This monograph represents “work in progress,” and will be periodically updated. It more than likely contains 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.

April 5, 2020
Rollout and Policy Improvement

Contents

2.1. Approximation in Value and Policy Space . . . . . . . p. 39
  2.1.1. Approximation in Value Space - One-Step and . . .
         Multistep Lookahead . . . . . . . . . . . . . . . . . . . . . . . . . . p. 39
  2.1.2. Approximation in Policy Space . . . . . . . . . p. 43
  2.1.3. Combined Approximation in Value and Policy . . .

2.2. General Issues of Approximation in Value Space . . . p. 49
  2.2.1. Model-Based and Model-Free Implementations . . . p. 49
  2.2.2. Off-Line and On-Line Implementations . . . . . . p. 50
  2.2.3. Methods for Cost-to-Go Approximation . . . . . . p. 52
  2.2.4. Methods for Simplification of the Lookahead . . .
         Minimization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 54
  2.2.5. Simplification of the Lookahead Minimization . . .
         by Q-Factor Approximation . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 55

2.3. Rollout and the Policy Improvement Principle . . . . . p. 58
  2.3.1. On-Line Rollout for Deterministic Discrete . . .
         Optimization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 62
  2.3.2. The Fortified Rollout Algorithm . . . . . . . . . . . . p. 71
  2.3.3. Truncated Rollout Algorithms . . . . . . . . . . . . . . . . . . . . . . . p. 74
  2.3.4. Rollout with an Expert . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 76

2.4. Stochastic Rollout and Monte Carlo Tree Search . . . . . p. 78
  2.4.1. Simulation-Based Implementation of the Rollout . .
         Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 80
2.4.2. Variance Reduction in Rollout - Comparing
    Advantages ......................................................... p. 83
2.4.3. Monte Carlo Tree Search ........................................ p. 85
2.4.4. Randomized Policy Improvement by Monte Carlo ...
    Tree Search ........................................................ p. 88
2.4.5. Rollout Parallelization ........................................ p. 89

2.5. On-Line Rollout for Deterministic Infinite-Spaces Problems -
    Optimization Heuristics ........................................... p. 90
2.5.1. Model Predictive Control ...................................... p. 93
2.5.2. Target Tubes and the Constrained Controllability .......
    Condition ......................................................... p. 100
2.5.3. Model Predictive Control with Terminal Cost ............ p. 104
2.5.4. Variants of Model Predictive Control ....................... p. 105

2.6. Multiagent Rollout ................................................. p. 108
2.6.1. Multiagent Coupling Through Constraints ................. p. 117
2.6.2. Multiagent Model Predictive Control ....................... p. 119
2.6.3. Asynchronous Distributed Multiagent Rollout ............ p. 120

2.7. Constrained Rollout for Deterministic Optimization .......
2.7.1. State-Constrained Rollout and Target Tubes .............. p. 125
2.7.2. Rollout with Trajectory Constraints ...................... p. 129
2.7.3. Constrained Multiagent Rollout ............................. p. 140

2.8. Discrete and Combinatorial Optimization Applications .... p. 142
2.8.1. Multidimensional Assignment ............................... p. 149

2.9. Rollout for Minimax Control ................................. p. 156

2.10. Notes and Sources ............................................. p. 159
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.
This process defines a suboptimal policy \( \tilde{\pi} = \{ \tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1} \} \) and is called one-step lookahead; see Fig. 2.1.1. There are several possibilities for selecting or computing the functions \( \tilde{J}_{k+1} \), 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,

\[
\tilde{Q}_k(x_k, u_k) = E\left\{ g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}(x_{k+1}) \right\},
\]

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

\[
\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k),
\]
Sec. 2.1 Approximation in Value and Policy Space

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 use 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 $\tilde{J}_{k+2}$ 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 $\tilde{J}_{k+\ell}^{k}(x_{k+\ell})$ 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 $\tilde{J}_{k+\ell}^{k}$ 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. 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.

As indicated in Fig. 2.1.3, there are three potential areas of approximation for infinite horizon problems: cost-to-go approximation, expected value approximation, and minimization approximation. This is similar to the finite horizon case; cf. Fig. 2.1.1.

A major advantage of the infinite horizon context is that only one approximate cost function $\hat{J}$ is needed, rather than the $N$ functions $\hat{J}_1, \ldots, \hat{J}_N$.


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.

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

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 a parametric class of some form. In particular, we can introduce a parametric family of policies (or approximation architecture, as we will call it in Chapter 3),

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

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 $\tilde{r}_k$ 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

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

during on-line operation of the system is often much easier compared with the lookahead minimization (2.1). For this reason, one of the major uses of approximation in policy space is to provide an approximate implementation.
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 \( x_k \).

of a known policy (no matter how obtained) for the purpose of convenient
on-line use.

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. This is not surprising because, when the control space
is finite, it is possible to view different controls as distinct categories of
objects, and to view any policy \( \pi = \{\mu_0, \ldots, \mu_N-1\} \) as a classifier that
assigns a state \( x_k \) at time \( k \) to category \( \mu_k(x_k) \).

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 describe a reverse process, namely how we can start from some 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.

**From Values to Policies**

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

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

(possibly with added regularization).† In particular, we may determine \(u^k_s\) 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 \((x^k_s, u^k_s)\), \(s = 1, \ldots, q\), which are viewed as state-category pairs, and then a state \(x_k\) is classified as being of “category” \(\tilde{\mu}_k(x_k, r_k)\). Parametric approximation architectures, and their training through the use of classification and regression techniques are described in
controls at given states, so \( \hat{\mu}_k \) is trained to match the behavior of the expert. Methods of this type are commonly referred to as \textit{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 \((x^s_k, u^s_k)\) through a one-step lookahead minimization of the form

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

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

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

In this case, we collect the sample state-control pairs \((x^s_k, u^s_k), s = 1, \ldots, q\), by using approximation in value space through Eq. (2.3) or Q-factor approximation through Eq. (2.4), and then apply approximation in policy space through Eq. (2.2) (i.e., approximation in policy space is built on top of approximation in value space).

\textbf{From Policies to Values to New Policies - Rollout}

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

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

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

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

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.

Chapter 3. An important modification is to use \textit{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 \( \hat{r} \).
Using the cost function or the Q-factors of a policy as a basis for approximation in value space and obtaining a new policy by lookahead minimization, constitutes the \textit{rollout algorithm}. This is one of the principal subjects 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 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 \textit{base policy}, the process of one-step or multistep lookahead with cost approximations $J_{k,\pi}$, defines a new policy $\tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$, which we will call the \textit{rollout policy}. One of the fundamental facts in DP is that the rollout policy has a \textit{policy improvement property}: $\tilde{\pi}$ has no worse cost than $\pi$, i.e., for all $x_k$ and $k$,

$$J_{k,\tilde{\pi}}(x_k) \leq J_{k,\pi}(x_k),$$

as we will show later in Section 2.4.

Since we generally cannot expect to be able to compute the base policy cost function values $J_{k,\pi}(x_k)$ at all states $x_k$, we may use instead cost function approximations that are constructed from data. One possibility is to use Monte Carlo simulation to collect many pairs of state and base policy costs $(x_k^s, J_{k,\pi}(x_k^s))$, from which to obtain cost function approximations $\tilde{J}_k$, for each of the stages $k = 1, \ldots, N$, through some form of training process. The functions $\tilde{J}_k$ approximate the base 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 (2.5) 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).

Let us also note a computationally expedient alternative to the rollout process just described, which is particularly useful for problems with a long horizon. This 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 at the end to compensate for these costs. We will call this process \textit{truncated rollout}; see Fig. 2.1.5.

\textbf{Perpetual Rollout and Approximate Policy Iteration}

The rollout process starts with a base policy, which, through approximation in value space, can generate state-control samples of the rollout policy. Once this is done, the rollout policy may be implemented by approximation in policy space and training using state-control pairs generated by the rollout policy, as discussed earlier. 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, where the approximations in value and policy space are done using limited amounts of data (e.g., use just a few samples of state-control pairs to perform a gradient-type iteration to update the parameters of a value network and/or a policy network). Methods of this type include algorithms such as
Sec. 2.2  General Issues of Approximation in Value Space

Approximation in Value Space

Figure 2.1.6 Schematic illustration of sequential approximation in value and policy space (or perpetual rollout). It produces a sequence of policies and their 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.

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 technically come under the model-based category, even if values of the functions \( g_k \) and \( f_k \) become available through complicated computer calculations (so the mathematical model is hidden inside the computer model). Still however, Monte Carlo simulation may enter the solution process of a deterministic problem for computational expediency (i.e., computing sums of a large number of terms) or other reasons. For example the games of chess and Go are perfectly deterministic, but the AlphaGo and AlphaZero programs (Silver et al. [SHM16], [SHS17]) use randomized policies and rely heavily on Monte Carlo tree search techniques, which use sampling and will be discussed in Section 2.4.3.

2.2.2 Off-Line and On-Line Implementations

In value space approximation, an important consideration is whether the functions \( \hat{J}_{k+1} \) and the corresponding suboptimal policy \( \{\hat{\mu}_0, \ldots, \hat{\mu}_{N-1}\} \) 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 \( \hat{\mu}_k(x_k) \) 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 \( \hat{J}_{k+1} \) is an important design choice. We thus distinguish between:

(i) **Off-line methods**, where the entire function \( \hat{J}_{k+1} \) in Eq. (2.1) is computed for every \( k \), before the control process begins. The values \( \hat{J}_{k+1}(x_{k+1}) \) 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 \( \hat{J}_{k+1}(x_{k+1}) \) for implementing the corresponding suboptimal policy using approximation in value space.

(ii) **On-line methods**, where most of the computation is performed just after the current state \( x_k \) becomes known, the values \( \hat{J}_{k+1}(x_{k+1}) \) 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.

† An example is when Monte Carlo integration is used to estimate integrals that are analytically available but are hard to compute.
Sec. 2.2  General Issues of Approximation in Value Space

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

On-Line Methods and Exploration

The choice between on-line and off-line methods is intimately connected with the timing of availability of data. If significant data is not available until just before or even after the control process begins, an on-line approach is typically necessary. A related issue within this context relates to whether the available data is sufficient for the purposes of control, or whether new supplementary data should be actively collected as control is being applied. The latter data collection process is known as exploration, and its design and organization may be a significant part of the overall decision making scheme. The area of multiarmed bandit optimization embodies some of the fundamental ideas of balancing the competing desires of exploitation and exploration (generate and evaluate controls that seem most promising in terms of performance versus applying inadequately explored controls and visiting unexplored parts of the state space); see Section 2.4.3.

We note that the exploration versus exploitation dilemma has also received considerable early attention in stochastic optimal control theory. In particular, there was a lot of discussion in the 60s about the dual purposes of stochastic sequential decision making: system identification (i.e., taking actions that aim towards efficient estimation of model parameters), and system control (i.e., taking actions that aim towards guiding the system towards low cost trajectories). The interplay between these two objectives is known as dual control; see Feldbaum [Fel60], Aström [Ast83], Wittenmark [Wit95]. One of its characteristics is the need for balance between “caution” (the need for conservatism in applying control, while the system is not fully known), and “probing” (the need for aggressiveness in applying control, in order to excite the system enough to be able to identify it). Just like exploration and exploitation, these notions cannot be easily quantified, but often manifest themselves in specific control schemes.

When we discuss infinite horizon problems and the policy iteration method in Chapter 4, we will encounter still another context where explo-
ration plays an important role. This is an off-line implementation context (briefly discussed in Section 2.1.3) whereby it is necessary to evaluate approximately the entire cost function of a given policy $\mu$ by simulation. To do this, we may need to generate cost samples using $\mu$, but this may bias the simulation by underrepresenting states that are unlikely to occur under $\mu$. As a result, the cost-to-go estimates of these underrepresented states may be highly inaccurate, causing potentially serious errors in the calculation of the next policy via the policy improvement operation. This is a major difficulty in simulation-based approximate policy iteration, particularly when the system is deterministic, or when the randomness embodied in the transition probabilities of the current policy is “relatively small,” since then few states may be reached from a given initial state when the current policy is simulated. There are several approaches to enhancing exploration within the policy iteration method, but no foolproof remedy. One possibility is to break down the simulation to multiple short trajectories and to ensure that the initial states employed form a rich and representative subset. Parallel computation may play an important role within this context. Another possibility is to artificially introduce some extra randomization in the simulation of the current policy $\mu$, by occasionally using randomly generated transitions rather than the ones dictated by $\mu$. We postpone the discussion of these and other possibilities for later.

Finally, let us note that the issue of exploration does not arise within the context of rollout algorithms (at least for “standard” rollout schemes). This contributes significantly to the reliability and the convenient implementation of these methods.

### 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 $\tilde{\mu}_k$. Again there are several exact and approximate methods for control selection, some of which will be discussed in this chapter (see Fig. 2.2.1). 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 $\tilde{J}_k$ in Eq. (2.1) are obtained from a given parametric class of functions $\tilde{J}_k(x_k, r_k)$, where $r_k$ is a parameter vector, selected by a suitable algorithm. The parametric class 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 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

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

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. (2.6). 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 (2.6) 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

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

(2.7)

The approach of turning a stochastic problem into a deterministic one by replacing uncertain quantities with single typical values highlights the possibility that \( J_{k+1} \) 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 (2.6) 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. (2.6) and (2.7). If \( U_k(x_k) \) is a finite set, the minimization can be
Sec. 2.2 General Issues of Approximation in Value Space

done by brute force, through exhaustive computation and comparison of the corresponding Q-factors. 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 (2.6)]. In particular, the Q-factors of different control choices \( u_k \in U_k(x_k) \) are uncoupled and their computation 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.

Also, we will later discuss a major type of simplification for the case where the control \( u_k \) consists of multiple components, i.e., \( u_k = (u^1_k, \ldots, u^m_k) \). We refer to this as the multagent case, in reference to the situation where each component is chosen by a separate autonomous agent. We will consider this case starting in Section 2.6, where we will introduce “one-agent-at-a-time” minimization schemes that reduce dramatically the amount of computation needed for lookahead minimization.

Finally, one more possibility to simplify or bypass the one-step lookahead minimization (2.6) 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 ap-
proximate Q-factors
\[
E \left\{ g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}(f_k(x_k, u_k, w_k)) \right\},
\]
(2.8)
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 (2.8) 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. (2.6). The steps are as follows:

---

**Summary of Q-Factor Approximation Based on Cost Function Approximation**

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

(a) Use the system equation or a simulator to collect a large number of “representative” quadruplets \((x^s_k, u^s_k, x^s_{k+1}, g^s_k)\), and corresponding Q-factors

\[
\beta^s_k = g^s_k + \tilde{J}_{k+1}(x^s_{k+1}), \quad s = 1, \ldots, q.
\]
(2.9)

Here \(x^s_{k+1}\) is the next state
\[
x^s_{k+1} = f_k(x^s_k, u^s_k, w^s_k)
\]
Sec. 2.2 General Issues of Approximation in Value Space

Figure 2.2.2 Schematic illustration of Q-factor approximation, assuming approximate cost functions $\hat{J}_{k+1}$ 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 $x^s_{k+1}$ and cost sample $g^s_k$. These correspond to a disturbance $w^s_k$ according to

$$x^s_{k+1} = f_k(x^s_k, u^s_k, w^s_k), \quad g^s_k = g_k(x^s_k, u^s_k, w^s_k).$$

The sample Q-factors $\beta^s_k$ are generated according to Eq. (2.9), and are used in the least squares regression (2.10) to yield a parametric Q-factor approximation $\hat{Q}_k$ and the policy implementation (2.11).

\[\beta^s_k = g^s_k + \hat{J}_{k+1}(x^s_{k+1})\]

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 (2.10) and the Q-factor minimization (2.11). Using the simulator to collect the samples (2.9) 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 $\beta_k^*$ [cf. Eq. (2.9)], and another to compute $\tilde{Q}_k$ through the regression (2.10). The approximation methods to obtain $\tilde{J}_{k+1}$ and $\tilde{Q}_k$ may be unrelated.

(3) The policy $\tilde{\mu}_k$ obtained through the minimization (2.11) is not the same as the one obtained through the minimization (2.6). 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 (2.10). 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. (2.10), 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. (2.10) to determine $\bar{r}_k$, 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 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
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 $\tilde{J}_{k+\ell}(x_k)$ 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 $\tilde{\mu}_k(x_k) = \tilde{u}_k$.

