$). Such an alternative may become attractive when extensions of enforced separation are considered for $(N+1)$-dimensional problems with large $N$. Then the analog of Eq. (2.63) will involve minimization over the (exponential in $N$) number of all $(N-2)$-tuples of graph arcs that can form a grouping with $(\ell, w)$, and may become very costly.
For each \( w \), let \( w_\ell \) be the worker assigned to machine \( \ell \), according to the solution of this problem. We can now optimally assign jobs \( j \) to machine-worker pairs \((\ell, w_\ell)\) by using as assignment costs

\[
b_{j_\ell} = a_{j_\ell w_\ell},
\]

and obtain a (suboptimal) 3-dimensional assignment \( \{(j_\ell, \ell, w_\ell) \mid \ell = 1, \ldots, m\} \). It can be seen that this approach amounts to enforced separation, whereby we replace the costs \( a_{j_\ell w_\ell} \) with the separable approximations \( b_{j_\ell} + c_{\ell w} \). In fact, it can be shown that if the problem is \( \epsilon \)-separable, in the sense that for some (possibly unknown) \( \beta_{j_\ell} \) and \( \gamma_{\ell w} \), and some \( \epsilon \geq 0 \), we have

\[
|\beta_{j_\ell} + \gamma_{\ell w} - a_{j_\ell w_\ell}| \leq \epsilon, \quad \forall j, \ell, \ w = 1, \ldots, m,
\]

then the assignment \( \{(j_\ell, \ell, w_\ell) \mid \ell = 1, \ldots, m\} \) obtained using the preceding enforced separation approach achieves the optimal cost of the problem within \( 4m\epsilon \) (see the author’s network optimization book [Ber98], Exercise 10.31).

The enforced separation approach is simple and can be generalized to problems with more than two stages as we will discuss shortly. Moreover, enforced separation heuristics also apply to several variants of the multidimensional assignment problem. For example, we may have transportation-type constraints, where multiple jobs can be performed on the same machine, and/or multiple machines can be operated by a single worker. In this case, our preceding discussion of the enforced separation heuristic applies similarly, except that we need to solve 2-dimensional transportation problems rather than 2-dimensional assignment problems.

**Using Enforced Separation as a Base Heuristic**

We will now describe the use of enforced separation as a base heuristic in the context of constrained rollout. The 3-dimensional assignment problem is posed as an optimal control problem involving \( m + 1 \) sequential choices: the machines assigned to the jobs are first selected one-by-one in some fixed order (\( m \) sequential choices), and then the workers assigned to the machines are selected simultaneously. To connect with our earlier optimal control formulation, trajectories here consist of an artificial initial state, the \( m \) successive choices of job-machine pairs (these correspond to the controls \( u_1^1, \ldots, u_m^m \)), and then the \( m \)-tuple of machine-worker pairs (these comprise the control \( u_1 \)). For each of the first \( m \) choices a job is selected and the machine to be assigned to this job is fixed by the rollout algorithm, through the use of the base heuristic of enforced separation. At the last stage the \( m \) machines are assigned simultaneously to workers using a 2-dimensional assignment algorithm.
To illustrate the fortified rollout algorithm, at the artificial initial state where no job-machine or machine-worker pair has been fixed, the enforced separation heuristic as described above is used to generate a (suboptimal) initial 3-dimensional assignment, which serves as the initial tentative best trajectory, and has cost denoted by $\hat{S}$.

