Approximation in Value Space
- Rollout Algorithms

Contents

2.1. Deterministic Discrete Spaces Finite Horizon Problems . p. 4
2.2. Approximation in Value Space . . . . . . . . . . . . . . . p. 13
2.3. Rollout Algorithms for Discrete Optimization . . . . . . p. 14
  2.3.1. Cost Improvement with Rollout - Sequential Consistency,
    Sequential Improvement . . . . . . . . . . . . . . . p. 19
  2.3.2. The Fortified Rollout Algorithm . . . . . . . . . . p. 26
  2.3.3. Using Multiple Base Heuristics - Parallel Rollout . . . p. 29
  2.3.4. Simplified Rollout Algorithms . . . . . . . . . . . p. 30
  2.3.5. Truncated Rollout with Terminal Cost Approximation p. 31
  2.3.6. Model-Free Rollout . . . . . . . . . . . . . . . p. 32
2.4. Rollout and Approximation in Value Space with Multistep .
  Lookahead . . . . . . . . . . . . . . . . . . . . . p. 36
  2.4.1. Iterative Deepening Using Forward Dynamic Programming . . . p. 42
  2.4.2. Incremental Multistep Rollout . . . . . . . . . . p. 44
2.5. Constrained Forms of Rollout Algorithms . . . . . . . p. 46
  2.5.1. Constrained Rollout for Discrete Optimization and Integer
    Programming . . . . . . . . . . . . . . . . . . . . p. 58
  Rollout . . . . . . . . . . . . . . . . . . . . . p. 62
2.7. Stochastic Rollout and Monte Carlo Tree Search . . . . p. 70
2.7.1. Simplified Rollout and Policy Iteration . . . . . . . p. 74
2.7.2. Certainty Equivalence Approximations . . . . . . . p. 75
2.7.3. Simulation-Based Implementation of the Rollout . . . .
   Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 76
2.7.4. Variance Reduction in Rollout - Comparing Advantagesp. 79
2.7.5. Monte Carlo Tree Search . . . . . . . . . . . . . . . . . . . . . . . . . p. 82
2.7.6. Randomized Policy Improvement by Monte Carlo . . .
   Tree Search . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 85
2.8. Rollout for Infinite-Spaces Problems - Optimization . . . .
   Heuristics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 86
2.8.1. Rollout for Infinite-Spaces Deterministic Problems . . p. 86
2.8.2. Rollout Based on Stochastic Programming . . . . . . . p. 90
2.8.3. Stochastic Programming with Certainty Equivalence . p. 93
2.9. Multiagent Rollout . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 94
2.9.1. Asynchronous and Autonomous Multiagent Rollout p. 105
2.10. Rollout for Bayesian Optimization and Sequential . . .
   Estimation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 109
2.11. Adaptive Control by Rollout with a POMDP . . . . . . .
   Formulation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 120
2.12. Rollout for Minimax Control . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 128
2.13. Notes, Sources, and Exercises . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 136
In this chapter, we discuss various aspects of approximation in value space and rollout algorithms, focusing primarily on the case where the state and control spaces are finite. In Sections 2.1-2.6, we consider finite horizon deterministic problems, which in addition to arising often in practice, offer some important advantages in the context of RL. In particular, a finite horizon is well suited for the use of rollout, while the deterministic character of the problem eliminates the need for costly on-line Monte Carlo simulation.

An interesting aspect of our methodology for discrete deterministic problems is that it admits extensions that we have not discussed so far. The extensions include multistep lookahead variants, as well as variants that apply to constrained forms of DP, which involve constraints on the entire system trajectory, and also allow the use of heuristic algorithms that are more general than policies within the context of rollout. These variants rely on the problem’s deterministic structure, and do not extend to stochastic problems.

Another interesting aspect of finite state deterministic problems is that they can serve as a framework for an important class of commonly encountered discrete optimization problems, including integer programming and combinatorial optimization problems such as scheduling, assignment, routing, etc. This brings to bear the methodology of approximation in value space, rollout, adaptive control, and MPC, and provides effective suboptimal solution methods for these problems.

In Sections 2.7-2.11, we consider various problems that involve stochastic uncertainty. In Section 2.12, we consider minimax problems that involve set membership uncertainty. The present chapter draws heavily on Chapters 2 and 3 of the book [Ber20a], and Chapter 6 of the book [Ber22a]. These books may be consulted for more details and additional examples.

While our focus in this chapter will be on finite horizon problems, our discussion applies to infinite horizon problems as well, because approximation in value space and rollout are essentially finite-stages algorithms, while the nature of the original problem horizon (be it finite or infinite) affects only the terminal cost function approximation. Thus in implementing approximating one-step or multistep approximation in value space, it makes little difference whether the original problem has finite or infinite horizon. At the same time, for conceptual purposes, we can argue that finite horizon problems, even when they involve a nonstationary system and cost per stage, can be transformed to infinite horizon problems, by introducing an artificial cost-free termination state that the system moves into at the end of the horizon; see Section 1.6.2. Through this transformation, the synergy of off-line training and on-line play based on Newton’s method is brought to bear, and the insights that we discussed in Chapter 1 in the context of an infinite horizon apply and explain the good performance of our methods in practice.
2.1 DETERMINISTIC DISCRETE SPACES FINITE HORIZON PROBLEMS

We recall from Chapter 1, Section 1.2, that in deterministic finite horizon DP problems, the state is generated nonrandomly over $N$ stages, through a system equation of the form

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

where $k$ is the time index, and $x_k$ is the state of the system, an element of some state space $X_k$, $u_k$ is the control or decision variable, to be selected at time $k$ from some given set $U_k(x_k)$, a subset of a control space $U_k$, that depends on $x_k$, $f_k$ is a function of $(x_k, u_k)$ that describes the mechanism by which the state is updated from time $k$ to time $k + 1$.

The state space $X_k$ and control space $U_k$ are arbitrary sets and may depend on $k$. Similarly, the system function $f_k$ can be arbitrary and may depend on $k$. The cost incurred at time $k$ is denoted by $g_k(x_k, u_k)$, and the function $g_k$ may depend on $k$. For a given initial state $x_0$, the total cost of a control sequence $\{u_0, \ldots, u_{N-1}\}$ is

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

where $g_N(x_N)$ is a terminal cost incurred at the end of the process. This is a well-defined number, since the control sequence $\{u_0, \ldots, u_{N-1}\}$ together with $x_0$ determines exactly the state sequence $\{x_1, \ldots, x_N\}$ via the system equation (2.1); see Figure 2.0.1. We want to minimize the cost (2.2) over all sequences $\{u_0, \ldots, u_{N-1}\}$ that satisfy the control constraints, thereby obtaining the optimal value as a function of $x_0$

$$J^*(x_0) = \min_{u_k \in U_k(x_k)} \max_{k=0, \ldots, N-1} J(x_0; u_0, \ldots, u_{N-1}).$$
Notice an important difference from the stochastic case: we optimize over sequences of controls \(\{u_0, \ldots, u_{N-1}\}\), rather than over policies that consist of a sequence of functions \(\pi = \{\mu_0, \ldots, \mu_{N-1}\}\), where \(\mu_k\) maps states \(x_k\) into controls \(u_k = \mu_k(x_k)\), and satisfies the control constraints \(\mu_k(x_k) \in U_k(x_k)\) for all \(x_k\). It is well-known that in the presence of stochastic uncertainty, policies are more effective than control sequences, and can result in improved cost. On the other hand for deterministic problems, minimizing over control sequences yields the same optimal cost as over policies, since the cost of any policy starting from a given state determines with certainty the controls applied at that state and the future states, and hence can also be achieved by the corresponding control sequence. This point of view allows more general forms of rollout, which we will discuss in this chapter: instead of using a policy for rollout, we will allow the use of more general heuristics for choosing future controls.

**The Exact DP Algorithm**

We recall from Chapter 1, Section 1.2, the DP algorithm for finite horizon deterministic problems. It constructs functions

\[
J^*_0(x_0), \ldots, J^*_{N-1}(x_{N-1}), J^*_N(x_N),
\]

sequentially, starting from \(J^*_N\), and proceeding backwards to \(J^*_{N-1}, J^*_{N-2}\), etc. The value \(J^*_k(x_k)\) will be viewed as the optimal cost of the tail subproblem that starts at state \(x_k\) at time \(k\) and ends at some state \(x_N\).