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 $\tilde{J}_{k+\ell}(x_k)$ 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

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

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
as 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

\[
x^*_{k+1} = f_k(x^*_k, u^*_k, w^*_k),
\]

and corresponding transition cost

\[
g_k(x^*_k, u^*_k, w^*_k),
\]

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,

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

cf. Section 2.2.5 and Eq. (2.10). The parametric Q-factor function thus obtained can be used for approximate implementation of the rollout policy; cf. Eq. (2.11). 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; c.f. 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 \( (x^*_k, u^*_k) \), \( s = 1, \ldots, q \), i.e., for each \( s \), \( u^*_k \) is the rollout control at state \( x^*_k \). We can then introduce a parametric family of policies \( \mu_k(x_k, r_k) \), where
$r_k$ is a parameter vector, and choose the parameter $r_k$ by solving the least squares/regression problem

$$\min_{r_k} \sum_{s=1}^{q} \left\| u_k^s - \tilde{\mu}_k(x_k^s, r_k) \right\|^2$$  \hspace{1cm} (2.12)

[cf. Eq. (2.2)]. This is approximation in policy space on top of approximation in value space. 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. (2.12).

**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 off-line 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 sequence of states \( \{x_{k+1}, \ldots, x_N\} \) and controls \( \{u_{k+1}, \ldots, u_{N-1}\} \) such that

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

and the corresponding cost

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

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

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

† For 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).
Figure 2.3.2 Schematic illustration of rollout with one-step lookahead for a deterministic problem. At state $x_k$, for every pair $(x_k, u_k)$, $u_k \in U_k(x_k)$, the base heuristic generates a Q-factor $\tilde{Q}_k(x_k, u_k)$ (cf. Eq. (2.14)), and selects the control $\tilde{\mu}_k(x_k)$ with minimal Q-factor.

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

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

(2.13)

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

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

(2.14)

see Fig. 2.3.2. The rollout process defines a suboptimal policy

$$\tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\},$$

referred to as the rollout policy, where for each $x_k$ and $k$, $\tilde{\mu}_k(x_k)$ is the control produced by the Q-factor minimization (2.13).

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 $N$.

**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
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

\[
g(c_0, c_1) + \cdots + g(c_{N-2}, c_{N-1}) + g(c_{N-1}, c_0). \tag{2.15}
\]

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. (2.15). 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. The overall corresponding computation for the rollout solution is bounded by a polynomial in \( N \), and is much smaller than the computation for exact DP.

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
\[
\{x_{k+1}, u_{k+1}, \ldots, x_N\}
\]
starting from state \( x_{k+1} \).

In other words, the base heuristic is sequentially consistent if it “stays the course”: when the starting state \( x_k \) is moved forward to the next state \( x_{k+1} \) of its state trajectory, the heuristic will not deviate from the remainder of the trajectory.
As an example, the reader may verify that the nearest neighbor heuristic described in the traveling salesman Example 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

$$J_{k, \tilde{\pi}}(x_k) \leq H_k(x_k), \quad \text{for all } x_k \text{ and } k, \quad (2.16)$$

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 $\bar{u}_k$ be the control applied by the base heuristic at $x_k$. Then we have

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

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

$$= \min_{u_k \in \mathcal{U}_k(x_k)} \left[ g_k(x_k, u_k) + H_{k+1}(f_k(x_k, u_k)) \right] \quad (2.17)$$

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

$$= H_k(x_k),$$

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.

(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 (2.16). 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

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

[cf. Eqs. (2.14)], and $H_{k+1}(f_k(x_k, u_k))$ denotes the cost of the trajectory 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

$$
\min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k) \leq H_k(x_k). \quad (2.18)
$$

In words, the sequential improvement property (2.18) states that

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

Note that when the heuristic is sequentially consistent it is also sequentially improving. This follows from the preceding relation, since for a sequentially consistent heuristic, the heuristic cost at $x_k$ is equal to the Q-factor of the control $\tilde{\pi}_k$ that the heuristic applies at $x_k$,

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

which is greater or equal to the minimal Q-factor at $x_k$. This implies Eq. (2.18). 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

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

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

Proof: Follows from the calculation of Eq. (2.17), by replacing the last two steps (that rely on sequential consistency) with Eq. (2.18). 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

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

the rollout algorithm rejects the optimal control $u_0^*$ in favor of the alternative control $\tilde{u}_0$. The inequality above will occur if the base heuristic chooses $\tilde{u}_1$ at $x_1^*$ (there is nothing to prevent this from happening, since the base heuristic is arbitrary), and moreover the cost $g_1(x_1^*, u_1) + g_2(\tilde{x}_2)$, which is equal to $H_1(x_1^*)$ is high enough.
Sec. 2.3  Rollout and the Policy Improvement Principle

Let us also verify that if the inequality (2.20) holds then the heuristic is not sequentially improving at $x_0$, i.e., that

$$H_0(x_0) < \min \{ g_0(x_0, u_0^*) + H_1(x_1^*), g_0(x_0, \tilde{u}_0) + H_1(\tilde{x}_1) \}.$$  

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

$$g_0(x_0, \tilde{u}_0) + H_1(\tilde{x}_1),$$

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

The preceding example 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 Base Heuristics - Parallel Rollout

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

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

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

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

where

$$
\hat{Q}_k^m(x_k, u_k) = g_k(x_k, u_k) + C(\hat{T}_k^m)
$$

is the cost of the trajectory $(x_k, u_k, \hat{T}_k^m)$. Note that the Q-factors of the different base heuristics can be computed independently and in parallel. In view of this fact, the rollout scheme just described is sometimes referred to as parallel rollout.

An interesting property, which can be readily verified by using the definitions, is that if all the base heuristics are sequentially improving, the same is true for the superheuristic. Indeed, let us write the sequential improvement condition (2.18) for each of the base heuristics

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

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

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

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

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

which is precisely the sequential improvement condition for the superheuristic.

† 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. In fact one may use a collection of multiple heuristics tailored to each state space subset, and at each state, select out of all the heuristics that apply, the one that yields minimum cost.
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

$$\mathcal{T}_k = \{x_0, u_0, \ldots, u_{k-1}, x_k\}$$

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

$$\mathcal{T}_k = \{x_k, \bar{u}_k, x_{k+1}, \bar{u}_{k+1}, \ldots, \bar{u}_{N-1}, x_N\}$$

with corresponding cost

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

The tentative best trajectory is such that $\mathcal{P}_k \cup \mathcal{T}_k$ 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 $\bar{u}_k$ or $\tilde{u}_k$ and moves to $\bar{x}_{k+1}$ or to $\tilde{x}_{k+1}$.

† 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

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

and the tentative best trajectory

$$\overline{T}_k = \{x_k, \pi_k, x_{k+1}, \pi_{k+1}, \ldots, \pi_{N-1}, \pi_N\}$$

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

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

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

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

Otherwise we add $(\pi_k, \pi_{k+1})$ to the permanent trajectory and set the tentative best trajectory to

$$\overline{T}_{k+1} = \{\pi_{k+1}, \pi_{k+1}, \ldots, \pi_{N-1}, \pi_N\}.$$

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 $\pi_{k+1}$. $\tilde{x}_{k+1}$, respectively. In particular, if $C(\overline{T}_k) \leq C(\tilde{T}_k)$ the algorithm sets the next state and corresponding tentative best trajectory to

$$x_{k+1} = \pi_{k+1}, \quad \overline{T}_{k+1} = \{\pi_{k+1}, \pi_{k+1}, \ldots, \pi_{N-1}, \pi_N\},$$

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

$$x_{k+1} = \tilde{x}_{k+1}, \quad \tilde{T}_{k+1} = \{\tilde{x}_{k+1}, \tilde{u}_{k+1}, \ldots, \tilde{u}_{N-1}, \tilde{x}_N\}. $$
In other words, the fortified rollout at $x_k$ follows the current tentative best trajectory $\overline{T}_k$ unless a lower cost trajectory $\tilde{T}_k$ is discovered by running the base heuristic from all possible next states $x_{k+1}$. It follows that at every state the 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.,

$$u_k = (u^1_k, \ldots, u^m_k), \quad U_k(x_k) = U^1_k(x_k) \times \cdots \times U^m_k(x_k),$$

and to take advantage of distributed computation, it may be attractive to decompose the optimization over $u_k$ in the rollout algorithm,

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

into an (approximate) parallel optimization over the components $u^i_k$ (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.

† Such a situation arises in the context of multiagent rollout for deterministic discrete optimization problems; see Sections 2.6-2.8.
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 \( x_{k+2} \). We select the state, say \( \tilde{x}_{k+2} \), that is associated with minimum cost, compute the controls \( \tilde{u}_k \) and \( \tilde{u}_{k+1} \) that lead from \( x_k \) to \( \tilde{x}_{k+2} \), and choose \( \tilde{u}_k \) as the next rollout control and \( x_{k+1} = f_k(x_k, \tilde{u}_k) \) as the next state; see Fig. 2.3.6.

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 trajec-
tories 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 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.

Let us also 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.

Finally, let us note that together with the variations just described, rollout encompasses a very broad collection of approximation in value space schemes. Indeed any \( \ell \)-step lookahead method of the type discussed in Section 2.1.1, with \( \ell > 1 \), can be viewed as a truncated rollout scheme with one-step lookahead, where the base policy is defined by the \((\ell - 1)\)-step minimization from state \( x_{k+1} \) to state \( x_{k+\ell} \), and the terminal cost approximation is \( \tilde{J}_{k+\ell}(x_{k+\ell}) \) (cf. Fig. 2.1.2). This viewpoint may be helpful if concepts developed in the context of rollout, such as sequential improvement, fortified rollout, multiagent rollout (to be discussed in Section 2.6), or constrained rollout (to be discussed in Section 2.7), can be fruitfully brought to bear on a multistep lookahead approximation in value space method that is seemingly unrelated to rollout. For example, in Section 2.5 we will relate rollout concepts of sequential improvement to Lyapunov stability ideas for analysis of model predictive control schemes.
2.3.4 Rollout with an Expert

We will now consider a rollout algorithm for discrete deterministic optimization for the case where we do not know the cost function of the problem. Instead we have access to a human or software “expert” who can rank any two feasible solutions, without assigning numerical values to them.

Similar to Section 1.3.1, the problem is to select a sequence $u = (u_0, \ldots, u_{N-1})$, with each $u_k$ belonging to a given finite set $\mathcal{U}_k$, so as to minimize a function $G(u)$ subject to the constraint $u \in \mathcal{U}_0 \times \cdots \times \mathcal{U}_{N-1}$.†

We assume the following:

† For simplicity we assume that the constraints on the components of $u$ are independent of each other, but it is also possible to deal with more general
(a) We can use an “expert” who can compare any two feasible sequences \( u \) and \( \overline{u} \), in the sense that he/she can determine whether \( G(u) > G(\overline{u}) \), or \( G(u) \leq G(\overline{u}) \).

(b) A base heuristic with the following property is available: Given any \( k < N - 1 \), and a partial solution \((u_0, \ldots, u_k)\), it generates a complete feasible solution by concatenating the given partial solution \((u_0, \ldots, u_k)\) with a sequence \((\tilde{u}_{k+1}, \ldots, \tilde{u}_{N-1})\). This complete feasible solution is denoted \( S_k(u_0, \ldots, u_k) = (u_0, \ldots, u_k, \tilde{u}_{k+1}, \ldots, \tilde{u}_{N-1}) \).

It can be seen that the deterministic rollout algorithm of Section 2.3.1 can be applied to this problem, even though the cost function \( G \) is unknown. The reason is that the rollout algorithm uses the cost function only as a means of ranking complete solutions in terms of their cost. Thus, if the ranking of any two solutions can be revealed by the expert, this is all that is needed.† In particular, the rollout algorithm of Section 2.3.1 can be described as follows:

We start with an artificial empty solution, and at the typical step, given the partial solution \((u_0, \ldots, u_k)\), \( k < N - 1 \), we generate all possible one-step-extended solutions \((u_0, \ldots, u_k, u_{k+1})\), \( u_{k+1} \in U_{k+1} \), and use the expert to rank the finite set of complete solutions \( S_{k+1}(u_0, \ldots, u_k, u_{k+1}) \), \( u_{k+1} \in U_{k+1} \).

We then select the component \( u_{k+1} \) that is ranked best by the expert, extend the partial solution \((u_0, \ldots, u_k)\) by adding \( u_{k+1} \), and repeat with the new partial solution \((u_1, \ldots, u_k, u_{k+1})\).

Except for the (mathematically inconsequential) use of an expert rather than a cost function, the preceding rollout algorithm can be viewed as a special case of the one of Section 2.3.1. As a result several of the rollout variants that we have discussed in that section (fortified rollout, rollout with multiple heuristics, and rollout that maintains multiple trajectories) can also be easily adapted to the context of the present section.

† Note that for this to be true, it is important that the problem be deterministic, and that the expert ranks solutions using some underlying (though unknown) cost function. In particular, the expert’s rankings should have a transitivity property: if \( u \) is ranked better than \( u' \) and \( u' \) is ranked better than \( u'' \), then \( u \) is ranked better than \( u'' \).
Learning to Emulate the Expert

Let us now consider the case where an expert is not available but can be emulated by training with the use of data. In particular, suppose that we are given a set of control sequence pairs \((u^s, \overline{u}^s), s = 1, \ldots, q\), with

\[
G(u^s) > G(\overline{u}^s), \quad s = 1, \ldots, q,
\]

(2.21)

which we can use for training. Such a set may be obtained in a variety of ways, including querying the expert. We may then train a parametric approximation architecture such as a neural network to produce a function \(\hat{G}(u, r)\), where \(r\) is a parameter vector, and use this function in place of the unknown \(G(u)\) to implement the preceding rollout algorithm.

A method, known as comparison training, has been suggested for this purpose, and has been used in a variety of game contexts, including backgammon and chess by Tesauro [Tes89b], [Tes01]. Briefly, given the training set of pairs \((u^s, \overline{u}^s), s = 1, \ldots, q\), which satisfy Eq. (2.21), we generate for each \((u^s, \overline{u}^s)\), two solution-cost pairs

\[
(u^s, 1), (\overline{u}^s, -1), \quad s = 1, \ldots, q.
\]

A parametric architecture \(\hat{G}(\cdot, r)\), involving a parameter vector \(r\), such as a neural network, is then trained by regularized regression with these data to produce an approximation \(\hat{G}(\cdot, \bar{r})\) to be used in place of \(G(\cdot)\) in a rollout scheme. We refer to Chapter 3 and to the aforementioned papers by Tesauro for implementation details of the regression procedure. See also Section 3.4 on parametric approximation in policy space through the use of classification methods.

### 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 \(J_{k, \hat{\pi}}(x_k)\) the cost corresponding to starting the rollout algorithm at state \(x_k\). We claim that

\[
J_{k, \hat{\pi}}(x_k) \leq J_{k, \pi}(x_k), \quad \text{for all } x_k \text{ and } k.
\]
We prove this inequality by induction similar to the deterministic case [cf. Eq. (2.17)]. 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 $x_k$,

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

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.
(d) 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” $p_iv_i/(1-p_i)$, 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

$$p_{i_1} \left( v_{i_1} + p_{i_2} \left( v_{i_2} + p_{i_3} \left( \cdots + p_{i_N} v_{i_N} \right) \cdots \right) \right).$$

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 by Bertsekas and Castaño [BeC99].

2.4.1 Simulation-Based Implementation of the Rollout Algorithm

A conceptually straightforward way to compute the rollout control at a
Sec. 2.4 Stochastic Rollout and Monte Carlo Tree Search

Given state $x_k$ and time $k$ is to consider each possible control $u_k \in U_k(x_k)$, and to generate a “large” number of simulated trajectories of the system starting from $(x_k, u_k)$. Thus a simulated trajectory is obtained from

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

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 $u_k \in U_k(x_k)$, denoted by $\hat{Q}_{k,\pi}(x_k, u_k)$. We then compute the (approximate) rollout control $\hat{\mu}_k(x_k)$ with the minimization

$$\hat{\mu}_k(x_k) \in \arg\min_{u_k \in U_k(x_k)} \hat{Q}_{k,\pi}(x_k, u_k). \quad (2.22)$$

**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 [TeG 96]; 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.
Rollout and Policy Improvement Chap. 2

Possible Moves Average Score by Monte-Carlo Simulation

Current Position and Dice Roll

*Best Score*

Figure 2.4.1 Illustration of rollout for backgammon. At a given position and roll of the dice, the set of all possible moves is generated, and the outcome of the game for each move is evaluated by “rolling out” (simulating to the end) many games using a suboptimal/heuristic backgammon player (the TD-Gammon player was used for this purpose in [TeG96]), and by Monte-Carlo averaging the scores. The move that results in the best average score is selected for play.