In the first $m$ rollout stages we select in sequence each job $j = 1, \ldots, m$, and we select a machine $\ell$ to assign to it, by using the enforced separation heuristic. The first rollout stage is as follows:

Stage 1.1: We take job $j = 1$, and fix its assignment to machine 1. We then apply the enforced separation heuristic by solving two 2-dimensional assignment problems. The first of these involves the assignment of machines to workers using as costs

$$c_{\ell w} = \begin{cases} a_{11w} & \text{if } \ell = 1, \\ \min_{j=2,\ldots,m} a_{j\ell w} & \text{if } \ell \neq 1, \end{cases}$$

[cf. Eq. (2.63)]. We thus obtain an assignment of machines to workers of the form $(\ell, w_\ell), \ell = 1, \ldots, m$. Having fixed the workers to be assigned to machines, we solve the 2-dimensional assignment problem of assigning the jobs $2, \ldots, m$ to the machines $2, \ldots, m$, where the costs are

$$b_{j\ell} = a_{j\ell w_\ell}, \quad j = 2, \ldots, m.$$

By “joining” the solutions of the two 2-dimensional problems just described, we obtain a 3-dimensional assignment that consists of $m$ job-machine-worker groupings whose cost we call $S_1$.

Stage 1.2: We take job $j = 1$, and fix its assignment to machine 2. We then apply the enforced separation heuristic by solving two 2-dimensional assignment problems. The first involves the assignment of machines to workers using as costs

$$c_{\ell w} = \begin{cases} a_{12w} & \text{if } \ell = 2, \\ \min_{j=2,\ldots,m} a_{j\ell w} & \text{if } \ell \neq 2, \end{cases}$$

[cf. Eq. (2.63)]. We obtain an assignment of machines to workers of the form $(\ell, w_\ell), \ell = 1, \ldots, w$. We then solve the 2-dimensional assignment problem of assigning the jobs $2, 3, \ldots, m$ to the machines $1, 3, \ldots, m$, where the costs are

$$b_{j\ell} = a_{j\ell w_\ell}, \quad j = 2, 3, \ldots, m, \ell = 1, 3, \ldots, m.$$

By “joining” the solutions of the two 2-dimensional problems, we obtain a 3-dimensional assignment, whose cost we call $S_2$.

Stage 1.t: For $t = 3, \ldots, m$, we continue the process described above, where we fix the assignment of job 1 to machine $t$. We then apply
the enforced separation heuristic by solving two 2-dimensional assignment problems, similar to the ones above: first assigning machines to workers using costs

\[ c_{\ell w} = \begin{cases} 
    a_{1tw} & \text{if } \ell = t, \\
    \min_{j=2, \ldots, m} a_{j\ell w} & \text{if } \ell \neq t,
\end{cases} \]

and obtaining an assignment of machines to workers of the form \((\ell, w_\ell), \ell = 1, \ldots, w\). We then solve the 2-dimensional assignment problem of assigning the jobs 2, \ldots, m to the machines 1, \ldots, t-1, t+1, \ldots, m, where the costs are

\[ b_{j\ell} = a_{j\ell w_\ell}, \quad j = 2, 3, \ldots, m, \quad \ell = 1, \ldots, t-1, t+1, \ldots, m. \]

By “joining” the solutions of the two 2-dimensional problems, we obtain a 3-dimensional assignment, whose cost we call \(S_t\).

**Stage 1:** We now have \(m\) 3-dimensional assignments, with corresponding costs \(S_1, \ldots, S_m\), where job 1 is fixed to the machines 1, \ldots, m, respectively. We choose the machine \(\tilde{\ell}\) for which \(S_{\tilde{\ell}}\) is minimized over \(\ell = 1, \ldots, m\), and we permanently assign job 1 to machine \(\tilde{\ell}\) if \(S_{\tilde{\ell}}\) is less or equal to \(\tilde{S}\), the cost of the enforced separation heuristic applied to the artificial initial state (the current tentative best trajectory), and we also adopt the corresponding 3-dimensional assignment as the new tentative best trajectory. Otherwise, we set the assignment of job 1 to a machine according to the current tentative best trajectory, which we leave unchanged.