**DP Algorithm for Deterministic Finite Horizon Problems**

Start with

\[
J^*_N(x_N) = g_N(x_N), \quad \text{for all } x_N, \quad (2.3)
\]

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

\[
J^*_k(x_k) = \min_{u_k \in U_k(x_k)} \left[ g_k(x_k, u_k) + J^*_{k+1}(f_k(x_k, u_k)) \right], \quad \text{for all } x_k. \quad (2.4)
\]

Note that at stage \(k\), the calculation in Eq. (2.4) must be done for all states \(x_k\) before proceeding to stage \(k - 1\). The key fact about the DP algorithm is that for every initial state \(x_0\), the number \(J^*_0(x_0)\) obtained at the last step, is equal to the optimal cost \(J^*(x_0)\). Indeed, a more general fact was shown in Section 1.2, namely that for all \(k = 0, 1, \ldots, N - 1\), and all states \(x_k\) at time \(k\), we have

\[
J^*_k(x_k) = \min_{u_m \in U_m(x_m)} J(x_k; u_k, \ldots, u_{N-1}), \quad (2.5)
\]
where \( J(x_k; u_k, \ldots, u_{N-1}) \) is the cost generated by starting at \( x_k \) and using subsequent controls \( u_k, \ldots, u_{N-1} \):

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

Thus, \( J^*_k(x_k) \) is the optimal cost for an \((N - k)\)-stage tail subproblem that starts at state \( x_k \) and time \( k \), and ends at time \( N \). Based on this interpretation of \( J^*_k(x_k) \), we call it the optimal cost-to-go from state \( x_k \) at stage \( k \), and refer to \( J^*_k \) as the optimal cost-to-go function or optimal cost function at time \( k \).

We have also discussed in Section 1.2 the construction of an optimal control sequence. Once the functions \( J^*_0, \ldots, J^*_N \) have been obtained, we can use a forward algorithm to construct an optimal control sequence \( \{u^*_0, \ldots, u^*_{N-1}\} \) and state trajectory \( \{x^*_1, \ldots, x^*_N\} \) for a given initial state \( x_0 \).

### Construction of Optimal Control Sequence \( \{u^*_0, \ldots, u^*_{N-1}\} \)

Set

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

and

\[
x^*_1 = f_0(x_0, u^*_0).
\]

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

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

and

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

Note an interesting conceptual division of the optimal control sequence construction: there is off-line training to obtain \( J^*_k \) by precomputation [cf. the DP Eqs. (2.3)-(2.4)], which is followed by on-line play to obtain \( u^*_k \) [cf. Eq. (2.6)]. This is analogous to the two algorithmic processes described in Section 1.1 in connection with computer chess and backgammon.

### Finite-State Deterministic Problems

For the first five sections of this chapter, we will consider the case where the state and control spaces are discrete and consist of a finite number of
Sec. 2.1 Deterministic Discrete Spaces Finite Horizon Problems

Figure 2.1.1 Illustration of a deterministic finite-state DP problem. Nodes correspond to states \( x_k \). Arcs correspond to state-control pairs \( (x_k, u_k) \). An arc \( (x_k, u_k) \) has start and end nodes \( x_k \) and \( x_{k+1} = f_k(x_k, u_k) \), respectively. The cost \( g_k(x_k, u_k) \) of the transition is the length of this arc. An artificial terminal node \( t \) is connected with an arc of cost \( g_N(x_N) \) with each state \( x_N \). The problem is equivalent to finding a shortest path from initial nodes of stage 0 to node \( t \).

elements. As we have noted in Section 1.2, such problems can be described with an acyclic graph specifying for each state \( x_k \) the possible transitions to next states \( x_{k+1} \). The nodes of the graph correspond to states \( x_k \) and the arcs of the graph correspond to state-control pairs \( (x_k, u_k) \). Each arc with start node \( x_k \) corresponds to a choice of a single control \( u_k \in U_k(x_k) \) and has as end node the next state \( f_k(x_k, u_k) \). The cost of an arc \( (x_k, u_k) \) is defined as \( g_k(x_k, u_k) \); see Fig. 2.1.1. To handle the final stage, an artificial terminal node \( t \) is added. Each state \( x_N \) at stage \( N \) is connected to the terminal node \( t \) with an arc having cost \( g_N(x_N) \). The control sequences \( \{u_0, \ldots, u_{N-1}\} \) correspond to paths originating at the initial state (a node at stage 0) and terminating at one of the nodes corresponding to the final stage \( N \). With this description it can be seen that a deterministic finite-state finite-horizon problem is equivalent to finding a minimum-length (or shortest) path from the initial nodes of the graph (stage 0) to the terminal node \( t \), as we have discussed in Section 1.2.

Shortest path problems arise in a great variety of application domains. While there are quite a few efficient polynomial algorithms for solving them, some practical shortest path problems are extraordinarily difficult because they involve an astronomically large number of nodes. For example deterministic scheduling problems of the type discussed in Example 1.2.1 can be formulated as shortest path problems, but with a number of nodes that grows exponentially with the number of tasks. For such problems neither exact DP nor any other shortest path algorithm can compute an exact optimal solution in practice. In what follows, we will aim to show that suboptimal solution methods, and rollout algorithms in particular, offer a viable alternative.

Many types of search problems involving games and puzzles also ad-
mit in principle exact solution by DP, but have to be solved by suboptimal methods in practice. The following is a characteristic example.

**Example 2.1.1 (The Four Queens Problem)**

Four queens must be placed on a $4 \times 4$ portion of a chessboard so that no
Sec. 2.1 Deterministic Discrete Spaces Finite Horizon Problems

queen can attack another. In other words, the placement must be such that every row, column, or diagonal of the $4 \times 4$ board contains at most one queen. Equivalently, we can view the problem as a sequence of problems; first, placing a queen in one of the first two squares in the top row, then placing another queen in the second row so that it is not attacked by the first, and similarly placing the third and fourth queens. (It is sufficient to consider only the first two squares of the top row, since the other two squares lead to symmetric positions; this is an example of a situation where we have a choice between several possible state spaces, but we select the one that is smallest.)

We can associate positions with nodes of an acyclic graph where the root node $s$ corresponds to the position with no queens and the terminal nodes correspond to the positions where no additional queens can be placed without some queen attacking another. Let us connect each terminal position with an artificial terminal node $t$ by means of an arc. Let us also assign to all arcs cost zero except for the artificial arcs connecting terminal positions with less than four queens with the artificial node $t$. These latter arcs are assigned a cost of 1 (see Fig. 2.1.2) to express the fact that they correspond to dead-end positions that cannot lead to a solution. Then, the four queens problem reduces to finding a minimal cost path from node $s$ to node $t$, with an optimal sequence of queen placements corresponding to cost 0.

Note that once the states/nodes of the graph are enumerated, the problem is essentially solved. In this $4 \times 4$ problem the states are few and can be easily enumerated. However, we can think of similar problems with much larger state spaces. For example consider the problem of placing $N$ queens on an $N \times N$ board without any queen attacking another. Even for moderate values of $N$, the state space for this problem can be extremely large (for $N = 8$ the number of possible placements with exactly one queen in each row is $8^8 = 16,777,216$). It can be shown that there exist solutions to the $N$ queens problem for all $N \geq 4$ (for $N = 2$ and $N = 3$, clearly there is no solution). Moreover effective (non-DP) search algorithms have been devised for its solution up to very large values of $N$.