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, with both players using some base policy? The answer is to view the game as a (one-player) optimal control problem, where one of the two players passively uses the base policy exclusively (TD-Gammon in the present example). The other player takes the role of the optimizer, and actively tries to improve on his base policy (TD-Gammon) by using rollout. Thus “policy improvement” in the context of the present example means that when playing against a TD-Gammon opponent, the rollout player achieves a better score on the average than if he/she were to play with the TD-Gammon strategy. In particular, the theory does not guarantee that a rollout player that is trained using TD-Gammon for both players will do better than TD-Gammon would against a non-TD-Gammon opponent. While this is a plausible practical hypothesis, it is one that can only be tested empirically.

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., the website 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

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

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

An alternative approach is to approximate the Q-factor difference $\tilde{Q}_{k,\pi}(x_k, u_k) - \tilde{Q}_{k,\pi}(x_k, \hat{u}_k)$ by sampling the difference

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

(2.23)

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

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

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 (2.23), which involves a common disturbance $w_k$ for $u_k$ and $\hat{u}_k$, may be far more accurate than the one obtained by differencing samples of $C_k(x_k, u_k, w_k)$ and
\( C_k(x_k, \hat{u}_k, \hat{w}_k) \), which involve two different disturbances \( w_k \) and \( \hat{w}_k \). Indeed, by introducing the zero mean sample errors
\[
D_k(x_k, u_k, w_k) = C_k(x_k, u_k, w_k) - \hat{Q}_{k,\pi}(x_k, u_k),
\]
it can be seen that the variance of the error in estimating \( \hat{Q}_{k,\pi}(x_k, u_k) - \hat{Q}_{k,\pi}(x_k, \hat{u}_k) \) with the former method will be no larger than with the latter method if and only if
\[
E_{w_k, \hat{w}_k} \left\{ \left| D_k(x_k, u_k, w_k) - D_k(x_k, \hat{u}_k, \hat{w}_k) \right|^2 \right\} \\
\geq E_{w_k} \left\{ \left| D_k(x_k, u_k, w_k) - \hat{D}_k(x_k, u_k, w_k) \right|^2 \right\}.
\]
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
\[
E\{D_k(x_k, u_k, w_k)\} \geq 0; \quad (2.24)
\]
i.e., the errors \( D_k(x_k, u_k, w_k) \) and \( \hat{D}_k(x_k, \hat{u}_k, w_k) \) 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 (2.24) 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
\[
E \left\{ \left| D_k(x_k, u_k, w_k) - D_k(x_k, \hat{u}_k, w_k) \right|^2 \right\} \leq \gamma E \left\{ \left| D_k(x_k, u_k, w_k) \right|^2 \right\}.
\]
Then we have, by using the generic relation \( ab \geq a^2 - |a| \cdot |b - a| \) for two scalars \( a \) and \( b \),
\[
D_k(x_k, u_k, w_k)D_k(x_k, \hat{u}_k, w_k) \\
\geq \left| D_k(x_k, u_k, w_k) \right|^2 \\
- \left| D_k(x_k, u_k, w_k) \cdot \left| D_k(x_k, \hat{u}_k, w_k) - D_k(x_k, u_k, w_k) \right| \right|
\]
from which we obtain
\[
E\{D_k(x_k, u_k, w_k)\}D_k(x_k, \hat{u}_k, w_k) \\
\geq E \left\{ \left| D_k(x_k, u_k, w_k) \right|^2 \right\} \\
- E \left\{ \left| D_k(x_k, u_k, w_k) \right| \cdot \left| D_k(x_k, \hat{u}_k, w_k) - D_k(x_k, u_k, w_k) \right| \right\} \\
\geq E \left\{ \left| D_k(x_k, u_k, w_k) \right|^2 \right\} - \frac{1}{2} E \left\{ \left| D_k(x_k, u_k, w_k) \right|^2 \right\} \\
- \frac{1}{2} E \left\{ \left| D_k(x_k, \hat{u}_k, w_k) - D_k(x_k, u_k, w_k) \right|^2 \right\} \\
\geq 1 - \frac{1}{2} \gamma E \left\{ \left| D_k(x_k, u_k, w_k) \right|^2 \right\},
\]
where for the second inequality we use the generic relation 

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

for two scalars \(a\) and \(b\), and for the third inequality we use Eq. (2.25).

Thus, under the assumption (2.25), the condition (2.24) holds and guarantees that by averaging cost difference samples rather than differencing (independently obtained) averages of cost samples, the simulation error variance does not increase.

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

\[A_{k,\pi}(x_k, u_k) = Q_{k,\pi}(x_k, u_k) - \min_{u_k \in U(x_k)} Q_{k,\pi}(x_k, u_k).\]

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 with multistep lookahead, one needs to maintain a lookahead tree, which is expanded as the relevant Q-factors are evaluated by simulation, and which balances the competing desires of exploitation and exploration (generate and evaluate controls that seem most promising in terms of performance versus assessing the potential of inadequately explored controls). Ideas that were developed in the context of multiarmed bandit problems have played an important role in the construction of this type of MCTS procedures (see the end-of-chapter references).

**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 \( \bar{u}_k \) that minimizes an approximate Q-factor

\[
\tilde{Q}_k(x_k, u_k) = E \left\{ Q_k(x_k, u_k, w_k) + J_{k+1} \left( f_k(x_k, u_k, w_k) \right) \right\}
\]

over \( u_k \in U_k(x_k) \), with \( Q_k(x_k, u_k) \) 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

\[
\delta(i_\ell = i) = \begin{cases} 
1 & \text{if } i_\ell = i, \\
0 & \text{if } i_\ell \neq i.
\end{cases}
\]
Sec. 2.4 Stochastic Rollout and Monte Carlo Tree Search

Factors

Current State

$Q_{1,n} + R_{1,n}$

Simulation

$Q_{2,n} + R_{2,n}$

Simulation

$Q_{3,n} + R_{3,n}$

Simulation

Sample Q-Factors

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).

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

$$T_{i,n} + R_{i,n}$$

of two indexes: an exploitation index $T_{i,n}$ and an exploration index $R_{i,n}$. Usually the exploitation index is chosen to be the empirical mean $Q_{i,n}$; see Fig. 2.4.2. 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 than the one above, and depends on the depth of lookahead (see Silver et al. [SHS17]).

Sampling policies for MCTS with multistep lookahead are based on similar sampling ideas to the case of one-step lookahead. A simulated trajectory is run from a node \( i \) of the lookahead tree that minimizes the sum \( T_{i,n} + R_{i,n} \) of an exploitation index and an exploration index. There are several schemes of this type, but the details are beyond our scope (see the end-of-chapter references).

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

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. (2.22)]. 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

† The paper [ACF02] refers to the rule given here as UCB1 and credits its motivation to the paper by Agrawal [Agr95].
Sec. 2.4  

Stochastic Rollout and Monte Carlo Tree Search

mapping onto a single control. In particular, the AlphaGo and AlphaZero programs use MCTS to generate and use for training purposes randomized policies, which specify at each board position the probabilities with which the various moves are selected.

A randomized policy can be used as a base policy in a rollout context in exactly the same way as a deterministic policy: for a given state $x_k$, we just generate sample trajectories and associated sample Q-factors, 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 $\hat{\mu}_k(x_k)$ that is “best” according to the results of the rollout [cf. Eq. (2.22)]. 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 $\hat{\mu}_k(x_k)$ from its base policy level, while proportionally decreasing the probabilities of the other controls.

The use of MCTS provides a related method to “improve” a randomized policy. In the process of 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 obtain the rollout randomized policy by moving the probabilities of the base policy in the direction suggested by the frequency counts, i.e., increase the probability of high-count controls and reduce the probability of the others. This type of policy improvement is reminiscent of gradient-type methods, and has been successful in some contexts; see the end-of-chapter references for such policy improvement implementations in AlphaGo, AlphaZero, and other 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, $u_k = (u^1_k, \ldots, u^m_k)$, with a separable control constraint $u^\ell_k \in U^\ell_k(x_k)$, $\ell = 1, \ldots, m$. 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 $U^\ell_k(x_k)$. 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 $nm$. In Section 2.6 we will also discuss the possibility of some of the agents computing their Q-factors 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 separate 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.
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 $x_k$.

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

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

(2.26)

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

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

(2.27)

with $H_{k+1}(x_{k+1})$ being the cost of the base heuristic starting from state $x_{k+1}$ [cf. Eq. (2.14)]. 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. (2.27) is also differentiable with respect to $u_k$, and its minimization (2.26) 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 (2.26)-(2.27) can be implemented by solving the $\ell$-stage deterministic optimal control problem,
which seamlessly concatenates the first stage minimization over $u_k$ [cf. Eq. (2.26)], with the $(\ell - 1)$-stage minimization of the base heuristic; see Fig. 2.5.1. This $\ell$-stage problem may be solvable on-line by standard continuous spaces nonlinear programming or optimal control methods.† An major paradigm of methods of this type, model predictive control, arises in control system design and is discussed in the next section. The following is a simple example of an important class of inventory storage and supply chain management processes.

**Example 2.5.1 (Supply Chain Management)**

Let us consider an example of a supply chain system, where a certain item is produced at a production center and fulfilled at a retail center. Stock of the item is shipped from the production center to the retail center, where it arrives with a delay of $\tau \geq 1$ time units, and is used to fulfill a known stream of demands $d_k$ over an $N$-stage horizon; see Fig. 2.5.2. We denote:

- $x_1^k$: The stock at hand at the production center at time $k$.
- $x_2^k$: The stock at hand at the retail center at time $k$, and used to fulfill demand (both positive and negative $x_2^k$ are allowed; a negative value indicates that there is backordered demand).
- $u_1^k$: The amount produced at time $k$.
- $u_2^k$: The amount shipped at time $k$ (and arriving at the retail center $\tau$ time units later).

The state at time $k$ is the stock available at the production and retail centers, $x_1^k, x_2^k$, plus the stock amounts that are in transit and have not yet arrived at the retail center $u_2^k-\tau-1, \ldots, u_2^k-1$.‡ The control $u_k = (u_1^k, u_2^k)$ is chosen from some constraint set that may depend on the current state, and is subject to production capacity and transport availability constraints. The system equation is

$$x_{k+1}^1 = x_k^1 + u_k^1 - u_k^2, \quad x_{k+1}^2 = x_k^2 + u_{k-\tau}^2 - d_k.$$

The cost at time $k$ consists of three components: a production cost that depends on $x_1^k$ and $u_1^k$, a transportation cost that depends on $u_2^k$, and

---

† 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.

‡ For a large delay $\tau$, the size of the state space can become very large. However, rollout schemes are not affected much by the size of the state space. For this reason, rollout may be very well suited for problems involving delayed effects of past states and controls.
Figure 2.5.2. Illustration of a simple supply chain system for Example 2.5.1.

A fulfillment cost that depends on $x^2_k$ [which includes positive costs for both excess inventory (i.e., $x^2_k > 0$) and for backordered demand (i.e., $x^2_k < 0$)]. The precise forms of these cost components are immaterial for the purposes of this example.

Here the control vector $u_k$ is often continuous (or a mixture of discrete and continuous components), so it may be essential for the purposes of rollout to use the continuous optimization framework of this section. In particular, at the current state (including the known stock in transit) some heuristic optimization is used to determine the stream of future production and shipment levels, and the first component of this stream is used as the control applied by rollout. As an example we may use as base policy one that brings the retail inventory to some target value $\ell$ stages ahead, and keeps it at that value in the remaining periods. This is a nonlinear programming or mixed integer programming problem that may be solvable with available software far more efficiently than by a discretized form of DP.

Note also an additional benefit of rollout: it can readily deal with online replanning, which is necessary when unexpected demand changes, and production or transport equipment failures occur. Moreover, it can accommodate stochastic demands by using forms of certainty equivalence.

### 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 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 car may be constrained by the presence of obstacles and hardware limitations (see Fig. 2.5.3). The solution obtained from a linear-quadratic model may not be suitable for such a problem, because quadratic penalties treat constraints “softly” and may produce trajectories that violate the constraints.

These inadequacies of the linear-quadratic formulation have motivated a methodology, called \textit{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.

The ideas of MPC were developed independently of the approximate DP/RL methodology and rollout in particular. However, the two fields are closely related, and there is much to be gained from understanding one field within the context of the other, as the subsequent development will aim to show.

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 prob-
lems 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 finite-dimensional vectors. 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 assume that the system can be kept at the
origin at zero cost, i.e.,

\[ f_k(0, \pi_k) = 0, \quad g_k(0, \pi_k) = 0 \quad \text{for some control } \pi_k \in U_k(0). \]

We also impose (possibly time-varying) state and control constraints

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

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.

Note that there are no restrictions on the nature of the sets \( X_k \) and
\( U_k(x_k) \): they are arbitrary, within the corresponding state and control
spaces. In particular, they can be continuous/infinite or discrete/finite.
Moreover we allow time varying constraints, stage costs, and system func-
tions.

The MPC Algorithm

Let us describe the MPC algorithm for the deterministic problem just de-
scribed. At the current state \( x_k \):

(a) MPC solves an \( \ell \)-step lookahead version of the problem, which re-
quires that \( x_{k+\ell} = 0 \). We assume that the positive integer \( \ell \) satisfies
a condition that guarantees the feasibility of this problem (see the
constrained controllability condition that follows).

(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

$$
\min_{u_i, i = k, \ldots, k+\ell-1} \sum_{i = k}^{k+\ell-1} g_i(x_i, u_i), \quad (2.28)
$$

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 \( \{\hat{u}_k, \hat{u}_{k+1}, \ldots, \hat{u}_{k+\ell-1}\} \) be a corresponding optimal control sequence. The MPC algorithm applies at stage $k$ the first component $\hat{u}_k$ of this sequence, and discards the remaining components; see Fig. 2.5.4.†

Note that we have not excluded the possibility that $U_k(x_k)$ has a discrete character, in which case the $\ell$-stage MPC problem (2.28) may be an integer programming problem. We assume the following.

### Constrained Controllability Condition

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

$$
u_k \in U_k(x_k), \ x_{k+1} \in X_{k+1}, \ u_{k+1} \in U_{k+1}(x_{k+1}) \ldots, \ x_{k+\ell-1} \in X_{k+\ell-1}, \ u_{k+\ell-1} \in U_{k+\ell-1}(x_{k+\ell-1}).
$$

† 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. (2.28)] 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.

Sequential Improvement 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 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. (2.18)].†

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 (2.30) is satisfied by the MPC algorithm. Indeed, from the sequential improvement condition (2.29), we have

\[
g_k(x_k, u_k) + H_{k+1}(x_{k+1}) \leq H_k(x_k), \quad k = 1, 2, \ldots.
\]

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

\[
H_{K+1}(x_{K+1}) + \sum_{k=0}^{K} g_k(x_k, u_k) \leq g_0(x_0, u_0) + H_1(x_1).
\]

Since (in view of the nonnegativity of the cost per stage) we have

\[
0 \leq H_{K+1}(x_{K+1}),
\]

\[†\] 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).
it follows that
\[ \sum_{k=0}^{K} g_k(x_k, u_k) \leq g_0(x_0, u_0) + H_1(x_1), \quad K \geq 1, \]
and taking the limit as \( K \to \infty \), we obtain
\[ \sum_{k=0}^{\infty} g_k(x_k, u_k) \leq g_0(x_0, u_0) + H_1(x_1). \] (2.32)
The expression
\[ g_0(x_0, u_0) + H_1(x_1) \]
in the right side above is the optimal cost of transfer from \( x_0 \) to \( x_\ell = 0 \) (i.e., the first \( \ell \)-stage problem solved by MPC). Since this transfer is feasible by the constrained controllability condition, the above expression is finite and the stability condition (2.30) is satisfied.

The line of analysis just provided was based on rollout ideas and the sequential improvement condition (2.31). However, it is also related to well known lines of Lyapunov stability analysis in control theory; see e.g., the end-of-chapter textbook references on MPC.

Example 2.5.2
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

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

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

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

so the sequential consistency condition of Section 2.4.1 is violated.

Regarding the choice of the horizon length $\ell$ 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.