The preceding procedure, the first step of constrained rollout, required the solution of \(2m\) 2-dimensional assignment problems, and yielded a permanent assignment of job 1 to a machine. The procedure is then repeated for job 2, taking into account that the assignment of job 1 to a machine has been fixed. This requires similarly the solution of \(2(m-1)\) 2-dimensional assignment problems, and yields a permanent assignment of job 2 to a machine, and an update of the current tentative trajectory. Repeating the procedure with jobs 3, \ldots, m in sequence, we obtain a permanent assignment of all the jobs to machines, and the corresponding 3-dimensional assignment (which has minimum cost over all the 3-dimensional assignments generated, in view of the use of fortified rollout). The total number of 2-dimensional assignment problems thus solved is

\[ 2m + 2(m-1) + 2(m-2) + \cdots + 2 = m^2. \]

Finally, given the permanent assignment of all the jobs to machines, say \((j_\ell, \ell), \ell = 1, \ldots, m\), we obtain the (permanent) assignment of workers to job-machine pairs, by using as costs the scalars

\[ c_{\ell w} = a_{j_\ell \ell w}, \quad \ell, w = 1, \ldots, m. \]
At this point we have obtained by rollout the final suboptimal 3-dimensional assignment, which by construction has the cost improvement property: it has no larger cost than the one obtained by the enforced separation base heuristic starting from the artificial initial condition.

Thus the total number of 2-dimensional assignment problems to be solved by the rollout algorithm is \( m^2 + 1 \). Each of these problems can be solved very fast using any one of a number of methods. However, because these problems and their solutions are similar, it is important to use a method that can exploit this similarity. A particularly favorable method in this regard is the author’s auction algorithm [Ber79] (see the book [Ber98] for a detailed development). The auction algorithm uses a price variable for each node, such as a worker or a machine, and then adjusts the prices through an auction-like process to achieve a form of economic equilibrium. One can then use the final prices obtained for one 2-dimensional assignment problem as an efficient starting point for the solution of a related 2-dimensional assignment problem. For this reason, the auction algorithm and its variations have been widely adopted for use in solving 2-dimensional assignment problems in the context of multidimensional assignment algorithms used for multitarget tracking applications, among others (see [PDB92], [Poo94], [PoR97], [PPB01], [CBH09]).

**Enforced Separation and Constrained Rollout for Multidimensional Assignment**

Let us now consider briefly the extension of the constrained rollout algorithm just described to \((N + 1)\)-dimensional assignment problems with \( N > 2 \). Here we will use an extension of the 3-dimensional enforced separation heuristic. We start again from the last stage, solve the last 2-dimensional assignment problem of the last stage by modifying the arc costs according to the analog of the minimization formula

\[
c_{\ell w} = \min_{j=1, \ldots, m} a_{j\ell w},
\]

[cf. Eq. (2.63)]. The only difference is that instead of minimizing \( a_{j\ell w} \) over jobs \( j \) as above, we minimize over all \((N - 1)\)-tuples of nodes of the groupings whose final two nodes are \((\ell, w)\). Once the assignments of the last stage are fixed, a similar procedure can be used to fix the assignments of the next-to-last stage, and so on. The total number of 2-dimensional assignment problems to be solved is \((m+1)(N-2)\). Thus the base heuristic’s computation time is polynomial in both \( m \) and \( N \).

The enforced separation heuristic just described for the artificial initial condition, can be used in suitably modified form for rollout, with the assignments of some stages fixed permanently by rollout, and additional assignments fixed one by one, by applying the enforced separation heuristic, with suitably modified costs that take into account the already fixed assignments.
2.9 ROLLOUT FOR MINIMAX CONTROL

The problem of optimal control of uncertain systems has traditionally been treated in a stochastic framework, whereby all disturbances \( w_0, \ldots, w_{N-1} \) are described by probability distributions, and the expected value of the cost is minimized. However, in many practical situations a stochastic description of the disturbances may not be available, and one may have information with less detailed structure, such as bounds on their magnitude. In other words, one may know a set within which the disturbances are known to lie, but may not know the corresponding probability distribution. Under these circumstances one may use a minimax approach, whereby the worst possible values of the disturbances within the given set are assumed to occur. Within this context, we take the view that the disturbances are chosen by an antagonistic opponent. This also brings up the connection of the minimax approach with the theory of games and their solution methodology.