The preceding example illustrates some of the difficulties of applying exact DP to discrete/combinatorial problems with the type of formulation that we have described. The state space typically becomes very large, particularly as $k$ increases. In the preceding example, to start a backward DP algorithm, we need to consider all the possible terminal positions, which are too many when $N$ is large. There is an alternative exact DP algorithm for deterministic problems, which proceeds forwards from the initial state. It is simply the backward DP algorithm applied to an equivalent shortest path problem, derived form one of Fig. 2.1.1 by reversing the directions of all the arcs, and exchanging the roles of the origin and the destination. It will be discussed in Section 2.4; see also [Ber17a], Chapter 2. Still, however, this forward DP algorithm cannot overcome the difficulty with a very large state space.
General Discrete Optimization Problems

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

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

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

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

We assume that each solution \(u\) has \(N\) components; i.e., it has the form \(u = (u_0, \ldots, u_{N-1})\), where \(N\) is a positive integer. We can then view the problem as a sequential decision problem, where the components \(u_0, \ldots, u_{N-1}\) are selected one-at-a-time. A \(k\)-tuple \((u_0, \ldots, u_{k-1})\) consisting of the first \(k\) components of a solution is called a \(k\)-solution. We associate \(k\)-solutions with the \(k\)th stage of the finite horizon discrete optimal control problem shown in Fig. 2.1.3. In particular, for \(k = 1, \ldots, N\), we view as the states of the \(k\)th stage all the \(k\)-tuples \((u_0, \ldots, u_{k-1})\). For stage \(k = 0, \ldots, N-1\), we view \(u_k\) as the control. The initial state is an artificial state denoted \(s\). From this state, by applying \(u_0\), we may move to any “state” \((u_0)\), with \(u_0\) belonging to the set

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

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

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

\[
U_k(u_0, \ldots, u_{k-1}) = \{ u_k \mid \text{for some } \overline{u}_{k+1}, \ldots, \overline{u}_{N-1} \text{ we have} \}
\]

\[
(u_0, \ldots, u_{k-1}, u_k, \overline{u}_{k+1}, \ldots, \overline{u}_{N-1}) \in U. \tag{2.8}
\]

These are the choices of \(u_k\) that are consistent with the preceding choices \(u_0, \ldots, u_{k-1}\), and are also consistent with feasibility. The last stage corresponds to the \(N\)-solutions \(u = (u_0, \ldots, u_{N-1})\), and the terminal cost is \(G(u)\); see Fig. 2.1.3. All other transitions in this DP problem formulation have cost 0.
Let $J^*_k(u_0, \ldots, u_{k-1})$ denote the optimal cost starting from the $k$-solution $(u_0, \ldots, u_{k-1})$, i.e., the optimal cost of the problem over solutions whose first $k$ components are constrained to be equal to $u_0, \ldots, u_{k-1}$. The DP algorithm is described by the equation

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

with the terminal condition

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

This algorithm executes backwards in time: starting with the known function $J^*_N = G$, we compute $J^*_{N-1}$, then $J^*_{N-2}$, and so on up to computing $J^*_0$. An optimal solution $(u^*_0, \ldots, u^*_{N-1})$ is then constructed by going forward through the algorithm

$$u^*_k \in \arg\min_{u_k \in U_k(u^*_0, \ldots, u^*_{k-1})} J^*_{k+1}(u^*_0, \ldots, u^*_{k-1}, u_k), \quad k = 0, \ldots, N-1, \quad (2.9)$$

where $U_0$ is given by Eq. (2.7), and $U_k$ is given by Eq. (2.8); first compute $u^*_0$, then $u^*_1$, and so on up to $u^*_{N-1}$; cf. Eq. (2.6).

Of course here the number of states typically grows exponentially with $N$, but we can use the DP minimization (2.9) as a starting point for approximation methods. For example we may try to use approximation in value space, whereby we replace $J^*_{k+1}$ with some suboptimal $\tilde{J}^*_{k+1}$ in Eq. (2.9). One possibility is to use as

$$\tilde{J}^*_{k+1}(u^*_0, \ldots, u^*_{k-1}, u_k),$$

Figure 2.1.3 Formulation of a discrete optimization problem as a DP problem with $N$ stages. There is a cost $G(u)$ only at the terminal stage on the arc connecting an $N$-solution $u = (u_0, \ldots, u_{N-1})$ upon reaching the terminal state. Note that there is only one incoming arc at each node.
the cost generated by a heuristic method that solves the problem sub-
optimally with the values of the first \( k + 1 \) decision components fixed at \( u_0^*, \ldots, u_{k-1}^*, u_k \). This is the rollout algorithm, which turns out to be a very simple and effective approach for approximate combinatorial optimization.

Let us finally note that while we have used a general cost function \( G \) and constraint set \( U \) in our discrete optimization model of this section, in many problems \( G \) and/or \( U \) may have a special (e.g., additive) structure, which is consistent with a sequential decision making process and may be computationally exploited. The traveling salesman Example 1.2.2 is a case in point, where \( G \) consists of the sum of \( N \) components (the intercity travel costs), one per stage.

**Constraint Programming**

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

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

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

Constraint programming problems can of course be formulated as DP problems using our earlier formulation (cf. Fig. 2.1.3). They can also be transformed into equivalent unconstrained (or less constrained) problems by using problem-dependent penalty functions that eliminate constraints while quantifying the level of constraint violation. As an illustration, the problem of finding a feasible solution of the system of constraints

\[
h_k(u_k, u_{k+1}) \leq 0, \quad k = 0, \ldots, N-1, \\
u_k \in U_k, \quad k = 0, \ldots, N-1,
\]

can be transformed into the equivalent DP problem of minimizing

\[
\sum_{k=1}^N \max \{0, h_k(x_k, u_k)\},
\]

subject to the system equation \( x_{k+1} = u_k \), and the control constraints \( u_k \in U_k, k = 0, \ldots, N-1 \). Other penalty functions can also be used, such as a quadratic; see the author’s nonlinear programming text [Ber16]. This approach is convenient, but it offers no guarantee that it can find a complete feasible solution \((u_0, \ldots, u_{N-1})\), even if one exists. It simply aims to minimize (suboptimally) a measure of the total constraint violation. However, in the process it may be able to find a complete feasible solution.
2.2 APPROXIMATION IN VALUE SPACE

The forward optimal control sequence construction of Eq. (2.6) is possible only after we have computed $J^*_k(x_k)$ by DP for all $x_k$ and $k$. Unfortunately, in practice this is often prohibitively time-consuming. However, a similar forward algorithmic process can be used if the optimal cost-to-go functions $J^*_k$ are replaced by some approximations $\tilde{J}_k$. This is the idea of approximation in value space that we discussed in Section 1.2.3. It constructs a suboptimal solution \( \{\tilde{u}_0, \ldots, \tilde{u}_{N-1}\} \) in place of the optimal \( \{u^*_0, \ldots, u^*_{N-1}\} \), by using $\tilde{J}_k$ in place of $J^*_k$ in the DP procedure (2.6).

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

Start with

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

and set

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

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

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

and

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

The expression

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

which is minimized in approximation in value space [cf. Eq. (2.10)] is known as the (approximate) Q-factor of $(x_k, u_k)$. Note that the computation of the suboptimal control (2.10) can be done through the Q-factor minimization

$$\tilde{u}_k \in \arg\min_{u_k \in U_k(\tilde{x}_k)} \tilde{Q}_k(\tilde{x}_k, u_k).$$

This suggests the possibility of using approximate off-line trained Q-factors in place of cost functions in approximation in value space schemes. However, contrary to the cost approximation scheme (2.10) and its multistep counterparts, the performance may be degraded through the errors in the off-line training of the Q-factors (depending on how the training is done).
Multistep Lookahead

The approximation in value space algorithm (2.10) involves a one-step lookahead minimization, since it solves a one-stage DP problem for each \( k \). We may also consider \( \ell \)-step lookahead, which involves the solution of an \( \ell \)-step deterministic DP problem, where \( \ell \) is an integer, \( 1 < \ell < N - k \), with a terminal cost function approximation \( \tilde{J}_{k+\ell} \).

As we have noted in Chapter 1, multistep lookahead typically provides better performance over one-step lookahead in approximation in value space schemes. For example in AlphaZero chess, long multistep lookahead is critical for good on-line performance. On the negative side, the solution of the multistep lookahead optimization problem is more time consuming than its one-step lookahead counterpart. However, the deterministic character of the lookahead minimization problem and the fact that it is solved for the single initial state \( x_k \) at each time \( k \) helps to limit the growth of the lookahead tree and to keep the computation manageable.

2.3 ROLLOUT ALGORITHMS FOR DISCRETE OPTIMIZATION

The construction of suitable approximate cost-to-go functions \( \tilde{J}_{k+1} \) for approximation in value space can be done in many different ways, including some of the principal RL methods. A method of particular interest for our course is rollout, whereby the approximate values \( \tilde{J}_{k+1}(x_{k+1}) \) in Eq. (2.10) are obtained when needed by running for each \( u_k \in U_k(x_k) \) a heuristic control scheme, called base heuristic, for a suitably large number of steps, starting from \( x_{k+1} = f_k(x_k, u_k) \).