### 2.5.2 Target Tubes and the Constrained Controllability Condition

We now return to the constrained controllability condition, which asserts that the state constraint sets $X_k$ are such that starting from anywhere within $X_k$, it is possible to drive to 0 the state of the system within some number of steps $\ell$, while staying within $X_m$ at each intermediate step $m = k + 1, \ldots, m = k + \ell - 1$. Unfortunately, this assumption masks some major complications.† In particular, the control constraint set may not

† Fundamentally, when the constrained controllability condition is not known to hold, we are essentially dealing with a constrained form of rollout, whereby the present control choice may affect the feasibility of future state constraints. Constrained rollout will be discussed in Section 2.7.
be sufficiently rich to compensate for natural instability tendencies of the system. As a result it may be impossible to keep the state within $X_k$ over a sufficiently long period of time, something that may be viewed as a form of instability. Here is an example involving an unstable system and inadequate control constraints.

**Example 2.5.3**

Consider the scalar linear system

$$x_{k+1} = 2x_k + u_k,$$

which is unstable, and the control constraint

$$|u_k| \leq 1.$$

Then if $0 < x_0 < 1$, it can be seen that by using the control $u_0 = -1$, the next state satisfies,

$$x_1 = 2x_0 - 1 < x_0,$$

and is closer to 0 than the preceding state $x_0$. Similarly, using controls $u_k = -1$, every subsequent state $x_{k+1}$ will get closer to 0 than $x_k$. Eventually, after a sufficient number of steps $\hat{k}$, with controls $u_k = -1$ for $k < \hat{k}$, the state $x_{\hat{k}}$ will satisfy

$$0 \leq x_{\hat{k}} \leq \frac{1}{2}.$$

Once this happens, the feasible control $u_{\hat{k}} = -2x_{\hat{k}}$ will drive the state $x_{\hat{k}+1}$ to 0.

Similarly, when $-1 < x_0 \leq 0$, by applying control $u_k = 1$ for a sufficiently large number of steps $\hat{k}$, the state $x_{\hat{k}}$ will be driven into the region $[-1/2, 0]$, and then the feasible control $u_{\hat{k}} = -2x_{\hat{k}}$ will drive the state $x_{\hat{k}+1}$ to 0.

Suppose now that the state constraint is of the form $X_k = [-\beta, \beta]$ for all $k$, and let us explore what happens for different values of $\beta$. The preceding discussion shows that if $0 < \beta < 1$ the constrained controllability assumption is satisfied, and for every initial state $x_0 \in X_0$, the states $x_k$ can be kept within $X_k$ and can be driven to 0 in a finite number $\ell$ of steps. The number $\ell$ depends on $\beta$, and in particular if $0 < \beta < 1/2$, $\ell$ can be taken equal to 1.

On the other hand, if $\beta \geq 1$, it is impossible to drive the state to 0 from every initial state $x_0 \in [1, \beta]$ without violating the constraint $|u_k| \leq 1$, so the constrained controllability assumption is violated. In fact if the initial state satisfies $|x_0| > 1$, the state trajectory diverges in the sense that $|x_k| \to \infty$ for any control sequence that satisfies the constraint $|u_k| \leq 1$; see Fig. 2.5.5. Thus if $\beta \geq 1$, either a larger control constraint set or an initial condition that is close to 0, or both, are needed to satisfy the constrained controllability condition.
Figure 2.5.5 Schematic illustration of state trajectories under MPC for Example 2.5.3. If the initial state lies within the set $(-1, 1)$ the constrained controllability condition is satisfied for sufficiently large $k$, and the MPC algorithm yields a stable controller. If the initial state lies outside this set, MPC cannot be implemented because the constrained controllability condition fails to holds. Moreover, the system is unstable starting from such an initial state. In this example, the largest reachable tube is $\{X, X, \ldots\}$ with $X = \{x \mid |x| \leq 1\}$.

The critical construction in the preceding example is to identify sets of states $\overline{X}_k \subset X_k$ such that starting from within $\overline{X}_k$ we are guaranteed to stay within the “tube” $\{\overline{X}_{k+1}, \overline{X}_{k+2}, \ldots, \overline{X}_N\}$ for all subsequent times with appropriate choices of control. Tubes of this type can serve as state constraints in MPC. Moreover, when the sets $\overline{X}_k$ are bounded, a guarantee that the state can stay within the tube amounts to a form of stability guarantee. In the remainder of this section, we will address the issue of how such tubes can be constructed.

Formally, a tube $\{\overline{X}_0, \overline{X}_1, \ldots, \overline{X}_N\}$ is just a sequence of subsets with $\overline{X}_k \subset X_k$ for all $k = 0, \ldots, N$. The tube is called reachable if it has the property that for every $k$ and $x_k \in \overline{X}_k$ there exists a $u_k \in U_k(x_k)$ such that $f_k(x_k, u_k) \in \overline{X}_{k+1}$. A reachable tube was also called an effective target tube in [Ber71], and for simplicity it will be called a target tube in this section; the latter name is widely used in the current literature.† If the original tube of state constraints $\{X_0, X_1, \ldots, X_N\}$ is not reachable, the constrained

† The concept of target tubes and finite and infinite time reachability in discrete-time systems was first formulated and analyzed in the author’s 1971 Ph.D. thesis ([Ber71], available on-line) and associated papers [BeR71], [Ber72], [BeR73], where various methods for constructing target tubes were given, for both deterministic and minimax/game problems, including easily characterized ellipsoids for linear systems. Reachability can be defined for finite horizon as well as infinite horizon problems. Reachability concepts were studied by several authors much later, particularly for MPC and related contexts, and have been generalized to continuous-time systems (see the end-of-chapter references).
controllability condition cannot be satisfied. In this case, it is necessary to compute a reachable tube to use as a set of state constraints in place of the original tube \( \{X_0, X_1, \ldots, X_N\} \). Thus obtaining a reachable tube is a prerequisite towards satisfying the constrained controllability assumption, and serves as the first step in the analysis and design of MPC schemes with state constraints.

Given an \( N \)-stage deterministic problem with state constraints \( x_k \in X_k, k = 0, \ldots, N \), we can obtain a reachable tube \( \{\overline{X}_0, \overline{X}_1, \ldots, \overline{X}_N\} \) by a recursive algorithm that starts with

\[
\overline{X}_N = X_N,
\]

and generates \( \overline{X}_k, k = 0, \ldots, N - 1 \), going backwards,

\[
\overline{X}_k = \{x_k \in X_k \mid \text{for some } u_k \in U_k(x_k) \text{ we have } f_k(x_k, u_k) \in \overline{X}_{k+1}\}.
\]

Generally, it is difficult to compute the sets \( \overline{X}_k \) of the reachable tube, but algorithms that produce inner approximations have been constructed. The author’s thesis [Ber71] and subsequent papers [BeR71], [Ber72], [BeR73], [Ber07], gave inner ellipsoidal approximations for both finite and infinite horizon problems with perfect and partial state information, which involve linear systems and ellipsoidal constraints. Other authors have developed polyhedral approximations; see the end-of-chapter references.

**Example 2.5.3 (continued)**

In this example, if the state constraints are

\[
X_k = \{x_k \mid |x_k| \leq 1\}, \quad k = 0, 1, \ldots, N,
\]

the tube \( \{X_k \mid k = 0, \ldots, N\} \) is reachable for any \( N \). However, this is not true for the constraint sets

\[
X_k = \{x_k \mid |x_k| \leq 2\}, \quad k = 0, 1, \ldots, N.
\]

For example for \( x_0 = 2 \) the next state \( x_1 = 2x_0 + u_k = 4 + u_k \) will not satisfy \( |x_1| \leq 2 \) for any of the feasible controls \( u_k \) with \( |u_k| \leq 1 \). Thus it is necessary to replace the tube of original constraints \( \{X_k \mid k = 0, \ldots, N\} \) with a reachable tube \( \{\overline{X}_k \mid k = 0, \ldots, N\} \). It can be verified that the largest tube that is reachable for some value of \( N \) is the one with

\[
\overline{X}_k = \{x_k \mid |x_k| \leq 1\}, \quad k = 0, 1, \ldots, N.
\]

Calculating a reachable tube is relatively easy for one-dimensional problems, but becomes complicated for multidimensional problems, where approximations are required in general. On the other hand, it can be seen that in order
for the MPC minimizations to be feasible for $\ell = 2$, the initial condition must satisfy $|x_0| \leq 3/4$.

There is a subtle (and relatively insignificant) difference between reachability of the tube $\{X_k \mid k = 0, \ldots, N\}$ and satisfying the constrained controllability condition for sufficiently large $\ell$. The latter implies the former, but the converse is not true, as the preceding example illustrates. In particular, if $X_k$ is given by Eq. (2.33), the tube is reachable, but the constrained controllability assumption is not satisfied because starting at the boundary of $X_k$, we cannot drive the state to 0 in any number of steps, using controls $u_k$ with $|u_k| \leq 1$. By contrast, if we remove from $X_k$ the boundary points 1 and -1, the tube is still reachable while the constrained controllability condition is satisfied. Generally, except for boundary effects of this type, tube reachability typically implies the constrained controllability condition.

### 2.5.3 Model Predictive Control with Terminal Cost

In a common variation of MPC, the requirement of driving the system state to 0 in $\ell$ steps in the $\ell$-stage MPC problem (2.28), is replaced by a terminal cost $G_{k+\ell}(x_{k+\ell})$. Thus at state $x_k$, we solve the problem

$$\min_{u_i, i=k, \ldots, k+\ell-1} \left[ G_{k+\ell}(x_{k+\ell}) + \sum_{i=k}^{k+\ell-1} g_i(x_i, u_i) \right],$$

subject to the system equation constraints

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

and the state and control constraints

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

Then the stability analysis we gave earlier [cf. Eq. (2.32)] goes through provided the terminal cost approximation is such that the sequential improvement condition [cf. Eq. (2.29)]

$$\min_{u_k \in U_k(x_k)} \left[ g_k(x_k, u_k) + H_{k+1}(f_k(x_k, u_k)) \right] \leq H_k(x_k), \quad (2.34)$$

is satisfied, where $H_k(x_k)$ and $H_{k+1}(x_{k+1})$ are the optimal costs of the corresponding $(\ell-1)$-stage optimization problems that start at $x_k$ and $x_{k+1}$, and incur a terminal cost $G_{k+\ell-1}(x_{k+\ell-1})$ and $G_{k+\ell}(x_{k+\ell})$, respectively, after $\ell - 1$ steps.
It turns out that a sufficient condition for this is that the terminal cost functions satisfy

\[ G_{k+\ell}(x_{k+\ell-1}) \geq g_{k+\ell-1}(x_{k+\ell-1}, u_{k+\ell-1}) + G_{k+\ell}(f_{k+\ell-1}(x_{k+\ell-1}, u_{k+\ell-1})) \]

for all \( k, x_{k+\ell-1}, \) and \( u_{k+\ell-1} \in U_{k+\ell-1}(x_{k+\ell-1}) \). Indeed, by combining the definition of \( H_k(x_k) \) and Eq. (2.35), we have for some control sequence \((\bar{u}_k, \ldots, \bar{u}_{k+\ell-2})\) and corresponding state sequence \((\bar{x}_{k+1}, \ldots, \bar{x}_{k+\ell-1})\),

\[
H_k(x_k) = G_{k+\ell-1}(\bar{x}_{k+\ell-1}) + g_k(x_k, \bar{u}_k) + \sum_{i=k+1}^{k+\ell-2} g_i(\bar{x}_i, \bar{u}_i) \\
\geq G_{k+\ell}(\bar{x}_{k+\ell}) + g_k(x_k, \bar{u}_k) + \sum_{i=k+1}^{k+\ell-1} g_i(\bar{x}_i, \bar{u}_i) \\
\geq \min_{u_i, i=k,\ldots,k+\ell-1} \left[ g_k(x_k, u_k) + G_{k+\ell}(x_{k+\ell}) + \sum_{i=k}^{k+\ell-1} g_i(x_i, u_i) \right] \\
= \min_{u_k \in U_k(x_k)} \left[ g_k(x_k, u_k) + H_{k+1}(f_k(x_k, u_k)) \right],
\]

thus proving the sequential improvement condition (2.34). By can now apply the argument that we gave earlier to prove the stability condition (2.30).

### 2.5.4 Variants of Model Predictive Control

The MPC schemes that we have described are just the starting point for a broad methodology with many variations. For a simple example of an MPC variant, instead of aiming to drive the state to 0 in \( \ell \) steps, we may aim to reach a sufficiently small neighborhood of the origin, within which a stabilizing controller, designed by other methods, may be used. Still another possibility is to generate terminal cost function values by simulation of some policy.

We have emphasized so far the connection of MPC with rollout, and the attendant on-line implementation of the MPC controller through solution of optimization problem (2.28) at each state \( k \); this is sometimes referred to as on-line MPC. However, it is also possible to view MPC outside the rollout framework, simply as a one-step or multistep lookahead approximation in value space scheme. In particular, we can view the MPC scheme of the preceding section as approximation in value space with \( \ell \)-step lookahead, and terminal cost function approximation \( \hat{G}_{k+\ell}(x_{k+\ell}) \); cf. Fig. 2.1.2. In this way additional ideas described earlier come into play, such as off-line training an MPC controller via approximation in policy space or perpetual rollout (cf. Section 2.1.3). Sometimes schemes of this type are referred to as learning-based MPC.
A generic difficulty for the application of MPC arises when the control constraint $U_k(x_k)$ consists of both discrete/integer and continuous parts (even though the stability analysis of Section 2.5.1 is not affected by discrete constraint sets). Then the $\ell$-step MPC minimization problem may become hard to solve. However, in some cases we may be able to work around this difficulty, as indicated in the following example.

**Example 2.5.4 (MPC for Integer Constraints)**

Suppose that the control has a discrete component $d$, taking values $d^1, \ldots, d^m$, and a continuous component $v$. We will view this problem in the context of rollout, and consider a base policy for each possible (but fixed) value of $d$. An example of such a policy would be to drive the state from $x_{k+1}$ to 0 in $\ell - 1$ steps with suitable choices of the controls $v_{k+1}, \ldots, v_{k+\ell-1}$, but with the discrete part of the controls $d_{k+1}, \ldots, d_{k+\ell-1}$ fixed at the same value as $d_0$. Then at state $x_k$, MPC will solve $m$ problems, corresponding to the $m$ values $d^1, \ldots, d^m$, i.e., set $d_k = \cdots = d_{k+\ell-1} = d^i$, for each $i = 1, \ldots, m$, and then drive the state to 0 in $\ell$ steps. These $m$ problems are similar to the one of Eq. (2.28). Assuming that some of these problems are feasible, the discrete control component to be used at time $k$ should be the one that corresponds to the feasible problem with minimal optimal value.

There are obvious computational difficulties with this scheme when $m$ is very large. However, the solution of the $m$ optimization problems corresponding to the discrete components at a given state can be easily parallelized for faster solution. Another limitation is that the discrete component of the control is kept fixed in the $\ell$-step minimization problem, which may create feasibility issues for this problem. We return to these points later in our discussion of multiagent and constrained rollout in Section 2.6.2 and 2.7.

There are also variants of MPC that can deal with uncertainty and system disturbances. As an illustration, let us provide an example of a scheme that combines MPC with certainty equivalent control ideas.

**Example 2.5.5 (MPC for Stochastic Systems and Certainty Equivalent Approximation)**

Consider the stochastic system

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

with the expected cost of a policy $\pi = \{\mu_0, \ldots, \mu_{N-1}\}$ starting at $x_0$ defined by

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

cf. the framework of Section 1.2. We assume that for all $k$, there are state and control constraints of the form

$$x_k \in X_k, \quad u_k \in U_k(x_k).$$
Let us describe a rollout/MPC method that generalizes the one given earlier for deterministic problems. It uses assumed certainty equivalence to define the base policy over \( \ell - 1 \) steps, where \( \ell > 1 \) is some integer. In particular, at a given state \( x_k \in X_k \), this method first fixes the disturbances \( w_{k+1}, \ldots, w_{k+\ell-1} \) to some typical values. It then applies the control \( \tilde{u}_k \) that minimizes over \( u_k \in U_k(x_k) \) the Q-factor
\[
\tilde{Q}_k(x_k, u_k) = E\left\{ g_k(x_k, u_k, w_k) + H_{k+1}(f_k(x_k, u_k, w_k)) \right\},
\]
where \( H_{k+1}(x_{k+1}) \) is the optimal cost of the deterministic transfer from \( x_{k+1} \) to 0 in \( \ell - 1 \) steps with controls \( \tilde{u}_m \) from the sets \( U_m(x_m), m = k + 1, \ldots, k + \ell - 1 \), and with the disturbances \( w_{k+1}, \ldots, w_{k+\ell-1} \) fixed at their typical values. Here we assume that this transfer is possible, i.e., that some form of constrained controllability condition guarantees this.