To be specific, consider a finite horizon context, and assume that the disturbances \( w_0, w_1, \ldots, w_{N-1} \) do not have a probabilistic description but rather are known to belong to corresponding given sets \( W_k(x_k, u_k) \subset D_k, k = 0, 1, \ldots, N - 1 \), which may depend on the current state \( x_k \) and control \( u_k \). The minimax control problem is to find a policy \( \pi = \{\mu_0, \ldots, \mu_{N-1}\} \) with \( \mu_k(x_k) \in U_k(x_k) \) for all \( x_k \) and \( k \), which minimizes the cost function

\[
J_\pi(x_0) = \max_{w_k \in W_k(x_k, u_k(x_k))} \left[ g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, \mu_k(x_k), w_k) \right].
\]

The DP algorithm for this problem takes the following form, which resembles the one corresponding to the stochastic DP problem (maximization is used in place of expectation):

\[
J^*_N(x_N) = g_N(x_N),
\]

\[
J^*_k(x_k) = \min_{u_k \in U_k(x_k)} \max_{w_k \in W_k(x_k, u_k)} \left[ g_k(x_k, u_k, w_k) + J^*_{k+1} \right].
\]

This algorithm can be explained by using a principle of optimality type of argument. In particular, we consider the tail subproblem whereby we are at state \( x_k \) at time \( k \), and we wish to minimize the “cost-to-go”

\[
\max_{w_k \in W_k(x_k, \mu_k(x_k))} \left[ g_N(x_N) + \sum_{i=k}^{N-1} g_i(x_i, \mu_i(x_i), w_i) \right],
\]

and we argue that if \( \pi^* = \{\mu^*_0, \mu^*_1, \ldots, \mu^*_N\} \) is an optimal policy for the minimax problem, then the truncated policy \( \{\mu^*_k, \mu^*_{k+1}, \ldots, \mu^*_{N-1}\} \) is...
optimal for the tail subproblem. The optimal cost of this subproblem is $J_k^*(x_k)$, as given by the DP algorithm (2.64)-(2.65). The algorithm expresses the intuitively clear fact that when at state $x_k$ at time $k$, then regardless of what happened in the past, we should choose $u_k$ that minimizes the worst/maximum value over $w_k$ of the sum of the current stage cost plus the optimal cost of the tail subproblem that starts from the next state. For a detailed mathematical derivation, we refer to the author’s textbook [Ber17], Section 1.6.

Approximation in value space with one-step lookahead applies at state $x_k$ a control

$$
\tilde{u}_k \in \arg \min_{u_k \in U(x_k)} \max_{w_k \in W_k(x_k,u_k)} \left[ g_k(x_k,u_k,w_k) + \tilde{J}_{k+1}(f_k(x_k,u_k,w_k)) \right],
$$

(2.66)

where $\tilde{J}_{k+1}(x_{k+1})$ is an approximation to the optimal cost-to-go $J^*_{k+1}(x_{k+1})$ from state $x_{k+1}$. Rollout is obtained when this approximation is the tail cost of some base policy $\pi = \{\mu_0, \ldots, \mu_{N-1}\}$:

$$
\tilde{J}_{k+1}(x_{k+1}) = J_{k+1,\pi}(x_{k+1}).
$$

Given $\pi$, we can compute $J_{k+1,\pi}(x_{k+1})$ by solving a deterministic maximization DP problem with the disturbances $w_{k+1}, \ldots, w_{N-1}$ playing the role of “optimization variables/controls.” For finite state, control, and disturbance spaces, this is a shortest path problem defined on an acyclic graph, since the variables $u_{k+1}, \ldots, u_{N-1}$ are fixed at the values dictated by the base policy. Thus, in the finite spaces case, it is straightforward to implement rollout: at $x_k$ we generate all next states of the form

$$
x_{k+1} = f_k(x_k,u_k,w_k)
$$
corresponding to all possible values of $u_k \in U_k(x_k)$ and $w_k \in W_k(x_k,u_k)$. We then run the maximization/shortest path problem described above from each of these possible next states. Finally, we obtain the rollout control $\tilde{u}_k$ by solving the minimax problem in Eq. (2.66).