The base heuristic can be any method, which starting from a state \( x_{k+1} \) generates a sequence of controls \( u_{k+1}, \ldots, u_{N-1} \), the corresponding sequence of states \( x_{k+2}, \ldots, x_N \), and the cost of the heuristic starting from \( x_{k+1} \), which we will generically denote by \( H_{k+1}(x_{k+1}) \) in this chapter:

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

This value of \( H_{k+1}(x_{k+1}) \) is the one used as the approximate cost-to-go \( \tilde{J}_{k+1}(x_{k+1}) \) in the corresponding approximation in value space scheme (2.10).

In this section, we will develop in more detail the theory of rollout with one-step lookahead minimization for deterministic problems, including the important issue of cost improvement. We will also illustrate several variants of the method, and we will consider questions of efficient implementation. We will then discuss examples of discrete optimization applications.

Let us consider a deterministic DP problem with a finite number of controls and a given initial state (so the number of states that can be reached from the initial state is also finite). We first focus on the pure
Section 2.3 Rollout Algorithms for Discrete Optimization

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

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

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

This form of rollout that uses one-step lookahead without truncation, and hence no terminal cost approximation. Given a state \( x_k \) at time \( k \), this algorithm considers the tail subproblems that start at every possible next state \( x_{k+1} \), and solves them suboptimally with the base heuristic.

Thus when at \( x_k \), rollout generates on-line the next states \( x_{k+1} \) that correspond to all \( u_k \in U_k(x_k) \), and uses the base heuristic to compute the sequence of states \( \{x_{k+1}, \ldots, x_N\} \) and controls \( \{u_{k+1}, \ldots, u_{N-1}\} \) such that

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

and the corresponding cost

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

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

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

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

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

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

\[
\tilde{Q}_k(x_k, u_k) = g_k(x_k, u_k) + H_{k+1}(f_k(x_k, u_k)); \tag{2.12}
\]
The rollout algorithm thus defines a suboptimal policy \( \tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\} \), referred to as the rollout policy, where for each \( x_k \) and \( k \), \( \tilde{\mu}_k(x_k) \) is the control produced by the Q-factor minimization (2.11).

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

In Section 1.2 we considered an example of rollout involving the traveling salesman problem and the nearest neighbor heuristic (cf. Examples 1.2.2 and 1.2.3). Let us consider another example, which involves a classical discrete optimization problem.

**Example 2.3.1 (Multi-Vehicle Routing)**

Consider \( m \) vehicles that move along the arcs of a given graph. Some of the nodes of the graph include a task to be performed by the vehicles. Each task will be performed only once, immediately after some vehicle reaches the corresponding node for the first time. We assume a horizon that is large enough to allow every task to be performed. The problem is to find a route for each vehicle so that the tasks are collectively performed by the vehicles in a minimum number of moves. To express this objective, we assume that for each move by a vehicle there is a cost of one unit. These costs are summed up to the point where all the tasks have been performed.

For a large number \( m \) of vehicles and a complicated graph, this is a nontrivial combinatorial problem. It can be approached by DP, like any discrete deterministic optimization problem, as we have discussed. In particular, we can view as state at a given stage the \( m \)-tuple of current positions of the vehicles together with the list of pending tasks. Unfortunately, however, the number of these states can be enormous (it increases exponentially with the number of tasks and the number of vehicles), so an exact DP solution is intractable.

This motivates an optimization in value space approach based on rollout. For this we need an easily implementable base heuristic that will solve suboptimally the problem starting from any state \( x_{k+1} \), and will provide the cost approximation \( \tilde{J}_{k+1}(x_{k+1}) \) in Eq. (2.10). One possibility is based on the vehicles choosing their actions selfishly and without coordination, along shortest paths to their nearest pending task.

To illustrate, consider the two-vehicle problem of Fig. 2.3.2. The base heuristic is to move each vehicle one step at a time towards its nearest pending task, until all tasks have been performed.

The rollout algorithm will work as follows. At a given state \( x_k \) [involving for example vehicle positions at the node pair (1, 2) and tasks at nodes 7 and 9, as in Fig. 2.3.2], we consider all possible joint vehicle moves (the controls \( u_k \) at the state) resulting in the node pairs (3,5), (4,5), (3,4), (4,4), corresponding
to the next states $x_{k+1}$ [thus, as an example (3,5) corresponds to vehicle 1 moving from 1 to 3, and vehicle 2 moving from 2 to 5]. We then run the base heuristic starting from each of these node pairs, and accumulate the incurred costs up to the time when both tasks are completed. For example starting from the vehicle positions/next state (3,5), the heuristic will produce the following sequence of moves:

- Vehicles 1 and 2 move from (3,5) to (6,2).
- Vehicles 1 and 2 move from (6,2) to (9,4), and the task at 9 is performed.
- Vehicles 1 and 2 move from (9,4) to (12,7), and the task at 7 is performed.

The two tasks are thus performed in a total of 6 vehicles moves once the move to (3,5) has been made.

The process of running the heuristic is repeated from the other three vehicle position pairs/next states (4,5), (3,4), (4,4), and the heuristic cost (number of moves) is recorded. We then choose the next state that corresponds to minimum cost. In our case the joint move to state $x_{k+1}$ that involves the pair (3,4) produces the sequence

- Vehicles 1 and 2 move from (3,4) to (6,7), and the task at 7 is performed.
- Vehicles 1 and 2 move from (6,7) to (9,4), and the task at 9 is performed.

and performs the two tasks in a total of 6 vehicle moves. It can be verified that it yields minimum first stage cost plus heuristic cost from the next state, as
Figure 2.3.3 Binary tree for the breakthrough problem. Each arc is either free or is blocked (crossed out in the figure). The problem is to find a path from the root to one of the leaves, which is free (such as the one shown with thick lines).

per Eq. (2.10). Thus, the rollout algorithm will choose to move the vehicles to state (3,4) from state (1,2). At that state the rollout process will be repeated, i.e., consider the possible next joint moves to the node pairs (6,7), (6,2), (6,1), (1,7), (1,2), (1,1), perform a heuristic calculation from each of them, compare, etc.

It can be verified that the rollout algorithm starting from the state (1,2) shown in Fig. 2.3.2 will attain the optimal cost (a total of 6 vehicle moves). It will perform much better than the heuristic, which starting from state (1,2), will move the two vehicles together to state (4,4), then to (7,7), then to (10,10), then to (12,12), and finally to (9,9), (a total of 10 vehicle moves). This is an instance of the cost improvement property of the rollout algorithm: it performs better than its base heuristic under appropriate conditions to be discussed next.

Let us finally note that the computation required by in rollout algorithm increases exponentially with the number $m$ of vehicles, since the number of $m$-tuples of moves at each stage increases exponentially with $m$. This is the type of problem where multiagent rollout can attain great computational savings; cf. Section 1.6.5, and the subsequent Section 2.9.

Here is an example of a search problem, whose exact solution complexity grows exponentially with the problem size, but can be addressed with a greedy heuristic as well as with the corresponding rollout algorithm.

**Example 2.3.2 (The Breakthrough Problem)**

Consider a binary tree with $N$ stages as shown in Fig. 2.3.3. Stage $k$ of the tree has $2^k$ nodes, with the node of stage 0 called root and the nodes of stage $N$ called leaves. There are two types of tree arcs: free and blocked. A free (or blocked) arc can (cannot, respectively) be traversed in the direction from
the root to the leaves. The objective is to break through the graph with a sequence of free arcs (a free path) starting from the root, and ending at one of the leaves. (A variant of this problem is to introduce a positive cost $c > 0$ for traversing a blocked arc, and 0 cost for traversing a free arc.)

One may use DP to discover a free path (if one exists) by starting from the last stage and by proceeding backwards to the root node. The $k$th step of the algorithm determines for each node of stage $N - k$ whether there is a free path from that node to some leaf node, by using the results of the preceding step. The amount of calculation at the $k$th step is $O(2^{N-k})$. Adding the computations for the $N$ stages, we see that the total amount of calculation is $O(N2^N)$, so it increases exponentially with the number of stages. For this reason it is interesting to consider heuristics requiring computation that is linear or polynomial in $N$, but may sometimes fail to determine a free path, even when a free path exists.