A drawback of this MPC method is that it may not be well suited for on-line implementation because of the substantial amount of computation required at each state \( x_k \): the term \( H_{k+1}(x_{k+1}) \) must be calculated for each possible next state \( x_{k+1} = f_k(x_k, u_k, w_k) \). A possible remedy is to introduce further approximation in value space. To this end, we can generate a large number of sample states \( x^s_{k+1}, s = 1, \ldots, q \), and calculate the terms \( H_{k+1}(x^s_{k+1}) \) by solving the corresponding deterministic \((\ell - 1)\)-stages minimization. We can then approximate the function \( H_{k+1} \) using some form of regression to train a parametric architecture \( \tilde{H}_{k+1} \) such as a neural network. Once this is done, the MPC controls can be generated on-line using the minimization
\[
\hat{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} E\left\{ g_k(x_k, u_k, w_k) + \tilde{H}_{k+1}(f_k(x_k, u_k, w_k)) \right\}.
\]

Note that this method may be further supplemented by approximation in policy space in the spirit of Sections 2.1.2 and 2.1.3.

The off-line MPC training approach of the preceding example may be applied more broadly in situations where the on-line computational requirements are excessive. Moreover, approximation of the MPC Q-factors through training may also be used to circumvent the need to satisfy the constrained controllability condition, in order to guarantee that the state constraints are satisfied over the course of the system’s trajectory. In particular, we may replace the lookahead minimization of MPC with an approximation that enforces the state constraints in some approximate way. One possibility is to modify the stage costs of the \((\ell - 1)\)-stages problem solved by MPC and add a penalty for state constraint violation.

**Example 2.5.6 (MPC with Soft Safety Constraints)**

Here we change \( g_i(x_i, u_i) \) to a cost \( \hat{g}_i(x_i, u_i) \) that reflects the state constraints through addition of a corresponding constraint violation penalty to \( g_i(x_i, u_i) \). We also introduce a terminal cost function approximation \( \hat{G} \), which may also
reflect a penalty for state constraint violation, and the \( \ell \)-stages optimization problem

\[
\min_{u_i, i=k, \ldots, k+\ell-1} \left[ g_k(x_k, u_k) + \sum_{i=k+1}^{k+\ell-1} \hat{g}_i(x_i, u_i) + G(x_{k+\ell}) \right],
\]

subject to the system equation constraints

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

and the control constraints \( u_i \in U_i(x_i), \quad i = k, \ldots, k + \ell - 1 \).

We now consider a training process that involves a large number of sample states \( x_s^k, s = 1, \ldots, q \), and a solution of the above problem (2.36) with each of the initial conditions \( x_k = x_s^k \). This yields an optimal trajectory \( \{x_s^{k+1}, u_s^{k+1}, \ldots, x_s^{k+\ell}\} \), for every \( s \), and an \((\ell - 1)\)-stages cost

\[
\hat{J}_{k+1}(x_s^{k+1}) = \sum_{i=k+1}^{k+\ell-1} \hat{g}_i(x_s^i, u_s^i) + G(x_{s, k+\ell}).
\]

The state-cost pairs \( \{x_s^{k+1}, \hat{J}_{k+1}(x_s^{k+1})\}, s = 1, \ldots, q \), thus obtained can be used to construct a cost function approximation \( \hat{H}_{k+1} \) by means of an off-line training process. The functions \( \hat{H}_{k+1} \) may in turn be used to construct an MPC controller via approximation in value space with one-step lookahead:

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

The scheme of the preceding example is simple, and is presented here to illustrate the general idea of combining cost and safety considerations. There are other more sophisticated schemes suggested in the literature. Some of these schemes may involve stochastic uncertainty and certainty equivalence approximations. They may also involve repeated applications of MPC in the spirit of the perpetual rollout schemes described in Section 2.1.3. We refer to the end-of-chapter references for further discussion.

### 2.6 Multiagent Rollout

We will now consider a special structure of the control space, whereby the control \( u_k \) consists of \( m \) components, \( u_k = (u^1_k, \ldots, u^m_k) \), 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

\[
U_k(x_k) = U^1_k(x_k) \times \cdots \times U^m_k(x_k).
\]
Sec. 2.6 Multiagent Rollout

For the sake of the following discussion, we assume that each set $U^\ell_k(x_k)$ is finite, but the ideas of this section apply to the case of infinite constraint sets as well.

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

$$\hat{u}_k \in \arg \min_{u_k \in U_k(x_k)} E\left\{ g_k(x_k, u_k, w_k) + J_{k+1, \pi}(f_k(x_k, u_k, w_k)) \right\}, \quad (2.38)$$

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 (2.37)]. 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

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

where $J_{k, \tilde{\pi}}(x_k), k = 0, \ldots, N$, is the cost-to-go of the rollout policy starting from state $x_k$. Indeed we will use the special multiagent constraint structure (2.37) 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 (2.38) 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” \((x_k, u_k^1), (x_k, u_k^1, u_k^2), \ldots, (x_k, u_k^1, \ldots, u_k^m)\), 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 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 is simplified at the expense of introducing \( m - 1 \) additional layers of states, and corresponding \( m - 1 \) cost-to-go functions

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

in addition to \( J_k(x_k) \). The increase in size of the state space does not adversely affect the operation of rollout, since the Q-factor minimization

---

† 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.
Sec. 2.6 Multiagent Rollout

(2.38) is performed for just one state at each stage. Also, in a different context, the increase in size of the state space can be dealt with by using function approximation, i.e., with the introduction of cost-to-go approximations

\[ \tilde{J}_1^k(x_k, u_1^k, r_1^k), \tilde{J}_2^k(x_k, u_1^k, u_2^k, r_2^k), \ldots, \tilde{J}_m^{m-1}(x_k, u_1^k, \ldots, u_{m-1}^{m-1}, r_{m-1}^k), \]

in addition to \( \tilde{J}_k(x_k, r_k) \), where \( r_k, r_1^k, \ldots, r_{m-1}^k \) are parameters of approximation architectures such as neural networks; see Chapter 3.

Agent-by-Agent 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 \( \mu_k = (\mu_1^k, \ldots, \mu_m^k) \), 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

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

In particular, for each stage \( k \), the algorithm requires a sequence of \( m \) minimizations, once over each of the agent controls \( u_1^k, \ldots, u_m^k \), 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 \( U_i^k(x_k) \), the computation required at each stage \( k \) is of order \( O(n) \) for each of the “states”

\[ x_k, (x_k, u_1^k), \ldots, (x_k, u_1^k, \ldots, u_{m-1}^k), \]

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

To elaborate, at \( (x_k, u_1^k, \ldots, u_{\ell-1}^k) \) with \( \ell \leq m \), and for each of the controls \( u_\ell^k \), we generate by simulation a number of system trajectories up to stage \( N \), with all future controls determined by the base policy. We average the costs of these trajectories, thereby obtaining the \( Q \)-factor corresponding to \( (x_k, u_1^k, \ldots, u_{\ell-1}^k, u_\ell^k) \). We then select the control \( u_\ell^k \) that corresponds to the minimal \( Q \)-factor, with the controls \( u_1^k, \ldots, u_{\ell-1}^k \) held fixed at the values computed earlier.

Prerequisite assumptions for the preceding algorithm to work in an on-line multiagent setting are:

(a) All agents have access to the current state \( x_k \).

(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 \( u_1^k, \ldots, u_{\ell-1}^k \) 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.

The following spiders-and-flies example illustrates how multiagent rollout may exhibit intelligence and agent coordination that is totally lacking from the base policy. The behavior described in the example has been supported by computational experiments with larger spiders-and-flies problems.

**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 may unnecessarily move both spiders in the same direction, when in fact only one is needed to capture the fly. This limitation is due to the lack of coordination between the spiders: each acts selfishly, ignoring the
Sec. 2.6  Multiagent Rollout

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. Moreover, the example admits a two-dimensional generalization, whereby the two spiders, starting from the same position, will separate under the rollout policy, with each moving towards a different spider, while they will move in unison in the base policy along the shortest path to the closest surviving fly. Again this will happen for both standard and agent-by-agent rollout.

This example also 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 is “farsighted” 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.

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
it). 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 \( \tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\} \) obtained by multiagent rollout satisfies

\[
J_{k,\tilde{\pi}}(x_k) \leq J_{k,\pi}(x_k), \quad \text{for all } x_k \text{ and } k, \tag{2.39}
\]

where \( \pi \) is the base policy.

† For an example where the standard rollout algorithm works better, consider a single-stage problem, where the objective is simply to minimize the first stage cost

\[g_0(\mu_0^1, \ldots, \mu_0^m).\]

Let \( \pi_0 = (\pi_0^1, \ldots, \pi_0^m) \) be the control applied by the base policy, and assume that \( \pi_0 \) is not optimal. Suppose that starting at \( \pi_0 \), the cost cannot be improved by varying any single control component. Then the agent-by-agent rollout algorithm stays at the suboptimal \( \pi_0 \), while the standard rollout algorithm finds an optimal control. Thus, for one-stage problems, the standard rollout algorithm will perform as well or better than the agent-by-agent rollout algorithm.

This example is best seen within the classical framework of the coordinate descent method for minimizing a function of \( m \) components. This method can get stuck at a nonoptimal point in the absence of appropriate conditions on the cost function, such as differentiability and/or convexity. However, within our context of multistage rollout and possibly stochastic disturbances, it appears that the consequences of such a phenomenon may not be serious. In fact, one can construct multi-stage examples where agent-by-agent rollout performs better than the standard all-components-at-once rollout.
Proof: We will show the inequality (2.39) by induction, but for simplicity, we will give the proof for the case of just two agents, i.e., $m = 2$. Clearly the inequality holds for $k = N$, since $J_{N, \tilde{\pi}} = J_{N, \pi} = g_N$. Assuming that it holds for index $k + 1$, we have for all $x_k$,

$$J_{k, \tilde{\pi}}(x_k) = E \left\{ g_k(x_k, \tilde{\mu}_1^k(x_k), \tilde{\mu}_2^k(x_k), w_k) ight\} + J_{k+1, \tilde{\pi}} \left( f_k(x_k, \tilde{\mu}_1^k(x_k), \tilde{\mu}_2^k(x_k), w_k) \right) \\
\leq E \left\{ g_k(x_k, \tilde{\mu}_1^k(x_k), \tilde{\mu}_2^k(x_k), w_k) ight\} + J_{k+1, \tilde{\pi}} \left( f_k(x_k, \tilde{\mu}_1^k(x_k), \tilde{\mu}_2^k(x_k), w_k) \right) \\
= \min_{u_2^k \in U_2^k(x_k)} E \left\{ g_k(x_k, \tilde{\mu}_1^k(x_k), u_2^k, w_k) ight\} + J_{k+1, \tilde{\pi}} \left( f_k(x_k, \tilde{\mu}_1^k(x_k), u_2^k, w_k) \right) \\
\leq E \left\{ g_k(x_k, \tilde{\mu}_1^k(x_k), u_2^k, w_k) ight\} + J_{k+1, \tilde{\pi}} \left( f_k(x_k, \tilde{\mu}_1^k(x_k), u_2^k, w_k) \right) \\
= \min_{u_1^k \in U_1^k(x_k)} E \left\{ g_k(x_k, u_1^k, \mu_2^k(x_k), w_k) ight\} + J_{k+1, \tilde{\pi}} \left( f_k(x_k, u_1^k, \mu_2^k(x_k), w_k) \right) \\
\leq E \left\{ g_k(x_k, u_1^k, \mu_2^k(x_k), w_k) ight\} + J_{k+1, \tilde{\pi}} \left( f_k(x_k, u_1^k, \mu_2^k(x_k), w_k) \right) \\
= J_{k, \pi}(x_k),$$

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 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 (2.39) is thus complete for the case $m = 2$. The proof for an arbitrary number of agents $m$ is entirely similar. \textit{Q.E.D.}
Multiagent Rollout Variants

The agent-by-agent rollout algorithm admits several variants. We describe briefly a few of these variants below.

(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 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) When the control constraint sets $U^k(x_k)$ are infinite, agent-by-agent rollout still applies, based on the tradeoff between control and state space complexity, cf. Fig. 2.6.1. In particular, when the sets $U^k(x_k)$ are intervals of the real line, each agent’s lookahead minimization problem can be performed with the aid of one-dimensional search methods.

(d) 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.

(e) 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 agent-by-agent policy improvement, and approximation in value and in policy space of intermediately generated policies (see Chapters 3 and 4, and the RL book [Ber19a], Section 5.7.3).

In the next two subsections we discuss multiagent rollout variants in some specialized settings: the case of a decoupled system equation and cost structure, and a case involving the model predictive control context.

2.6.1 Multiagent Coupling Through Constraints

We have considered so far the case where the agents have independent constraint sets, so the control constraint set has the Cartesian product form

$$U_k(x_k) = U^1_k(x_k) \times \cdots \times U^m_k(x_k),$$
and/or the cost function. In a different type of multiagent problem, the system equation and cost function are decoupled, but the agents share control resources, so they are coupled through the control constraints. Here the state has the form

\[ x_k = (x^1_k, \ldots, x^m_k), \]

and evolves according to state equations that are decoupled with respect to the agents,

\[ x^\ell_{k+1} = f^\ell_k(x^\ell_k, u^\ell_k, w^\ell_k), \quad \ell = 1, \ldots, m, \]

while the stage cost function has the separable form

\[ g_k(x_k, u_k, w_k) = \sum_{\ell=1}^m g^\ell_k(x^\ell_k, u^\ell_k, w^\ell_k). \]

However, the control constraint is nonseparable and has the form

\[ u_k = (u^1_k, \ldots, u^m_k) \in U_k(x_k). \quad (2.40) \]

For this type of problem, we may consider a variant of the multiagent rollout algorithm. The base policy is assumed to satisfy the constraints (2.40) for all \( x_k \). The control components are again chosen one-agent-at-a-time in a given order, but the choice of the typical component \( u^\ell_k \) must satisfy the control feasibility constraint together with the components \( \tilde{u}_k^1, \ldots, \tilde{u}_k^{\ell-1} \) chosen by rollout for the preceding agents, and the components \( \bar{u}_k^{\ell+1}, \ldots, \bar{u}_k^m \) chosen by the base policy for the following agents.

Since each of the \( m \) lookahead minimization problems at state \( x_k \) involves a single agent, it involves a much smaller constraint set, with a dramatic reduction in the corresponding computational requirements. Moreover, it can be seen that thanks to the assumption that the base policy is feasible, each agent’s rollout minimization will maintain feasibility of the generated rollout policy. At the same time it can be verified that the proof of Prop. 2.6.1 goes through, so that the cost improvement property is satisfied.

We should note, however, that there is a potential difficulty in the one-agent-at-a-time lookahead minimization approach outlined above, because the constraints \( U_k(x_k) \) may restrict the degrees of freedom for changing individual components \( u^\ell_k \). For example, suppose that \( u_k \) is constrained to be a probability distribution, so that

\[ U_k(x_k) = \left\{ u_k \geq 0 \left| \sum_{\ell=1}^m u^\ell_k = 1 \right. \right\}. \quad (2.41) \]
Then it is impossible to change a single component of \( u_k \) while keeping the remaining \( m - 1 \) components fixed.

There are several ways to remedy such a situation. One possibility is to design the base heuristic so that it selects the control components one-at-a-time, with each component \( u_k^\ell \) chosen by taking into account the choices of the preceding components \( u_k^1, \ldots, u_k^{\ell-1} \), i.e., choose \( u_k^\ell \) from some subset of the form

\[
\hat{U}_k(x_k, u_k^1, \ldots, u_k^{\ell-1}),
\]

which takes into account feasibility of the future components \( u_k^{\ell+1}, \ldots, u_k^m \).

Then at state \( x_k \), agent \( \ell \) selects the component \( u_k^\ell \) by Q-factor minimization from the set

\[
\hat{U}_k(x_k, \tilde{u}_k^1, \ldots, \tilde{u}_k^{\ell-1}),
\]

where \( \tilde{u}_k^1, \ldots, \tilde{u}_k^{\ell-1} \) are the rollout choices of the preceding agents, and the subsequent choices \( u_k^{\ell+1}, u_k^m, u_{k+1}^1, u_{k+1}^2, \ldots \) obtained from the base heuristic.