It can be seen therefore that the implementation of minimax rollout is similar to rollout for stochastic DP problems. Instead of Monte-Carlo simulation, we compute the values $\tilde{J}_{k+1}(x_{k+1})$ by solving a deterministic maximization DP problem. Moreover the variants of rollout discussed in Sections 2.3 and 2.4 have analogs in the minimax context, e.g., truncation with terminal cost approximation, multistep lookahead, and multiagent rollout. In fact when multistep lookahead is used, special techniques such as alpha-beta pruning may be used to accelerate the computations by eliminating unnecessary portions of the lookahead tree. These techniques are well-known in the context of the two-person computer game methodology, and are used widely in games such as chess.
2.10 NOTES AND SOURCES

In this chapter, we have stressed the idea that in RL the key objects to approximate are values and policies. This leads to a broad division between approximation in value space and approximation in policy space approaches, and highlights the possibility of synergistic value space and policy space approximations.

The structure of approximation in value space involves limited lookahead minimization, and is depicted in the key Fig. 2.2.1. There we have argued that there are three candidate areas for approximation to consider:

(a) Cost-to-go function approximation, which defines the approximate Q-factors at a given state.

(b) Simplification of expected values involved in the calculation of the approximate Q-factors.

(c) Simplification of the Q-factor minimization over all admissible controls.

There are several candidate approaches for each of these three approximations. Significantly, the three approximations are largely decoupled from each other, allowing a broad spectrum of mixtures of algorithmic choices.

Approximation in policy space is based on somewhat different algorithmic ideas than approximation in value space: optimization within a restricted class of parametrized policies. It requires a methodological viewpoint that is less connected with DP, but it may be combined with approximation in value space, as we have noted in Section 2.1.

In Section 1.5 we provided a list of several exact DP and approximate DP/RL textbooks and survey papers. In what follows in this section, and in other similar end-of-chapter sections, we will aim at a more targeted if inevitably incomplete set of citations to specific research topics.

Sections 2.1, 2.2: Approximation in value space has been considered in an ad hoc manner since the early days of DP, motivated by the curse of dimensionality. The idea was reframed and coupled with model-free simulation methods that originated in the 1980s in artificial intelligence.

Section 2.3, 2.4: The main idea of rollout algorithms, obtaining an improved policy starting from some other suboptimal policy, has appeared in several DP application contexts. The name “rollout” was coined by Tesauro in specific reference to rolling the dice in the game of backgammon [TeG96]. In Tesauro’s proposal, a given backgammon position is evaluated by “rolling out” many games starting from that position, using a simulator, and the results are averaged to provide a “score” for the position; see Example 2.4.2. The use of the name “rollout” has gradually expanded beyond its original context; for example the samples collected through trajectory simulation are referred to as “rollouts” by some authors. In this
Rollout and Policy Improvement Chap. 2

In this chapter, we will adopt the original intended meaning: policy improvement starting from a base policy, which is evaluated with some form of Monte Carlo simulation.

The application of rollout algorithms to discrete deterministic optimization problems, the notions of sequential consistency, sequential improvement, fortified rollout, and the use of multiple heuristics (also called “parallel rollout”) were first given in the paper by Bertsekas, Tsitsiklis, and Wu [BTW97], and also in the books by Bertsekas and Tsitsiklis [BeT96], and Bertsekas [Ber98]. Rollout algorithms for stochastic problems were further formalized in the papers by Bertsekas [Ber97b], [Ber05a], [Ber05b], and Bertsekas and Castanon [BeC99]. A survey of rollout in discrete optimization is given by the author in [Ber13a].