Thus, one may suboptimally use a greedy algorithm, which starts at the root node, selects a free outgoing arc (if one is available), and tries to construct a free path by adding successively nodes to the path. At the current node, if one of the outgoing arcs is free and the other is blocked, the greedy algorithm selects the free arc. Otherwise, it selects one of the two outgoing arcs according to some fixed rule that depends only on the current node (and not on the status of other arcs). Clearly, the greedy algorithm may fail to find a free path even if such a path exists, as can be seen from Fig. 2.3.3. On the other hand the amount of computation associated with the greedy algorithm is $O(N)$, which is much faster than the $O(N2^N)$ computation of the DP algorithm. Thus we may view the greedy algorithm as a fast heuristic, which is suboptimal in the sense that there are problem instances where it fails while the DP algorithm succeeds.

One may also consider a rollout algorithm that uses the greedy algorithm as the base heuristic. There is an analysis that compares the probability of finding a breakthrough solution with the greedy and with the rollout algorithm for random instances of binary trees (each arc is independently free or blocked with given probability $p$). This analysis is given in Section 6.4 of the book [Ber17a], and shows that asymptotically, the rollout algorithm requires $O(N)$ times more computation, but has an $O(N)$ times larger probability of finding a free path than the greedy algorithm.

This tradeoff is qualitatively typical: the rollout algorithm achieves a substantial performance improvement over the base heuristic at the expense of extra computation that is equal to the computation time of the base heuristic times a factor that is a low order polynomial of the problem size.

### 2.3.1 Cost Improvement with Rollout - Sequential Consistency, Sequential Improvement

The definition of the rollout algorithm leaves open the choice of the base heuristic. There are several types of suboptimal solution methods that can be used as base heuristics, such as greedy algorithms, local search, genetic algorithms, and others.
Intuitively, we expect that the rollout policy’s performance is no worse than the one of the base heuristic: since rollout optimizes over the first control before applying the heuristic, it makes sense to conjecture that it performs better than applying the heuristic without the first control optimization. However, some special conditions must hold in order to guarantee this cost improvement property. We provide two such conditions, *sequential consistency* and *sequential improvement*, introduced in the paper by Bertsekas, Tsitsiklis, and Wu [BTW97], and we later show how to modify the algorithm to deal with the case where these conditions are not met.

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

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

starting from state \(x_k\), it also generates the sequence

\[ \{x_{k+1}, u_{k+1}, \ldots, x_N\} \]

starting from state \(x_{k+1}\).

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

As an example, the reader may verify that the nearest neighbor heuristic described in the traveling salesman Example 1.2.3 and the heuristics used in the multivehicle routing Example 2.3.1 are sequentially consistent. Similar examples include the use of various types of greedy/myopic heuristics (Section 6.4 of the book [Ber17a] provides additional examples).† Generally most heuristics used in practice satisfy the sequential consistency condition at “most” states \(x_k\). However, some heuristics of interest may violate this condition at some states.

A sequentially consistent base heuristic can be recognized by the fact that it will apply the same control \(u_k\) at a state \(x_k\), no matter what position \(x_k\) occupies in a trajectory generated by the base heuristic. Thus a base

---

† A subtle but important point relates to how one breaks ties while implementing greedy base heuristics. For sequential consistency, one must break ties in a consistent way at various states, i.e., using a fixed rule at each state encountered by the base heuristic. In particular, randomization among multiple controls, which are ranked as equal by the greedy optimization of the heuristic, violates sequential consistency, and can lead to serious degradation of the corresponding rollout algorithm’s performance.
heuristic is sequentially consistent if and only if it defines a legitimate DP policy. This is the policy that moves from \( x_k \) to the state \( x_{k+1} \) that lies on the state trajectory \( \{ x_k, x_{k+1}, \ldots, x_N \} \) that the base heuristic generates. Similarly the policy moves from \( x_n \) to the state \( x_{n+1} \) for \( n = k+1, \ldots, N-1 \).

We will now show that the rollout algorithm obtained with a sequentially consistent base heuristic has a fundamental cost improvement property: it yields no worse cost than the base heuristic. The amount of cost improvement cannot be easily quantified, but is determined by the performance of the Newton step associated with the rollout policy, so it can be very substantial; cf. the discussion of Chapter 1.

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

\[
J_{k, \tilde{\pi}}(x_k) \leq H_k(x_k), \quad \text{for all } x_k \text{ and } k, \tag{2.13}
\]

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

**Proof:** We prove this inequality by induction. Clearly it holds for \( k = N \), since

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

Assume that it holds for index \( k+1 \). For any state \( x_k \), let \( \pi_k \) be the control applied by the base heuristic at \( x_k \). Then we have

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

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

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

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

\[
= H_k(x_k),
\]

where:

(a) The first equality is the DP equation for the rollout policy \( \tilde{\pi} \).
(b) The first inequality holds by the induction hypothesis.
(c) The second equality holds by the definition of the rollout algorithm.
(d) The third equality is the DP equation for the policy that corresponds to the base heuristic (this is the step where we need sequential consistency).
This completes the proof of the cost improvement property (2.13). Q.E.D.

Sequential Improvement

We will next show that the rollout policy has no worse performance than its base heuristic under a condition that is weaker than sequential consistence. Let us recall that the rollout algorithm $\tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$ is defined by the minimization

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

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

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

[cf. Eq. (2.12)], and $H_{k+1}(f_k(x_k, u_k))$ denotes the cost of the trajectory of the base heuristic starting from state $f_k(x_k, u_k)$.

**Definition 2.3.2:** We say that the base heuristic is *sequentially improving* if for all $x_k$ and $k$, we have

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

In words, the sequential improvement property (2.15) states that

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

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

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

which is greater or equal to the minimal Q-factor at $x_k$. This implies Eq. (2.15). A sequentially improving heuristic yields policy improvement as the next proposition shows.
Proposition 2.3.2: (Cost Improvement Under Sequential Improvement) Consider the rollout policy \( \tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\} \) obtained with a sequentially improving base heuristic, and let \( J_{k, \tilde{\pi}}(x_k) \) denote the cost obtained with \( \tilde{\pi} \) starting from \( x_k \) at time \( k \). Then

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

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

**Proof:** Follows from the calculation of Eq. (2.14), by replacing the last two steps (which rely on sequential consistency) with Eq. (2.15). Q.E.D.

Thus the rollout algorithm obtained with a sequentially improving base heuristic, will improve or at least will perform no worse than the base heuristic, from every starting state \( x_k \). In fact the algorithm has a monotonic improvement property, whereby it discovers a sequence of improved trajectories. In particular, let us denote the trajectory generated by the base heuristic starting from \( x_0 \) by

\[
T_0 = (x_0, u_0, \ldots, x_{N-1}, u_{N-1}, x_N),
\]

and the final trajectory generated by the rollout algorithm starting from \( x_0 \) by

\[
T_N = (x_0, \tilde{u}_0, \tilde{x}_1, \tilde{u}_1, \ldots, \tilde{x}_{N-1}, \tilde{u}_{N-1}, \tilde{x}_N).
\]

Consider also the intermediate trajectories generated by the rollout algorithm given by

\[
T_k = (x_0, \tilde{u}_0, \tilde{x}_1, \tilde{u}_1, \ldots, \tilde{x}_k, u_k, \ldots, x_{N-1}, u_{N-1}, x_N), \quad k = 1, \ldots, N - 1,
\]

where

\[
(\tilde{x}_k, u_k, \ldots, x_{N-1}, u_{N-1}, x_N),
\]

is the trajectory generated by the base heuristic starting from \( \tilde{x}_k \). Then, by using the sequential improvement condition, it can be proved (see Fig. 2.3.4) that

\[
\text{Cost of } T_0 \geq \cdots \geq \text{Cost of } T_k \geq \text{Cost of } T_{k+1} \geq \cdots \geq \text{Cost of } T_N. \tag{2.16}
\]

Empirically, it has been observed that the cost improvement obtained by rollout with a sequentially improving heuristic is typically considerable and often dramatic. In particular, many case studies, dating to the middle 1990s, indicate consistently good performance of rollout; see the last section of this chapter for a bibliography. The DP textbook [Ber17a] provides some detailed worked-out examples (Chapter 6, Examples 6.4.2, 6.4.5, 6.4.6, and
Figure 2.3.4 Proof of the monotonicity property (2.16). At $\tilde{x}_k$, the $k$th state generated by the rollout algorithm, we compare the “current” trajectory $T_k$ whose cost is the sum of the cost of the current partial trajectory $(x_0, \tilde{u}_0, \tilde{x}_1, \tilde{u}_1, \ldots, \tilde{x}_k)$ and the cost $H_k(\tilde{x}_k)$ of the base heuristic starting from $\tilde{x}_k$, and the trajectory $T_{k+1}$ whose cost is the sum of the cost of the partial rollout trajectory $(x_0, \tilde{u}_0, \tilde{x}_1, \tilde{u}_1, \ldots, \tilde{x}_k)$, and the Q-factor $\hat{Q}_k(\tilde{x}_k, \tilde{u}_k)$ of the base heuristic starting from $(\tilde{x}_k, \tilde{u}_k)$. The sequential improvement condition guarantees that

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

which implies that

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

If strict inequality holds, the rollout algorithm will switch from $T_k$ and follow $T_{k+1}$; cf. the traveling salesman Example 1.2.3.