As another possibility, we may group components together and consider group-by-group lookahead minimization, so that it is possible to make nontrivial changes of components within a group, while keeping the remaining components unchanged. For example in the probability distribution constraint case (2.41), we may optimize over simultaneous changes of two or more components while keeping the remaining components fixed. The method for forming groups is problem-dependent and may be based on insight into the problem at hand. Moreover, one may consider multiple group-by-group minimizations, with different group formations and group orders, and select the rollout control that yields the best results.

### 2.6.2 Multiagent Model Predictive Control

Let us consider the MPC context of Section 2.5.1 for the case of a “hybrid” form of control at time \( k \), where \( u_k \) consists of both discrete and continuous components. In particular, we have

\[
u_k = (y_k^1, \ldots, y_k^m, v_k),
\]

where \( y_k^1, \ldots, y_k^m \) are discrete control variables, taking a finite number of values, and \( v_k \) is a continuous control variable. Thus the system equation of the MPC framework is written as

\[
x_{k+1} = f_k(x_k, y_k^1, \ldots, y_k^m, v_k).
\]

Our framework of Section 2.5.1 can deal with such forms of control, but the one-step lookahead minimization of MPC may become very computationally expensive when \( m \) is large. This motivates a form a component-by-component one-step lookahead minimization, whereby each component
is chosen using discrete optimization with the future components fixed by a base policy. For example, $y^1_k, \ldots, y^m_k$ may represent system configuration variables that are chosen at the start of each time period. The base policy may consist of a “nominal configuration” $\bar{y}^1_k, \ldots, \bar{y}^m_k$, and a continuous control policy that drives the state to 0 in $(\ell - 1)$ steps with minimum cost, assuming the nominal configuration is used throughout the $(\ell - 1)$-step period.

In a component-by-component version of MPC, at state $x_k$ the components $y^1_k, \ldots, y^m_k$ are chosen one-at-a-time, with all future components fixed at the values determined by the nominal configuration/base policy, and the continuous component $v_k$ is then chosen to drive the state to 0 at some future time with minimum cost and with all future discrete components fixed at the nominal configuration. Moreover, by combining the cost improvement analysis of Section 2.6.1 with the stability analysis of Section 2.5.1, it can be verified that the MPC scheme produces a stable policy, assuming it can be implemented (i.e., an appropriate constrained controllability condition is satisfied).

One of the difficulties with the scheme just described is that in some applications the configuration choice $y^\ell_k$ may restrict future configuration choices. This amounts to control constraints that extend across multiple time steps. In methodological terms, this is a significant change, which affects critically the DP methodology, and renders MPC inapplicable in the standard form that we discussed in Section 2.5.1. To deal with this type of more complex constraints, it is necessary to reformulate the DP algorithm as well as the rollout algorithm. We postpone the discussion of possibilities of this type for Section 2.7, where we discuss constrained forms of rollout.

### 2.6.3 Asynchronous Distributed Multiagent Rollout

Our motivation for multiagent rollout so far has been computational efficiency: reduce the computational requirements of the one-step lookahead minimization when the control consists of multiple components. This results in substantial computational savings when all the computation is centralized at a single location.

However, multiagent rollout is also well-suited for problems where the control components are computed by separate autonomous agents. In this context the control components may be naturally computed in distributed and asynchronous fashion, 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 $\tilde{u}^\ell_k$ before knowing the rollout controls of some of the agents $1, \ldots, \ell - 1$, and uses the controls

$$\mu^1_k(x_k), \ldots, \mu^{\ell-1}_k(x_k)$$
of the base policy in their place. Note that there is a two-fold computational saving from such an algorithm, first by reducing the number of Q-factors to compute at state \( x_k \) from \( n^m \) to \( nm \), and then by reducing the time to compute some of these Q-factors in parallel.

Unfortunately, however, problems with multiple agents and asynchronous, distributed computation are notoriously difficult. They give rise to sequential decision problems with “nonclassical information patterns,” where agent decisions are made with limited coordination, and without exact knowledge of other agents’ control choices. Such problems can be very complex, partly because they cannot be addressed by exact DP. While natural forms of multiagent rollout may work well for such problems, they 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 in the preceding example stems from inadequate coordination between the two agents. In particular, each agent uses rollout to compute his/her local control, while thinking that the other agent will use the base policy control. If instead the two agents were to coordinate their control choices, they would have applied an optimal policy. In the terminology of decentralized optimization (also known as “team optimization”;

Sec. 2.6  Multiagent Rollout
multiagent rollout may not be able to improve a policy that is "person-by-person-optimal," i.e., each agent’s control component is optimal, assuming that the control components of all other agents are kept fixed. Such a policy may not be optimal, when all agents act in full coordination, except under special conditions; see the end-of-chapter references.

The simplicity of the preceding example raises serious questions as to whether the cost improvement property (2.39) 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 that are geographically far apart (and hence are less coupled) forego any coordination.

**Example 2.6.4 (Spiders-and-Flies)**

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 as 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 for-
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.

malism, 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 spiders.

Let us also 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
Parallelization Through Redundant Computation

In multiagent settings the agents often have some autonomy, so they can compute their control components independently and in parallel, perhaps with some form of limited interagent communication. One way to do this while maintaining the cost improvement property is for each agent to replicate the computation of the rollout controls of all agents, in the same order as the other agents. While this mode of operation entails redundancy of computations, the agents do not have to wait for other agents to communicate their computed control components.

This mode of redundant computation also allows for some flexibility regarding the information used by each agent through the use of estimates of quantities of interest and imperfect replication. In particular, the agents may try to replicate each other’s computation, while using estimates of the other agents’ controls, or estimates of the state. In this way the computation of a stage may be speeded up through parallelization, while the estimates of the other agents’ controls and/or the state may be updated over time through information exchange between the agents. Of course schemes of this type should be adapted to the character of the problem at hand, and may offer limited performance guarantees.

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. We will first consider the special case of rollout with just state constraints, in the spirit of the MPC framework of Section 2.5. We then develop a constrained rollout algorithm for problems with more general constraints, which relies on a base heuristic 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 agent-by-agent 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 with both a spatial and a temporal structure.
2.7.1 State-Constrained Rollout and Target Tubes

Consider a deterministic optimal control problem with system
\[ x_{k+1} = f_k(x_k, u_k), \]
cost per stage \( g_k(x_k, u_k) \), and state and control constraints
\[ x_k \in X_k, \quad u_k \in U_k(x_k), \quad k = 0, 1, \ldots. \]

We would like to apply one-step lookahead rollout using some base heuristic, which starting from a state \( x_k \in X_k \), generates a trajectory \((u_k, x_{k+1}, \ldots, u_{N-1}, x_N)\) that satisfies \( u_i \in U(x_i) \) for all \( i = k, \ldots, N - 1 \). This is similar to the rollout framework of Section 2.3.1, except for a substantial complication: the additional state constraints \( x_i \in X_i, i = k + 1, \ldots, N \), may be violated when starting from some states \( x_k \in X_k \) and using the base heuristic.

We will now modify the rollout methodology to take into account the state constraints. There are two issues in this connection:

(a) How to ensure that the rollout trajectory satisfies the state constraints.

(b) How to guarantee the cost improvement property of rollout.

To address these issues, it is necessary to modify the state constraints by introducing a target tube that enforces feasibility of the rollout policy. This is similar to the use of target tubes in connection with MPC in Section 2.5.1, but the context is different because we are using a general base heuristic, rather than the specialized base policy that underlies MPC (drive the state to 0 in \( \ell - 1 \) steps with minimum cost). The critical construction is to identify sets of states \( \overline{X}_k \subset X_k \) such that starting from within \( \overline{X}_k \) we are guaranteed to stay within the “tube” \( \{\overline{X}_{k+1}, \overline{X}_{k+2}, \ldots, \overline{X}_N\} \) for all subsequent times with feasible choices of control obtained by the rollout algorithm. We referred to nonempty tubes of this kind as reachable in Section 2.5.1.

The largest reachable tube is generated by the algorithm given in Section 2.5.1 in the context of MPC. It starts with
\[ \overline{X}_N = X_N, \]
and generates \( \overline{X}_k, k = 0, \ldots, N - 1 \), going backwards:
\[ \overline{X}_k = \{x_k \in X_k \mid \text{for some } u_k \in U_k(x_k) \text{ we have } f_k(x_k, u_k) \in \overline{X}_{k+1}\}. \]
Then any algorithm, which at state $x_k$, selects a control from within the set

$$U_k(x_k) = \{ u_k \in U_k(x_k) \mid f_k(x_k, u_k) \in X_{k+1} \}$$

has the property that it generates a feasible trajectory when started from a state within $X_0$. This yields an algorithm for maintaining feasibility: any policy $\pi = \{ \mu_0, \ldots, \mu_{N-1} \}$ that satisfies

$$\mu_k(x_k) \in \overline{U}_k(x_k), \quad \text{for all } x_k \in X_k \text{ and } k,$$

satisfies the state and control constraints of the problem, provided the initial state $x_0$ belongs to the set $X_0$. This holds true regardless of whether the choice of $\mu_k(x_k)$ is made with the aid of the base heuristic.

Thus if a reachable tube $\{X_0, X_1, \ldots, X_N\}$ is available, a natural approach is to start from an initial state in $X_0$, and at the current state $x_k$ compute a trial choice of control $u_k$ using some version of the rollout algorithm. If $u_k \in \overline{U}_k(x_k)$, then apply this control, and otherwise choose $u_k$ arbitrarily from within $\overline{U}_k(x_k)$. Such an algorithm is guaranteed to satisfy the state and control constraints, but it may not guarantee cost improvement over the base heuristic.

Another major difficulty with the preceding approach is that while the target tube $\{X_0, X_1, \ldots, X_N\}$ is the largest that is reachable, it may be difficult to compute. This suggests algorithms that produce inner approximations to the sets $X_k$. As noted in Section 2.5.1, there are algorithms that construct ellipsoidal, polyhedral, and other inner tube approximations. We will discuss another possibility: an inner approximation that is related to the base heuristic.

In particular, let us assume that the base heuristic is sequentially consistent, so that it is defined by a base policy $\pi = \{ \mu_0, \ldots, \mu_{N-1} \}$. Let us consider the tube $\{\hat{X}_0, \hat{X}_1, \ldots, \hat{X}_N\}$ that corresponds to the base policy, and is defined by

$$\hat{X}_k = \{ x_k \in X_k \mid f_k(x_k, \mu_k(x_k)) \in \hat{X}_{k+1} \},$$

with $\hat{X}_N = X_N$, and assume that the sets $\hat{X}_0, \ldots, \hat{X}_{N-1}$ are nonempty. Consider the rollout algorithm, which at any state $x_k \in \hat{X}_k$ generates a rollout control according to

$$\hat{\mu}_k(x_k) \in \arg \min_{u_k \in \hat{U}_k(x_k)} \left[ g_k(x_k, u_k) + J_{k+1, \pi}(f_k(x_k, u_k)) \right],$$

where $J_{k+1, \pi}$ is the cost-to-go function of the base policy at stage $k + 1$, and $\hat{U}_k(x_k)$ is the set

$$\hat{U}_k(x_k) = \{ u_k \in \hat{U}_k(x_k) \mid f_k(x_k, u_k) \in \hat{X}_{k+1} \},$$
which is nonempty by the definition of the sets \( \bar{X}_k \) and \( \bar{X}_{k+1} \) [since the control applied by the base policy at \( x_k \) belongs \( \bar{U}_k(x_k) \)].

It can be seen that starting from an initial state within \( \bar{X}_0 \), this rollout algorithm maintains feasibility in the sense that it satisfies the control constraints, and keeps the state within the tube

\[ \{ \bar{X}_0, \bar{X}_1, \ldots, \bar{X}_N \}, \]

so it satisfies the state constraints as well. Moreover, the algorithm guarantees cost improvement, in the sense that

\[ J_{k, \bar{\pi}}(x_k) \leq J_{k, \pi}(x_k), \quad \text{for all } x_k \in \bar{X}_k, \]

where \( \bar{\pi} = \{ \bar{\mu}_0, \ldots, \bar{\mu}_{N-1} \} \) is the rollout policy. The reason is that in view of the sequential consistency of the base policy, the set \( \bar{U}_k(x_k) \) always contains the base policy control \( \mu_k(x_k) \) for all \( x_k \in \bar{X}_k \), and the proof of cost improvement, given in Prop. 2.3.2, goes through.

Note that while the tube \( \{ \bar{X}_0, \bar{X}_1, \ldots, \bar{X}_N \} \) may be hard to construct analytically, it may be computationally approximated by running the base policy starting from many initial sample states \( x_0 \), and generating a subset of states within each set \( X_k \). In the special case where it is known that the sets \( \bar{X}_k \) are convex, one can obtain a better approximation by forming the convex hulls of the generated sets of sample states. This approach has been proposed and implemented in the context of MPC and a linear system with set membership disturbances in the work of Rosolia and Borrelli [RoB20a], [RoB20b], who have embedded it within an approximate policy iteration scheme.

The preceding discussion also applies to deterministic infinite horizon problems with stationary system equation \( x_{k+1} = f(x_k, u_k) \), cost per stage \( g(x, u) \), state and control constraints \( x \in X \) and \( u \in U(x) \), and a termination state. Within this infinite horizon context, the reachable tubes \( \{ X_0, X_1, \ldots, X_N \} \) and \( \{ \bar{X}_0, \bar{X}_1, \ldots, \bar{X}_N \} \) must be replaced by corresponding infinitely reachable tubes \( \{ \bar{X}, \bar{X}, \ldots \} \) and \( \{ \bar{X}, \bar{X}, \ldots \} \). Here \( \bar{X} \) is the largest set within which the state can be kept for an infinite number of stages, and \( \bar{X} \) is the set within which the state can be kept for an infinite number of stages by using the base policy.† The rollout policy has the form

\[ \bar{\mu}(x) \in \arg \min_{u \in \bar{U}(x)} \left[ g(x, u) + J_\pi(f(x, u)) \right], \quad x \in \bar{X}, \]

where \( J_\pi \) is the cost-to-go function of the base policy \( \pi \), and \( \bar{U}(x) \) is the set

\[ \bar{U}(x) = \{ u \in U(x) \mid f(x, u) \in \bar{X} \}, \quad x \in \bar{X}. \]

The following example illustrates the main ideas with a continuous-spaces shortest path problem.

† The concept of infinite time reachability of target tubes for discrete-time systems and set-membership uncertainty was formulated and analyzed in the author’s paper [Ber72], to which we refer for further discussion.
Figure 2.7.1. Illustration of rollout for a continuous spaces shortest path problem. The state constraint $X$ is bordered by a square from the outside and by a circle from the inside, with termination state a sphere of radius $\epsilon$, where $\epsilon$ is a small positive constant. The control can move the state along any direction by an amount $[\epsilon, 1]$. The base policy moves along a straight line towards the termination state, but this line must not cross outside the feasible region $X$. The largest infinitely reachable set is $X$ is the entire state constraint set $X$, while the infinitely reachable set $\tilde{X}$ using the base policy is the shaded region shown in the figure.

Example 2.7.1 (Rollout for Continuous-Spaces Shortest Path Problems)

Consider a two-dimensional set $X$ that is bordered by a square from the outside and by a circle from the inside, as shown in Fig. 2.7.1. The objective is to reach the termination state (a sphere of radius $\epsilon$, where $\epsilon$ is a small positive constant), using control that at any state $x \in X$, can move the state along any direction by an amount $[\epsilon, 1]$. Except at the termination state, there is a positive cost per stage at all states, which is immaterial for the purposes of this example. Consider a base policy that moves the state along any straight line that passes through the destination, but does not cross outside the feasible region $X$. The infinitely reachable tubes $\{X, \tilde{X}, \ldots\}$ and $\{X, \tilde{X}, \ldots\}$ are as shown in the figure. The rollout policy starting from any $x_0 \in \tilde{X}$ attains feasibility and has no worse cost than the base policy. However, this rollout policy is defined only within the set $\tilde{X}$. 
2.7.2 Rollout with Trajectory Constraints

We now consider rollout for deterministic optimal control with general trajectory constraints. We will pay special attention to discrete optimization problems. There are many methods available for 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 $u$, each consisting of all $N$ components. This suggests that rollout may be better suited for problems that have a naturally sequential character.