There have been many works and applications relating to rollout algorithms. See Christodoulouas [Chr97], Duin and Voss [DuV99], Secomandi [Sec00], [Sec01], [Sec03], Ferris and Voelker [FeV02], [FeV04], Mc Govern, Moss, and Barto [MMB02], Savagaonkar, Givan, and Chong [SGC02], Bert simas and Popescu [BeP03], Guerriero and Mancini [GuM03], Tu and Pattipati [TuP03], Wu, Chong, and Givan [WCG03], Chang, Givan, and Chong [CGC04], Meloni, Pacciarelli, and Pranzo [MPP04], Yan, Diaconis, Rusevichientong, and Van Roy [YDR04], Besse and Chaib-draa [BeC08], Sun et al. [SZL08], Bertazzi et al. [BBG13], Sun et al. [SLJ13], Tesauro et al. [TGL13], Antunes and Heemels [AnH14], Beyme and Leung [BeL14], Good son, Thomas, and Ohlmann [GTO15], Khashooei, Antunes, and Heemels [KAH15], Li and Womer [LiW15], Mastin and Jallet [Maj15], Huang, Jia, and Guan [HJG16], Simroth, Holfeld, and Brunsch [SHB15], Lan, Guan, and Wu [LGW16], Ulmer [Ulm17], Bertazzi and Secomandi [BeS18], Guer riero, Di Puglia, and Macrina [GDP18], Sarkale et al. [SNC18], Ulmer et al. [UGM18], Chu, Xu, and Li [CXL19]. These works discuss variants and problem-specific adaptations of rollout algorithms for a broad variety of practical problems, and consistently report positive computational experience.

The idea of rollout that uses limited lookahead, adaptive pruning of the lookahead tree, and rollout truncation with cost function approximation at the end of the rollout was suggested by Tesauro and Galperin [TeG96] in the context of backgammon. Related ideas appeared earlier in the paper by Abramson [Abr90], in a game playing context.

The paper and the 2007 1st edition of the monograph by Chang, Hu, Fu, and Marcus [CFH05], [CFH13] proposed and analyzed adaptive sampling in connection with DP, and early forms of Monte Carlo tree search, including statistical tests to control the sampling process. The name “Monte Carlo tree search” (Section 2.4.2) has become popular, and in its current use, it encompasses a broad range of methods that involve adaptive sampling, rollout, extensions to sequential games, and the use of statistical tests. We refer to the papers by Coulom [Cou06], the survey by Browne et al. [BPW12], and the discussions by Chang et al. [CFH16] and Fu [Fu17].
Statistical tests for adaptive sampling has been inspired by works on multiarmed bandit problems; see Lai and Robbins [LaR85], Agrawal [Agr95], Burnetas and Katehakis [BuK97], Meuleau and Bourgine [MeB99], Auer, Cesa-Bianchi, and Fischer [ACF02], Peret and Garcia [PeG04], Kocsis and Szepesvari [KoS06], Dimitrakakis and Lagoudakis [DiL08], Audibert, Munos, and Szepesvari [AMS09], and the monograph by Munos [Mun14].

The technique for variance reduction in the calculation of Q-factor differences (Section 2.4.2) was given in the author’s paper [Ber97b].

**Section 2.5**: The MPC approach is popular in a variety of control system design contexts, and particularly in chemical process control and robotics, where meeting explicit control and state constraints is an important practical issue. However, it requires a mathematical model and it is better suited for deterministic than for stochastic problems.

The literature on MPC is voluminous. For a survey, which gives many of the early references, see Morari and Lee [MoL99], and for a more recent survey, see Mayne [May14]. For related textbooks, see Maciejowski [Mac02], Camacho and Bordons [CaB04], Kouvaritakis and Cannon [Koc15], and Borrelli, Bemporad, and Morari [BBM17].