Exercises 6.11, 6.14, 6.15, 6.16). The price for the performance improvement is extra computation that is typically equal to the computation time of the base heuristic times a factor that is a low order polynomial of $N$. It is generally hard to quantify the amount of performance improvement, but the computational results obtained from the case studies are consistent with the Newton step interpretations that we discussed in Chapter 1.

The books [Ber19a] (Section 2.5.1) and [Ber20a] (Section 3.1) show that the sequential improvement condition is satisfied in the context of MPC, and is the underlying reason for the stability properties of the MPC scheme. On the other hand the base heuristic underlying the classical form of the MPC scheme is not sequentially consistent (see the preceding references).

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

**Example 2.3.3 (Sequential Improvement Violation)**

Consider the 2-stage problem shown in Fig. 2.3.5, which involves two states
Figure 2.3.5 A 2-stage problem with states $x^*_1, \tilde{x}_1$ at stage 1, and states $x^*_2, \tilde{x}_2$ at stage 2. The controls and corresponding transitions are as shown in the figure. The rollout choice at the initial state $x_0$ is strictly suboptimal, while the base heuristic choice is optimal. The reason is that the base heuristic is not sequentially improving and makes the suboptimal choice $u^*_1$ at $x^*_1$, but makes the different (optimal) choice $u_0^*$ when run from $x_0$.

at each of stages 1 and 2, and the controls shown. Suppose that the unique optimal trajectory is $(x_0, u_0^*, x^*_1, u^*_1, x^*_2)$, and that the base heuristic produces this optimal trajectory starting at $x_0$. The rollout algorithm chooses a control at $x_0$ as follows: it runs the base heuristic to construct a trajectory starting from $x^*_1$ and $\tilde{x}_1$, with corresponding costs $H_1(x^*_1)$ and $H_1(\tilde{x}_1)$. If

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

(2.17)

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

Let us also verify that if the inequality (2.17) 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 and the monotonicity property (2.16) suggest a simple enhancement to the rollout algorithm, which detects when the sequential improvement condition is violated and takes corrective measures. In this algorithmic variant, called **fortified rollout**, we maintain the best trajectory obtained so far, and keep following that trajectory up to the point where we discover another trajectory that has improved cost.

### 2.3.2 The Fortified Rollout Algorithm

In this section we describe a rollout variant that implicitly enforces the sequential improvement property. This variant, called the **fortified rollout algorithm**, starts at $x_0$, and generates step-by-step a sequence of states \( \{x_0, x_1, \ldots, x_N\} \) and corresponding sequence of controls. Upon reaching state $x_k$ we have the trajectory

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

that has been constructed by rollout, called **permanent trajectory**, and we also store a **tentative best trajectory**

\[
\mathcal{T}_k = \{x_0, u_0, \ldots, u_{k-1}, x_k, \tilde{x}_{k+1}, \tilde{u}_{k+1}, \ldots, \tilde{x}_{N-1}, \tilde{x}_N\}
\]

with corresponding cost

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

The tentative best trajectory $\mathcal{T}_k$ 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 (see Fig. 2.3.6).

In particular, upon reaching state $x_k$, we run the rollout algorithm as earlier, i.e., for every $u_k \in U_k(x_k)$ and next state $x_{k+1} = f_k(x_k, u_k)$, we run the base heuristic from $x_{k+1}$, and find the control $\tilde{u}_k$ that gives the best trajectory, denoted

\[
\tilde{T}_k = \{x_0, u_0, \ldots, u_{k-1}, x_k, \tilde{u}_k, \tilde{x}_{k+1}, \tilde{u}_{k+1}, \ldots, \tilde{u}_{N-1}, \tilde{x}_N\}
\]

with corresponding cost

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

Whereas the ordinary rollout algorithm would choose control $\tilde{u}_k$ and move to $\tilde{x}_{k+1}$, the fortified algorithm compares $C(\mathcal{T}_k)$ and $C(\tilde{T}_k)$, and depending
Figure 2.3.6 Schematic illustration of fortified rollout. After $k$ steps, we have constructed the permanent trajectory

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

and the tentative best trajectory

$$\hat{\mathcal{T}}_k = \{x_0, u_0, \ldots, u_{k-1}, x_k, \hat{u}_k, \hat{x}_{k+1}, \hat{u}_{k+1}, \ldots, \hat{u}_{N-1}, \hat{x}_N\},$$

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

$$\tilde{\mathcal{T}}_k = \{x_0, u_0, \ldots, u_{k-1}, x_k, \tilde{u}_k, \tilde{x}_{k+1}, \tilde{u}_{k+1}, \ldots, \tilde{u}_{N-1}, \tilde{x}_N\}.$$ 

If the cost of the end-to-end trajectory $\tilde{\mathcal{T}}_k$ is lower than the cost of $\mathcal{T}_k$, we add $\{(\hat{u}_k, \hat{x}_{k+1})\}$ to the permanent trajectory and set the tentative best trajectory to $\hat{\mathcal{T}}_{k+1} = \hat{\mathcal{T}}_k$. Otherwise we add $\{(\tilde{u}_k, \tilde{x}_{k+1})\}$ to the permanent trajectory and keep the tentative best trajectory unchanged: $\hat{\mathcal{T}}_{k+1} = \hat{\mathcal{T}}_k$. On which of the two is smaller, chooses $\hat{u}_k$ or $\tilde{u}_k$ and moves to $\hat{x}_{k+1}$ or to $\tilde{x}_{k+1}$, respectively. In particular, if

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

the algorithm sets the next state and corresponding tentative best trajectory to

$$x_{k+1} = \tilde{x}_{k+1}, \quad \tilde{T}_{k+1} = \tilde{\mathcal{T}}_k,$$

and if

$$C(\mathcal{T}_k) > C(\hat{\mathcal{T}}_k),$$

it sets the next state and corresponding tentative best trajectory to

$$x_{k+1} = \hat{x}_{k+1}, \quad \hat{T}_{k+1} = \hat{\mathcal{T}}_k.$$
In other words the fortified rollout at $x_k$ follows the current tentative best trajectory $\overline{T}_k$ unless a lower cost trajectory $\tilde{T}_k$ is discovered by running the base heuristic from all possible next states $x_{k+1}$. It follows that at every state the tentative best trajectory has no larger cost than the initial tentative best trajectory, which is the one produced by the base heuristic starting from $x_0$. Moreover, it can be seen that if the base heuristic is sequentially improving, the rollout algorithm and its fortified version coincide. Experimental evidence suggests that it is often important to use the fortified version if the base heuristic is not known to be sequentially improving. Fortunately, the fortified version involves hardly any additional computational cost.

As expected, when the base heuristic generates an optimal trajectory, the fortified rollout algorithm will also generate the same trajectory. This is illustrated by the following example.

**Example 2.3.4**

Let us consider the application of the fortified rollout algorithm to the problem of Example 2.3.3 and see how it addresses the issue of cost improvement. The fortified rollout algorithm stores as initial tentative best trajectory the optimal trajectory $(x_0, u_0^*, x_1^*, u_1^*, x_2^*)$ generated by the base heuristic at $x_0$. Then, starting at $x_0$, it runs the heuristic from $x_1^*$ and $\tilde{x}_1$, and (despite the fact that the ordinary rollout algorithm prefers going to $\tilde{x}_1$ rather than $x_1^*$) it discards the control $\tilde{u}_0$ in favor of $u_0^*$, which is dictated by the tentative best trajectory. It then sets the tentative best trajectory to $(x_0, u_0^*, x_1^*, u_1^*, x_2^*)$.