We assume that the state $x_k$ takes values in some (possibly infinite) set and the control $u_k$ takes values in some finite set, and $f_k$ is some function. A sequence of the form

$$ T = (x_0, u_0, x_1, u_1, \ldots, u_{N-1}, x_N), \quad (2.42) $$

where

$$ x_{k+1} = f_k(x_k, u_k), \quad k = 0, 1, \ldots, N - 1, \quad (2.43) $$

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

$$ \min_{T \in C} G(T), \quad (2.44) $$

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

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

$$ G(x_0, u_0, x_1, u_1, \ldots, u_{N-1}, x_N) = g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k), \quad (2.45) $$

$^\dagger$ 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 (cf. Section 2.3.4). The expert should also be able to determine whether a given trajectory $T$ satisfies the constraint $T \in C$. 
where the controls satisfy the time-uncoupled constraints

\[ u_k \in U_k(x_k), \quad k = 0, 1, \ldots, N - 1, \quad (2.46) \]

(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.

For an example of 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 2.7.2 later in this section).

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

\[ g_N^m(x_N) + \sum_{k=0}^{N-1} g_k^m(x_k, u_k) \leq b^m, \quad m = 1, \ldots, M, \quad (2.47) \]

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

Another situation that involves difficult trajectory constraints arises when the control contains some discrete components, which once chosen must remain fixed for multiple time periods. We described an example of this type in our discussion of multiagent MPC in Section 2.6.2. Within that context, some of the control components involved some "configuration parameters," which may affect the system equation for several time periods. Situations of this type may be addressed with the constrained rollout methodology of this section, thereby leading to corresponding constrained forms of MPC.

### Transforming Constrained DP Problems to Unconstrained Problems

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:

\[ y_{k+1} = (y_k, u_k, f_k(x_k, u_k)). \quad (2.48) \]
Sec. 2.7 Constrained Rollout for Deterministic Optimization

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 (2.45)-(2.46), 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 (2.48) 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

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

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 $(y_k, u_k, R(y_{k+1}))$, which is denoted by $T_k(y_k, u_k)$:

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

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

(2.49)
Our rollout algorithm starts from a given initial state \( \tilde{y} \) produced by the base heuristic starting from \( \tilde{x} \). It considers successive partial trajectories \( \tilde{y} \) possible for only a subset of controls. 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 \( T_k(\tilde{y}_k, \tilde{u}_k) \):

\[
\tilde{u}_k \in \arg \min_{u_k \in U_k(\tilde{y}_k)} G(T_k(\tilde{y}_k, u_k));
\]

see Fig. 2.7.2. The objective is to produce a feasible final complete trajectory \( \tilde{y}_N \), which has a cost \( G(\tilde{y}_N) \) that is no larger than the cost of \( R(\tilde{y}_0) \) produced by the base heuristic starting from \( \tilde{y}_0 \), i.e.,

\[
G(\tilde{y}_N) \leq G(R(\tilde{y}_0)).
\]
Sec. 2.7 Constrained Rollout for Deterministic Optimization

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

\[
y_k = (\tilde{x}_0, \tilde{u}_0, \tilde{x}_1, \ldots, \tilde{u}_{k-1}, \tilde{x}_k)
\]  

(2.53)

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

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

The algorithm then forms the set of controls

\[
U_k(\tilde{y}_k) = \{ u_k | T_k(\tilde{y}_k, u_k) \in C \}
\]

that is consistent with feasibility [cf. Eq. (2.50)], and chooses a control \( \tilde{u}_k \in U_k(\tilde{y}_k) \) according to the minimization

\[
\tilde{u}_k \in \arg \min_{u_k \in U_k(\tilde{y}_k)} G(T_k(\tilde{y}_k, u_k)), \quad (2.54)
\]

[cf. Eq. (2.52)], where

\[
T_k(\tilde{y}_k, u_k) = (\tilde{y}_k, \tilde{u}_k, \tilde{R}(f_k(\tilde{x}_k, u_k)));
\]

[cf. Eq. (2.49)]. Finally, the algorithm sets

\[
\tilde{x}_{k+1} = f_k(\tilde{x}_k, \tilde{u}_k), \quad \tilde{y}_{k+1} = (\tilde{y}_k, \tilde{u}_k, \tilde{x}_{k+1})
\]

[cf. Eq. (2.51)], 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 (2.52) 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., \( \tilde{R}(\tilde{y}_0) \in C \).

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 (2.54) 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

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

it will then follow that the cost improvement condition \( G(\hat{y}_N) \leq G(R(\hat{y}_0)) \) [cf. Eq. (2.5)] 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. (2.48)] 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 \textit{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
\[
G(y_k \cup R(y_k)) \geq \min_{u_k \in U_k(y_k)} G(T_k(y_k, u_k)). \quad (2.55)
\]

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 \(\tilde{y}_0 = \tilde{x}_0\), i.e., \(R(\tilde{y}_0) \in C\). Then for each \(k\), the set \(U_k(\tilde{y}_k)\) is nonempty, and we have
\[
G(R(\tilde{y}_0)) \geq G(T_0(\tilde{y}_0, \tilde{u}_0))
\geq G(T_1(\tilde{y}_1, \tilde{u}_1))
\geq \cdots
\geq G(T_{N-1}(\tilde{y}_{N-1}, \tilde{u}_{N-1}))
= G(\tilde{y}_N),
\]
where
\[
T_k(\tilde{y}_k, \tilde{u}_k) = (\tilde{y}_k, \tilde{u}_k, R(\tilde{y}_{k+1}));
\]
cf. Eq. (2.49). 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. (2.55)].
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
\[
T_0(\tilde{y}_0, \tilde{u}_0) = (\tilde{x}_0, \tilde{u}_0, \tilde{x}_1, u'_1, x'_1, u'_2, \ldots, u'_{N-1}, x'_N).
\]
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 \((2.55)\) 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, \( \tilde{y}_k \cup R(\tilde{y}_k) \) 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
\[
g^m_N(x_N) + \sum_{k=0}^{N-1} g^m_k(x_k, u_k) \leq b^m, \quad m = 1, \ldots, M,
\]
[cf. Eq. \((2.47)\)]. 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 \( \hat{V}^m(x_k) \) be the value of the

† Indeed the sequential consistency assumption may be replaced by the weaker condition that the algorithm satisfies
\[
\tilde{y}_k \cup R(\tilde{y}_k) \in \{ T_k(\tilde{y}_k, u_k) \mid u_k \in U_k(\tilde{y}_k) \}.
\]
In fact this is the idea behind the fortified variant of the algorithm to be discussed shortly: essentially, the complete trajectory \( \tilde{y}_k \cup R(\tilde{y}_k) \) is artificially added to the set \( \{ T_k(\tilde{y}_k, u_k) \mid u_k \in U_k(\tilde{y}_k) \} \).
mth 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.,

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

Then, it can be seen that the base heuristic is sequentially improving if in addition to Eq. (2.55), the partial trajectory \( R(x_k) \) satisfies

\[
g_k^m(x_k, u_k) + \hat{V}^m(x_{k+1}) \leq \hat{V}^m(x_k), \quad m = 1, \ldots, M. \tag{2.56}
\]

The reason is that if a complete trajectory of the form

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

belongs to \( C \), then we have

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

and from Eq. (2.56), it follows that

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

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. (2.50)] 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 \( \tilde{x}_k \), i.e., \( \hat{T}_k \) has the form
\[ \hat{T}_k = (\tilde{x}_0, \tilde{u}_0, \tilde{x}_1, \ldots, \tilde{u}_{k-1}, \tilde{x}_k, \tilde{x}_{k+1}, \ldots, \tilde{x}_N), \quad (2.57) \]
for some \( \tilde{x}_{k+1} = f_k(\tilde{x}_k, \tilde{u}_k), \ldots, \tilde{x}_{N-1} = \tilde{x}_N \) such that
\[ x_{k+1} = f_k(\tilde{x}_k, u_k), \quad x_{t+1} = f_t(x_t, u_t), \quad t = k + 1, \ldots, N - 1. \]
Initially, \( \hat{T}_0 \) is the complete trajectory \( R(\tilde{y}_0) \), generated by the base heuristic starting from \( \tilde{y}_0 \), which is assumed to be feasible. At stage \( k \), the algorithm forms the subset \( \hat{U}_k(\tilde{y}_k) \) of controls \( u_k \in U_k(\tilde{y}_k) \) such that the corresponding \( T_k(\tilde{y}_k, u_k) \) is not only feasible, but also has cost that is no larger than the one of the current tentative best trajectory:
\[ \hat{U}_k(\tilde{y}_k) = \left\{ u_k \in U_k(\tilde{y}_k) \mid G(T_k(\tilde{y}_k, u_k)) \leq G(\hat{T}_k) \right\}. \]
There are two cases to consider at state \( k \):

(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
\[ \tilde{u}_k = \arg\min_{u_k \in U_k(\tilde{y}_k)} G(T_k(\tilde{y}_k, u_k)), \quad \tilde{x}_{k+1} = f_k(\tilde{x}_k, \tilde{u}_k), \]
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
\[ \tilde{u}_k = \pi_k, \quad \tilde{x}_{k+1} = \pi_{k+1}, \]
and \( \pi_k, \pi_{k+1} \) are the control and state subsequent to \( \tilde{x}_k \) in the current tentative best trajectory \( \hat{T}_k \) [cf. Eq. (2.57)], and leaves \( \hat{T}_k \) unchanged, i.e.,
\[ \hat{T}_{k+1} = \hat{T}_k. \]

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

Note that if the base heuristic is sequentially improving, and the fortified rollout algorithm will generate the same complete trajectory as the (nonfortified) rollout algorithm given earlier, with the tentative best trajectory $\hat{T}_{k+1}$ being equal to the complete trajectory $T_k(\hat{y}_k, \hat{u}_k)$ for all $k$. The reason is that if the base heuristic is sequentially improving the controls $\hat{u}_k$ generated by the nonfortified algorithm belong to the set $\hat{U}_k(\hat{y}_k)$ [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 $\hat{y}_0$. 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 $\hat{y}_0$.

**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 $\hat{y}_k$ of Eq. (2.53), we can maintain a tree of partial trajectories that is rooted at $\hat{y}_0$. 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 $\hat{y}_k$. Then we extend $\hat{y}_k$ 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.
2.7.3 Constrained Multiagent Rollout

Let us assume a special structure of the control space, where the control $u_k$ consists of $m$ components, $u_k = (u^1_k, \ldots, u^m_k)$, each belonging to a corresponding set $U^\ell_k(x_k)$, $\ell = 1, \ldots, m$. Thus the control space at stage $k$ is the Cartesian product

$$U_k(x_k) = U^1_k(x_k) \times \cdots \times U^m_k(x_k).$$

(2.58)

Similar to Section 2.6, we refer to this as the multiagent case, motivated by the special case where each component $u^\ell_k$, $\ell = 1, \ldots, m$, is chosen by a separate agent $\ell$ at stage $k$. Then the rollout minimization (2.54) 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 $U^\ell_k(x_k)$ [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 (2.58)]. Thus the rollout algorithm requires order $O(n^m)$ applications of the base heuristic per stage.

Similar to the stochastic unconstrained case of Section 2.6, we will construct an alternative rollout algorithm that achieves the cost improvement property (2.5) with much smaller computational cost, namely order $O(nm)$ applications of the base heuristic per stage. The key idea is to reformulate the problem by “unfolding” the control $u_k$ into its $m$ components, which are applied one agent-at-a-time.

We thus introduce a modified but equivalent problem, involving one-at-a-time agent control selection. In particular, at the generic state $x_k$, we break down the control $u_k$ into the sequence of the $m$ controls $u^1_k, u^2_k, \ldots, u^m_k$, and between $x_k$ and the next state $x_{k+1} = f_k(x_k, u_k)$, we introduce artificial intermediate “states” $(x_k, u^1_k), (x_k, u^1_k, u^2_k), \ldots, (x_k, u^1_k, \ldots, u^{m-1}_k)$, and corresponding transitions. The choice of the last control component $u^m_k$ at “state” $(x_k, u^1_k, \ldots, u^{m-1}_k)$ marks the transition at cost $g_k(x_k, u_k)$ to the next state $x_{k+1} = f_k(x_k, u_k)$ according to the system equation; see Fig. 2.7.3. 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. 2.7.3, with a base heuristic suitably modified so that it completes a partial trajectory of the form

$$(y_k, (x_k, u^1_k), (x_k, u^1_k, u^2_k), \ldots, (x_k, u^1_k, \ldots, u^\ell_k)), \quad \ell = 1, \ldots, m.$$

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

$$(x_k, u^1_k), (x_k, u^1_k, u^2_k), \ldots, (x_k, u^1_k, \ldots, u^{m-1}_k).$$
Figure 2.7.3. Equivalent formulation of the $N$-stage deterministic optimal control problem for the multiagent case where the control $u_k$ consists of $m$ components $u_1^k, u_2^k, \ldots, u_m^k$:

$$u_k = (u_1^k, \ldots, u_m^k) \in U_1^k(x_k) \times \cdots \times U_m^k(x_k).$$

The figure depicts the $k$th stage transitions; cf. Fig. 2.6.1.

In particular, for each stage $k$, the algorithm requires a sequence of $m$ minimizations, one over each of the control components $u_1^k, \ldots, u_m^k$, with the past controls already determined by the rollout algorithm, and the future controls determined by running the base heuristic. Assuming a maximum of $n$ elements in the control component spaces $U_\ell^k$, $\ell = 1, \ldots, m$, the computation required at each stage $k$ is of order $O(n)$ for each of the “states”

$$x_k, (x_k, u_1^k), \ldots, (x_k, u_1^k, \ldots, u_{m-1}^k),$$

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

To elaborate, for all $k$ and $\ell \leq m$, at the current partial trajectory

$$(\tilde{x}_0, \tilde{u}_0, \ldots, \tilde{x}_k, \tilde{u}_k^{\ell-1}),$$

and for each of the controls $u_\ell^k$, we use the base heuristic to generate a complementary partial trajectory

$$(u_{k+1}^{\ell+1}, \ldots, u_m^k, x_{k+1}^k, u_{k+1}^k, \ldots, x_{N-1}^k, u_{N-1}^k, x_N), \quad (2.59)$$

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 (2.59), 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 2.7.3 (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 2.7.4 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.

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:

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

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 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 \((u_0)\), with \( u_0 \) belonging to the set

\[ U_0 = \{ u_0 \mid \text{there exists a solution of the form } (u_0, \overline{u}_1, \ldots, \overline{u}_{N-1}) \in C \}. \]

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

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

This is 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.

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.

† The reverse is also true, namely that any constrained deterministic optimal control problem of the form (2.42)-(2.44), can be converted to the general discrete optimization form (2.60), simply by expressing the states \( x_k \) as functions of the preceding controls \( u_0, \ldots, u_{k-1} \) through the system equation (2.43), and eliminating them from the cost function expression and the constraints. This abstraction of the problem may be of value in some contexts because of its inherent simplicity.
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 a total of $M$ “clients.” Each client $i = 1, \ldots, M$ has a demand $d_i$ for services that may be satisfied at a location $k = 0, \ldots, N - 1$ at a cost $a_{ik}$ per unit. If a facility is placed at location $k$, it has capacity to serve demand up to a known level $c_k$.

We introduce a 0-1 integer variable $u_k$ to indicate with $u_k = 1$ that a facility is placed at location $k$ at a cost $b_k$ and with $u_k = 0$ that a facility is
Figure 2.8.2 Schematic illustration of the facility location problem; cf. Example 2.8.1. Clients are matched to facilities, and the location of the facilities is subject to optimization.

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

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

(2.61)

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

(2.62)

together with

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

(2.63)

We wish to minimize the cost

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

(2.64)

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

On the other hand, when all the variables $u_k$ are fixed at some 0 or 1 values, the problem belongs to the class of linear transportation problems (see e.g., [Ber98]), and can be solved by fast polynomial algorithms. Thus the
essential difficulty of the problem is how to select the 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 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 \((2.64)\) subject to the constraints \((2.61)-(2.63)\), 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 problems, and fix \( u_1 \), depending on which of the two resulting costs is smallest, etc.