The view of MPC as a rollout algorithm, first suggested in the author’s review paper [Ber05a], provides a suboptimal control perspective and a connection with the RL methodology. The stability analysis given here is based on the work of Keerthi and Gilbert [KeG88]. The recent paper by Krener [Kre19] discusses methods to estimate the optimal cost function for use as terminal cost function approximation, aiming to achieve stabilization with MPC lookahead that is as small as possible. Generally, any problem approximation method that yields a “good” estimate of the optimal cost-to-go function is a promising candidate for terminal cost function approximation in the context of MPC and rollout. Proving stability of the overall scheme, however, depends on whether this cost function approximation satisfies a sequential improvement condition (sometimes also referred to as a “Lyapunov condition” in the control literature).

In our account of MPC, we have restricted ourselves to deterministic problems possibly involving tight state constraints as well as control constraints. Problems with stochastic uncertainty and state constraints are more challenging because of the difficulty of guaranteeing that the constraints are satisfied; see the survey by Mayne [May14] for a review of various approaches that have been used in this context. The textbook [Ber17], Section 6.4, describes MPC for problems with set membership uncertainty and state constraints, using target tube/reachability concepts, which originated in the author’s PhD thesis and related papers [Ber71], [Ber72], [BeR71], [BeR73]. Target tubes were also used subsequently in MPC and other contexts by several authors; see the surveys by Blanchini [Bla99] and Mayne [May14]. Reachability for continuous-time games has been studied by Mitchell, Bayen, and Tomlin [MBT05]. For an alternative
recent approach to reachability, which is based on the notion of Conditional Value-at-Risk (CVaR), see Chapman et al. [CLT19].

**Sections 2.6:** The material on multiagent rollout comes from the author’s recent paper [Ber19c].

**Sections 2.7:** The material on constrained rollout comes from the author’s recent paper [Ber20a]. For a discussion of constrained DP problems, the reader may consult the author’s textbook [Ber17], and the journal literature, which contains several proposals for suboptimal solution of the problem in the case where the constraints are of the form (2.43), using among others, multiobjective optimization ideas; see e.g., Jaffe [Jaf84], Martins [Mar84], Guerrier and Musmanno [GuM01], and Stewart and White [StW91], who also survey earlier work.

**Section 2.8:** The material on the application of constrained multiagent rollout to combinatorial optimization problems comes from the author’s paper [Ber20a]. To compare rollout with alternative methods, we note that there are several classes of algorithms for solving such problems. Prominent among them are local search methods, which aim to improve a complete solution \( u \) by searching for a better solution within a neighborhood of \( u \), i.e., subset of solutions that are “close” to \( u \) in some sense. In particular, given a solution \( u \), a local search method selects among the solutions in the neighborhood \( N(u) \) a successor solution \( \pi \), according to some rule. The process is then repeated with \( \pi \) replacing \( u \) (or stops when some termination criterion is met). Several classes of local search methods are available, such as genetic algorithms, tabu search, simulated annealing, and others (see for example the book [Ber98], which gives many references to the literature). Sometimes the local search involves randomization of some form, within the search neighborhood, giving rise to a broad class of random search methods.

A common characteristic of local search and random search methods is that each successive solution is complete, i.e., it consists of all of its \( N \) components. Methods such as rollout, which are based on DP ideas, are different in that they generate a solution piecemeal, component-by-component. They may thus be better suited for optimal control and other problems, which have a naturally sequential character, such as scheduling and route planning. Moreover rollout algorithms have a cost improvement property, which provides a reliable foundation for their application. Local search methods do not usually offer such a guarantee. Random search methods typically offer a theoretical guarantee of convergence to a global optimum, but this guarantee is seldom meaningful, because in practice the number of iterations to converge tends to be very large.

**Section 2.9:** Exact and approximate minimax shortest path algorithms (also called “robust shortest path planning” problems) are discussed in the author’s paper [Ber19f].