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

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

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

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

into an (approximate) parallel optimization over the components $u_i^k$ (or subgroups of these components). However, as a result of approximate optimization over $u_k$, the cost improvement property may be degraded, even if

$\dagger$ The base heuristic may also be run from a subset of the possible next states $x_{k+1}$, as in the case where a simplified version of rollout is used; cf. Section 2.3.4. Then fortified rollout will still guarantee a cost improvement property.
Sec. 2.3 Rollout Algorithms for Discrete Optimization

2.3.3 Using Multiple Base Heuristics - Parallel Rollout

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

In particular, let us assume that we have $m$ heuristics, and that the $\ell$th of these, given a state $x_{k+1}$, produces a trajectory

$$\tilde{T}_{k+1}^\ell = \{x_{k+1}, \tilde{u}_{k+1}^\ell, x_{k+2}, \ldots, \tilde{u}_{N-1}^\ell, \tilde{x}_N^\ell\},$$

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

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

where

$$\hat{Q}_k^\ell(x_k, u_k) = g_k(x_k, u_k) + C(\tilde{T}_{k+1}^\ell)$$

is the cost of the trajectory $(x_k, u_k, \tilde{T}_{k+1}^\ell)$. Note that the Q-factors of the different heuristics can be computed independently and in parallel. In view of this fact, the rollout scheme just described is sometimes referred to as parallel rollout.

An interesting property, which can be readily verified by using the definitions, is that if all the heuristics are sequentially improving, the same is true for the superheuristic, something that is also suggested by Fig. 2.3.4. Indeed, let us write the sequential improvement condition (2.15) for each of the base heuristics

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

† A related practically interesting possibility is to introduce a partition of the state space into subsets, and a collection of multiple heuristics that are specially tailored to the subsets. We may then select the appropriate heuristic to use on each subset of the partition. In fact one may use a collection of multiple heuristics tailored to each subset of the state space partition, and at each state, select out of all the heuristics that apply, the one that yields minimum cost.
where \( \tilde{Q}_k^\ell(x_k, u_k) \) and \( H_k^\ell(x_k) \) are Q-factors and heuristic costs that correspond to the \( \ell \)th heuristic. Then by taking minimum over \( \ell \), we have

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

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

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

which is precisely the sequential improvement condition (2.15) for the superheuristic.

### 2.3.4 Simplified Rollout Algorithms

We will now consider a rollout variant, called simplified rollout, which is motivated by problems where the control constraint set \( U_k(x_k) \) is either infinite or finite but very large. Then the minimization

\[
\hat{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k), \quad (2.18)
\]

[cf. Eqs. (2.11) and (2.12)], may be unwieldy, since the number of Q-factors

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

is accordingly infinite or large.

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

\[
\overline{U}_k(x_k) \subset U_k(x_k).
\]

The rollout control \( \hat{\mu}_k(x_k) \) in this variant is one that attains the minimum of \( \tilde{Q}_k(x_k, u_k) \) over \( u_k \in \overline{U}_k(x_k) \):

\[
\hat{\mu}_k(x_k) \in \arg \min_{u_k \in \overline{U}_k(x_k)} \tilde{Q}_k(x_k, u_k). \quad (2.19)
\]

An example is when \( \overline{U}_k(x_k) \) results from discretization of an infinite set \( U_k(x_k) \). Another possibility is when by using some preliminary approximate optimization, we can identify a subset \( \overline{U}_k(x_k) \) of promising controls by using some heuristic method, and to save computation, we restrict attention to this subset. A related possibility is to generate \( \overline{U}_k(x_k) \) by some random search method that explores intelligently the set \( U_k(x_k) \) with the aim to minimize \( \tilde{Q}_k(x_k, u_k) \) [cf. Eq. (2.18)].
It turns out that the proof of the cost improvement property of Prop. 2.3.2,
\[ J_{k, \hat{\pi}}(x_k) \leq H_k(x_k), \quad \text{for all } x_k \text{ and } k, \]
goes through if the following modified sequential improvement property holds:
\[
\min_{u_k \in U_k(x_k)} \hat{Q}_k(x_k, u_k) \leq H_k(x_k). \tag{2.20}
\]
This can be seen by verifying that Eq. (2.20) is sufficient to guarantee that the monotone improvement Eq. (2.16) is satisfied. The condition (2.20) is very simple to satisfy if the base heuristic is sequentially consistent, in which case the control \( \overline{u}_k \) selected by the base heuristic satisfies
\[ \hat{Q}_k(x_k, \overline{u}_k) = H_k(x_k). \]
In particular, for the property (2.20) to hold, it is sufficient that \( U_k(x_k) \) contains the base heuristic choice \( \overline{u}_k \).

The idea of replacing the minimization (2.18) by the simpler minimization (2.19) can be extended. In particular, by working through the preceding argument, it can be seen that any policy \( \hat{\pi} = \{\hat{\mu}_0, \ldots, \hat{\mu}_{N-1}\} \) such that \( \hat{\mu}_k(x_k) \) satisfies the condition
\[ \hat{Q}_k(x_k, \hat{\mu}_k(x_k)) \leq H_k(x_k), \]
for all \( x_k \) and \( k \), guarantees the modified sequential improvement property (2.20), and hence also the cost improvement property. A prominent example of such an algorithm arises in the multiagent case where \( u \) has \( m \) components, \( u = (u^1, \ldots, u^m) \), and the minimization over \( U_k^1(x_k) \times \cdots \times U_k^m(x_k) \) is replaced by a sequence of single component minimizations, one-component-at-a-time; cf. Section 1.6.5.

### 2.3.5 Truncated Rollout with Terminal Cost Approximation

An important variation of rollout algorithms is truncated rollout with terminal cost approximation. Here the rollout trajectories are obtained by running the base policy from the leaf nodes of the lookahead tree, but they are truncated after a given number of steps, while a terminal cost approximation is added to the heuristic cost to compensate for the resulting error. This is important for problems with a large number of stages, and it is also essential for infinite horizon problems where the rollout trajectories have infinite length.

One possibility that works well for many problems is to simply set the terminal cost approximation to zero. Alternatively, the terminal cost
function approximation may be obtained by using some sophisticated off-line training process that may involve an approximation architecture such as a neural network or by using some heuristic calculation based on a simplified version of the problem. This form of truncated rollout may also be viewed as an intermediate approach between standard rollout where there is no truncation and cost function approximation, and approximation in value space without any rollout.

2.3.6 Model-Free Rollout

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

We consider the general discrete optimization problem of selecting a control sequence $u = (u_0, \ldots, u_{N-1})$ to minimize a function $G(u)$. For simplicity we assume that each component $u_k$ is constrained to lie in a given constraint set $U_k$, but extensions to more general constraint sets are possible. We assume the following:

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

$$S_k(u_0, \ldots, u_k, \tilde{u}_{k+1}) = (u_0, \ldots, u_k, \tilde{u}_{k+1}, \ldots, \tilde{u}_{N-1}).$$

The base heuristic is also used to start the algorithm from an artificial empty solution, by generating all components $\tilde{u}_0 \in U_0$ and a complete feasible solution $(\tilde{u}_0, \ldots, \tilde{u}_{N-1})$, starting from each $\tilde{u}_0 \in U_0$.

(b) An “expert” is available that can compare any two feasible solutions $u$ and $\overline{u}$, in the sense that he/she can determine whether

$$G(u) > G(\overline{u}), \text{ or } G(u) \leq G(\overline{u}).$$

It can be seen that deterministic rollout can be applied to this problem, even though the cost function $G$ is unknown. The reason is that the rollout algorithm uses the cost function only as a means of ranking complete solutions in terms of their cost. Hence, if the ranking of any two solutions can be revealed by the expert, this is all that is needed.† In fact,

† Note that for this to be true, it is important that the problem is deterministic, and that the expert ranks solutions using some underlying (though unknown) cost function. In particular, the expert’s rankings should have a transitivity property: if $u$ is ranked better than $u'$ and $u'$ is ranked better than $u''$, then $u$ is ranked better than $u''$. 
the constraint sets \(U_0, \ldots, U_{N-1}\) need not be known either, as long as they can be generated by the base heuristic. Thus, the rollout algorithm can be described as follows (see Fig. 2.3.7):

We start with an artificial empty solution, and at the typical step, given the partial solution \((u_0, \ldots, u_k), k < N - 1\), we use the base heuristic to generate all possible one-step-extended solutions

\[
(u_0, \ldots, u_k, \tilde{u}_{k+1}), \quad \tilde{u}_{k+1} \in U_{k+1},
\]

and the set of complete solutions

\[
S_k(u_0, \ldots, u_k, \tilde{u}_{k+1}), \quad \tilde{u}_{k+1} \in U_{k+1}.
\]

We then use the expert to rank this set of complete solutions. Finally, we select the component \(u_{k+1}\) that is ranked best by the expert, extend the partial solution \((u_0, \ldots, u_k)\) by adding \(u_{k+1}\), and repeat with the new partial solution \((u_1, \ldots, u_k, u_{k+1})\).