It is easily seen that if the initial base heuristic choice (placing a facility at every candidate location) is feasible, i.e., \( \sum_{i=1}^{M} d_i \leq \sum_{k=0}^{N-1} c_k \), the rollout algorithm will yield a feasible solution with cost that is no larger than the cost corresponding to the initial application of the base heuristic. In fact it can be verified that the base heuristic here is sequentially 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 are defined by fairly similar data. It is thus important to use an algorithm that is capable of using effectively the final solution of one transportation problem as a starting point for the solution of the next. The auction algorithm for transportation problems (Bertsekas and Castaño [BeC89]) is particularly well-suited for this purpose.

**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; cf. Section 2.3.4); 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 refer to Pedrielli et al. [PLS20] for further discussion of this rollout approach.

One more aspect of the problem that is worth noting is that there are at most three choices for control at each state, while the problem is deterministic. As a result, the problem is a good candidate for the use of multistep lookahead. In particular, with \(\ell\)-step lookahead, the number of Q-factors to be computed at each state increases from 3 (or less) to \(3^{\ell}\) (or less).

**Example 2.8.3 (Constrained Directed Spanning Trees)**

Let us consider a spanning tree-type problem involving a directed graph with nodes 0, 1, \ldots, \(N\). Node \(N\) is special and is viewed as a “root” node. At each node \(k \in \{0, \ldots, N - 1\}\) there is a set of outgoing arcs \(u_k \in U_k\). We are
interested in collections of arcs involving one arc per node,

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

with \( u_k \in U_k, k = 0, \ldots, N - 1 \). We require that these arcs do not form any directed cycle, so that \( u \) specifies a directed spanning tree that is rooted at node \( N \). We wish to find \( u \) that minimizes a given cost function \( G(u) \) subject to certain additional constraints, which we do not specify further. The set of all constraints on \( u \) (including the directed spanning tree constraint) is denoted abstractly as \( u \in C \), so the problem comes within our constrained optimization framework of this section.

Suppose that we have an initial solution

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

which is feasible. Let us apply the constrained rollout algorithm with the base heuristic, which given a partial trajectory

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

generates the complete trajectory

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

Thus the rollout algorithm, given a partial trajectory

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

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

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

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

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

It can be seen that the base heuristic is sequentially improving, and that the rollout algorithm will produce a sequence of feasible solutions with nonincreasing cost at each step. An interesting aspect of this rollout algorithm is that it can be applied multiple times with the final solution of one rollout application used to specify the base heuristic of the next rollout application. Moreover, a different order of nodes may be used in each rollout application. The algorithm will eventually terminate, in the sense that it can make no further progress. However, there is no guarantee that the final solution thus obtained will be optimal.
Sec. 2.8  Discrete and Combinatorial Optimization  Applications  149

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.

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 \(\mathcal{N}_0, \mathcal{N}_1, \ldots, \mathcal{N}_N\), and referred to as layers. The arcs of the graphs are directed and are of the form \((i, j)\), where \(i\) is a node in a layer \(\mathcal{N}_k, k = 0, 1, \ldots, N-1\), and \(j\) is a node in the corresponding next layer \(\mathcal{N}_{k+1}\). Thus we have a directed graph 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 \(\mathcal{N}_k\) contains the same number of nodes, say \(m\), and that there is a unique arc connecting each node in a given layer with each of the nodes in 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.

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 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
Three-Dimensional Assignment Problem

Jobs $j$  
Machines $\ell$  
Workers $w$

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 $m = 4$.

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 is the polynomially solvable 2-dimensional
Rollout and Policy Improvement  Chap. 2

The machines-to-workers assignment problem

\[
\text{minimize } \sum_{w=1}^{m} \sum_{\ell=1}^{m} c_{\ell w} v_{\ell w} \\
\text{subject to } \sum_{\ell=1}^{m} v_{\ell w} = 1, \quad \forall \ w = 1, \ldots, m, \\
\sum_{w=1}^{m} v_{\ell w} = 1, \quad \forall \ \ell = 1, \ldots, m, \\
v_{\ell w} = 0 \text{ or } 1, \quad \forall \ \ell, \ w = 1, \ldots, m,
\]

where

\[ c_{\ell w} = \min_{j=1, \ldots, m} a_{j \ell w}, \quad (2.65) \]

\( v_{\ell w} \) 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.†

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 \( a_{j \ell w} \) 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

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

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.

† An alternative is to define each cost \( c_{\ell w} \) as a “representative” cost \( a_{j_{\ell w} \ell w} \) (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. (2.65) 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.
within $4n\epsilon$ (see the author’s network optimization book \cite{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 $u_0^1, \ldots, u_0^m$), 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. (2.65)]. 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
\[ b_{j\ell} = a_{j\ell w_\ell}, \quad j, \ell = 2, \ldots, m. \]
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. (2.65)]. 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 $1, \ldots, t-1, t+1,$
$1, \ldots, m$, where the costs are
\[ b_{j\ell} = a_{j\ell w_\ell}, \quad j = 2, 3, \ldots, m, \ell = 1, \ldots, t-1, t+1, \ldots, m. \]
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 correspond-
ing 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 $S_\ell$ is minimized over $\ell = 1, \ldots$, and we permanently assign job 1 to machine $\tilde{\ell}$ if $S_\ell$ is less or equal to $\hat{S}$, 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. (2.65)]. 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 \( W_k(x_k, u_k) \subseteq D_k \), \( k = 0, 1, \ldots, N - 1 \), 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

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

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):

\[
J_\pi^*(x_N) = g_N(x_N),
\]

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

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 \( \pi^* = \{ \mu_0^*, \mu_1^*, \ldots, \mu_{N-1}^* \} \) 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 (2.66)-(2.67). 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.

† The minimax approach to decision and control has its origins in the 50s and 60s, and is also referred to by other names, depending on the underlying context, such as robust control, robust optimization, control with a set membership description of the uncertainty, and games against nature.
Approximation in value space with one-step lookahead applies at state $x_k$ a control

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

where $\tilde{J}_{k+1}(x_{k+1})$ is an approximation to the optimal cost-to-go $J^*_k(x_{k+1})$ 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 $\hat{u}_k$ by solving the minimax problem in Eq. (2.68).

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.

**Example 2.9.1 (Pursuit-Evasion Problems)**

Consider a pursuit-evasion problem with state $x = (z_1, z_2)$, where $z_1$ is the location of the minimizer/pursuer and $z_2$ is the location of the maximizer/evader, in a (finite node) graph defined in two- or three-dimensional space. There is also a cost-free and absorbing termination state $t$ that consists of a subset of pairs $(z_1, z_2)$ that includes all pairs with $z_1 = z_2$. We assume that all arc costs are positive, so the objective for the pursuer is to
reach the termination state (i.e., catch the evader) with minimum cost. A suitable base policy \( \mu \) is for the pursuer to follow a shortest path from \( z_1 \) to \( z_2 \) under the assumption that the evader will stay at his current location \( z_2 \) at all future times. To do this for all \((z_1, z_2)\) requires the solution of an all-pairs shortest path problem, which is possible by using for example the Floyd-Warshall algorithm [AMO89], [Ber98].

Suppose now that we have precomputed \( \mu(x) \) for all \( x = (z_1, z_2) \) with this shortest path computation. Let also \( Y(x, u) \) denote the set of possible states \( y \) that can be reached following the choice of the evader, and let \( g(x, u, y) \) denote the corresponding cost. Then the maximization

\[
\max_{y \in Y(x, u)} \left\{ g(x, u, y) + J_\mu(y) \right\}
\]

that is needed for the on-line computation of the rollout control requires the calculation of \( J_\mu(y) \) for each \( y \in Y(x, u) \). Knowing \( \mu \), each of these calculations is a tractable longest path computation in an acyclic graph.

Note that the preceding algorithm can be adapted for the imperfect information case where the pursuer knows \( z_2 \) imperfectly. This is possible by using a form of assumed certainty equivalence: the pursuer’s base policy and the evader’s maximization can be computed by using an estimate of the current location \( z_2 \) instead of the unknown true location.

In the preceding pursuit-evasion example, the choice of the base policy was facilitated by the special structure of the problem. Generally, however, finding a suitable base policy whose cost function \( J_\mu \) can be conveniently computed is an important problem-dependent issue. Finally, let us note that like all rollout algorithms, the minimax rollout algorithm may be well-suited for on-line replanning in problems where data may be changing or be revealed during the process of path construction.

### 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 book, we will adopt the original intended meaning: policy improvement starting from a base policy, which is evaluated with some form of Monte Carlo simulation and/or optimization calculation.

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 for 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 Castanón [BeC99]. A survey of rollout in discrete optimization is given by the author in [Ber13a].

The expert-based rollout in the form given in Section 2.3.4 was first discussed in the RL book [Ber19a], but it is inspired by the method of *comparison training* proposed by Tesauro [Tes89a], [Tes89b], [Tes01], and
subsequently used by several other authors (for some recent references, see [DNW16], [TCW19]). This is a general method for training an approximation architecture to choose between two alternatives, using a dataset of expert choices in place of an explicit cost function.

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], McGovern, Moss, and Barto [MMB02], Savagaonkar, Givan, and Chong [SGC02], Bertsimas and Popescu [BeP03], Guerriero and Mancini [GuM03], Tu and Pattipati [TuP03], Wu, Chong, and Givan [WCG03], Chang, Givan, and Chong [CGC04], Meloni, Pacciarelli, and Pranzer [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], Goodson, Thomas, and Ohlmann [GTO15], Khashoeei, Antunes, and Heemels [KAH15], Li and Womer [LiW15], Mastin and Jaillet [MaJ15], Huang, Jia, and Guan [HJG16], Sinroth, Holfeld, and Brunsch [SHB15], Lan, Guan, and Wu [LGW16], Ulmer [Ulm17], Bertazzi and Secomandi [BeS18], Guerriero, Di Puglia, and Macrina [GDM19], Sarkale et al. [SNC18], Ulmer at al. [UGM18], Arcari, Hewing, and Zeilinger [AHZ19], 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 favorable computational experience. Collectively, these works together with the extensive positive experience with model predictive control make a convincing case for the reliability of the rollout approach.

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) and the corresponding analysis were given in the author’s paper [Ber97b]. The idea of using Q-factor differences as the basis for control selection will be revisited in Sections 3.3.3 and 3.3.4, where we will discuss advantage updating and the use of training to construct cost function difference approximations.

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. Some early widely cited papers are Clarke, Mohtadi, and Tuffs [CMT87a], [CMT87b], and Keerthi and Gilbert [KeG88]. 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], Goodwin, Seron, and De Dona [GSD06], and Camacho and Bordons [CaoB07]. The recent textbooks by Kouvaritakis and Cannon [KoC16], Borrelli, Bemporad, and Morari [BBM17], and Rawlings, Mayne, and Diehl [RMD17] collectively provide an extensive and comprehensive view of the current state of the field.

Some authors divide MPC methods into several categories, such as classical (dealing with deterministic settings), robust (allowing for set membership uncertainty), and stochastic (allowing for stochastic uncertainty). We have primarily focused on the better-developed classical case, although several of the general RL methods that we discuss are well-suited for other types of MPC settings. Moreover, our treatment is quite general in that we allow arbitrary state and control spaces, i.e., discrete as well as continuous.

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. It also highlights the connection between the sequential improvement property in rollout and stability guarantees in MPC. For example, thanks to the use of sequential improvement concepts, the stability analysis of Section 2.5.1 allows general control constraint sets including those that are discrete/integer, and shows that control space continuity is not relevant for stability analysis except for special problem structures. Moreover, discrete control constraints that couple the controls of multiple periods, which are common in the control of hybrid systems, can be dealt with by using the constrained rollout ideas of Section 2.7.

The first stability analysis of MPC was given by Keerthi and Gilbert [KeG88]. The author’s review paper [Ber05a] gave a stability analysis that applies to problems with general state and control spaces, based on the MPC connection with rollout. A stability analysis with discrete constraint
sets was given by Rawlings and Risbeck [RaR17]. The paper by Krener [Kre19] discusses methods to estimate the optimal cost function for use as terminal cost function, aiming to achieve stabilization with MPC lookahead that is as small as possible (cf. Section 2.5.3). 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 terminal cost function satisfies a sequential improvement condition such as Eq. (2.34) (sometimes also referred to as a “Lyapunov condition” in the control literature); see Section 2.5.3.

Problems of MPC with set membership or stochastic uncertainty and state constraints are more challenging than deterministic ones. A major difficulty is to guarantee 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 author’s paper [Ber05a] and textbook [Ber17], Section 6.4, describe 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. The terms “reachability” and “minimax reachability” were introduced by the author in his thesis; the terms “invariant sets,” “invariant tube,” and others have also been used with the same meaning. In particular, the paper by Kolmanovsky and Gilbert [KoG98], uses the term “disturbance invariant set.” The survey by Blanchini [Bla99], and the book by Blanchini and Miani [BlM08] use both terms “invariance” and “reachability,” and extend these notions to continuous-time systems. For a broad discussion of issues of set membership uncertainty and minimax control in the context of MPC, among others, see the survey by Mayne [May14]. Reachability for continuous-time games has been studied by Mitchell, Bayen, and Tomlin [MBT05]. For an alternative approach to reachability, which is based on the notion of Conditional Value-at-Risk (CVaR), see Chapman et al. [CLT19].

Methods for MPC that are based on concepts of approximate constraint violation (cf. Example 2.5.6) have been given in several works. The general idea is to compute simultaneous cost function and target tube approximations in either a single policy iteration/rollout framework or a multiple policy iteration/learning framework of the type to be discussed in Chapter 4. For some recent references, see Ames et al. [AXG16], Hewing et al. [HWM19], Hewing and Zeilinger [HeZ19], Rosolia and Borrelli [RoB20a], [RoB20b].

Sections 2.6: The material on multiagent rollout comes from the author’s recent paper [Ber19c]. This paper also discusses extensions to infinite horizon policy iteration algorithms, and explores the connections with the concept of person-by-person optimality from team theory. This con-
cept has a long history (Marschak [Mar55], Radner [Rad62], Witsenhausen [Wit71]), and was researched extensively in the 70s; see the review paper by Ho [Ho80], and the book by Zoppoli, Parisini, Baglietto, and Sanguineti [ZPB19] for a recent presentation.

We note that the term “multiagent” has been used with several different meanings in the literature. For example some authors place emphasis on the case where the agents do not have common information when selecting their decisions. This gives rise to sequential decision problems with “nonclassical information patterns,” which can be very complex, partly because they cannot be addressed by exact DP. In our discussion, we follow a different approach and starting point. We first consider multiagent problems where the agents have common/perfect state information, which can be addressed in principle by exact DP. We then derive RL methods within this context, and we address more complex (nonclassical information pattern) problems with forms of certainty equivalence, whereby each agent applies controls obtained by a common/perfect state information method, but with the state replaced by an estimate.

The discussion of the multiagent form of MPC in Section 2.6.2 is new and relies on the view of MPC as a rollout algorithm. This view also allows the use of the constrained form of rollout of Section 2.7 within the context of MPC.

Sections 2.7: The material on state-constrained rollout in Section 2.7.1 is new, and has been inspired by the papers by Rosolia and Borrelli [RoB20a], [RoB20b], which introduced the idea of simultaneous iterative construction of safe sets and value functions in MPC. The constrained rollout algorithm and its analysis, including the notions of sequential consistency and sequential improvement, was first given in the author’s MPC review paper [Ber05a]. The author’s recent paper [Ber20a] provided the extension to multiagent rollout. For a discussion of constrained DP problems, the reader may consult the 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 (2.47), using among others, multiobjective optimization ideas; see e.g., Jaffe [Jaf84], Martins [Mar84], Guerriero 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 $\overline{u}$, 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 focuses on graph-related applications and 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. While random search methods typically offer a theoretical guarantee of convergence to a global optimum, 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]. These problems involve finding a shortest path in a directed graph where the transitions between nodes are subject to uncertainty. Here the objective is to guarantee that a special destination state is reached with a minimum cost path under the worst possible instance of the uncertainty. Section 5.4 of the paper includes a discussion of related rollout algorithms.

Minimax shortest path problems are most fruitfully viewed within the context of infinite horizon, since in the presence of uncertainty, it may not be possible to specify a finite horizon within which all policies are guaranteed to reach the termination state. The paper [Ber19f] develops value iteration, policy iteration, and Dijkstra-like finitely terminating algorithms for exact solution of these problems.

Robust shortest path problems arise, among others, in planning and pursuit-evasion contexts. They also arise in MPC-type schemes involving set membership uncertainty. Within this context, the MPC lookahead minimization can be viewed as a robust shortest path planning problem. This viewpoint, however, requires, that the state and control spaces are either discrete or are suitably discretized.