Except for the (mathematically inconsequential) use of an expert rather than a cost function, the preceding rollout algorithm can be viewed as a special case of the one given earlier. As a result several of the rollout variants that we have discussed so far (rollout with multiple heuristics, simplified rollout, and fortified rollout) can also be easily adapted.

**Example 2.3.5 (RNA Folding)**

In a classical problem from computational biology, we are given a sequence of nucleotides, represented by circles in Fig. 2.3.8, and we want to “fold” the sequence in an “interesting” way (introduce pairings of nucleotides that result in an “interesting” structure). There are some constraints on which pairings are possible, but we will not go into the details of this (some types of constraints may require the use of the constrained rollout framework of Section 2.5). A common constraint is that the pairings should not “cross,” i.e., given a pairing \((i_1, i_2)\) there should be no pairing \((i_3, i_4)\) where either \(i_3 < i_1\) and \(i_1 < i_4 < i_2\), or \(i_1 < i_3 < i_2\) and \(i_2 < i_4\). This type of problem has a long history of solution by DP, starting with the paper by Zuker and Stiegler [ZuS81]. There are several formulations, where the aim is to optimize some criterion, e.g., the number of pairings, or the “energy” of the folding. However, biologists do not agree on a suitable criterion, and have developed software to generate “reasonable” foldings, based on semi-heuristic reasoning.

We will develop a rollout approach that makes use of such software without discussing their underlying principles.

We formulate the folding problem as a discrete optimization problem involving a pairing decision at each nucleotide in the sequence with at most three choices (open a pairing, close a pairing, do nothing); see Fig. 2.3.8. To apply rollout, we need a base heuristic, which given a partial folding, generates a complete folding (this is the partial folding software shown in Fig. 2.3.8). Two complete foldings can be compared by some other software, called
We assume that we do not know $G$ and/or $U_0, \ldots, U_{N-1}$. Instead we have a base heuristic, which given a partial solution $(u_0, \ldots, u_k)$, outputs all next controls $\tilde{u}_{k+1} \in U_{k+1}$, and generates from each a complete solution 

$$S_k(u_0, \ldots, u_k, \tilde{u}_{k+1}) = (u_0, \ldots, u_k, \tilde{u}_{k+1}, \ldots, \tilde{u}_{N-1}).$$

Also, we have a human or software “expert” that can rank any two complete solutions without assigning numerical values to them. The control that is selected from $U_{k+1}$ by the rollout algorithm is the one whose corresponding complete solution is ranked best by the expert.

Learning to Imitate the Expert

To implement model-free rollout, we need both a base heuristic and an
expert. None of these may be readily available, particularly the expert, which involves a hidden cost function that is implicitly used to rank complete solutions. Within this context, it is worth considering the case where an expert is not available but can be emulated by training with the use of data. In particular, suppose that we are given a set of control sequence pairs \((u^s, \overline{u}^s), \) \(s = 1, \ldots, q,\) with
\[
G(u^s) > G(\overline{u}^s), \quad s = 1, \ldots, q, 
\]  
which we can use for training. Such a set may be obtained in a variety of ways, including querying the expert. We may then train a parametric approximation architecture such as a neural network to produce a function \(\tilde{G}(u, r),\) where \(r\) is a parameter vector, and use this function in place of the unknown \(G(u)\) to implement the preceding rollout algorithm.

A method, known as comparison training, has been suggested for this purpose, and has been used in a variety of game contexts, including backgammon and chess by Tesauro \cite{Tesauro1989a}, \cite{Tesauro2001}. 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 \(\tilde{G}(\cdot, r),\) involving a parameter vector \(r,\) such as a neural network, is then trained by some form of regression with these data to produce an approximation \(\tilde{G}(\cdot, \bar{r})\) to be used in place of \(G(\cdot)\) in a rollout scheme. We refer to Chapter 3 and to the aforementioned papers by Tesauro for implementation details of the regression procedure. See also Section 3.4 on parametric approximation in policy space through the use of classification methods.

**Figure 2.3.8** Schematic illustration of rollout for the RNA folding problem. The current state is the partial folding depicted on the left side. There are at most three choices for control at each state.
Learning the Base Policy’s Q-Factors

In another type of imitation approach, we view the base policy decisions as being selected by a process the mechanics of which are not observed except through its generated cost samples at the various stages. In particular, the stage costs starting from any given partial solution \( (u_0, \ldots, u_k) \) are added to form samples of the base policy’s Q-factors \( Q_k(u_0, \ldots, u_k) \). In this way we can obtain Q-factor samples starting from many partial solutions \( (u_0, \ldots, u_k) \). Moreover, a single complete solution \( (u_0, \ldots, u_N-1) \) generated by the base policy provides multiple Q-factor samples, one for each of the partial solutions \( (u_0, \ldots, u_k) \).

We can then use the sample (partial solution, cost) pairs in conjunction with a training method (see Chapter 3) in order to construct parametric approximations

\[
\hat{Q}_k(u_0, \ldots, u_k, r_k), \quad k = 1, \ldots, N,
\]

to the true Q-factors \( Q_k(u_0, \ldots, u_k) \), where \( r_k \) is the parameter vector. Once the training has been completed and the Q-factors \( \hat{Q}_k(u_0, \ldots, u_k, r_k) \) have been obtained for all \( k \), we can construct complete solutions step-by-step, by selecting the next component \( \hat{u}_{k+1} \), given the partial solution \( (u_0, \ldots, u_k) \), through the minimization

\[
\hat{u}_{k+1} \in \arg \min_{u_{k+1} \in U_{k+1}} \hat{Q}_{k+1}(\tilde{u}_0, \ldots, \tilde{u}_k, u_{k+1}, r_{k+1}).
\]

Note that even though we are “learning” the base policy, our aim is not to imitate it, but rather to generate a rollout policy. The latter policy will make better decisions than the base policy, thanks to the cost improvement property of rollout. This points to an important issue of exploration: we must ensure that the training set of sample (partial solution, cost) pairs is broadly representative, in the sense that it is not unduly biased towards sample pairs that are generated by the base policy.

2.4 ROLLOUT AND APPROXIMATION IN VALUE SPACE WITH MULTISTEP LOOKAHEAD

We will now consider approximation in value space with multistep lookahead minimization, possibly also involving some form of rollout. Figure 2.4.1 describes the case of pure (nontruncated) form of rollout with two-step lookahead for deterministic problems. In particular, suppose that after \( k \) steps we have reached state \( x_k \). We then consider the set of all possible two-step-ahead states \( x_{k+2} \), we run the base heuristic starting from each of them, and compute the two-stage cost to get from \( x_k \) to \( x_{k+2} \), plus the cost of the base heuristic from \( x_{k+2} \). We select the state, say \( \hat{x}_{k+2} \), that
is associated with minimum cost, compute the controls $\tilde{u}_k$ and $\tilde{u}_{k+1}$ that lead from $x_k$ to $\tilde{x}_{k+2}$, choose $\tilde{u}_k$ as the next control and $x_{k+1} = f_k(x_k, \tilde{u}_k)$ as the next state, and discard $\tilde{u}_{k+1}$.

The extension of the algorithm to lookahead of more than two steps is straightforward: instead of the two-step-ahead states $x_{k+2}$, we run the base heuristic starting from all the possible $\ell$-step ahead states $x_{k+\ell}$, etc. For cases where the $\ell$-step lookahead minimization is very time consuming, we may consider variants involving approximations aimed at simplifying the associated computations.

An important variation is truncated rollout with terminal cost approximation. Here the rollout trajectories are obtained by running the base heuristic from the leaf nodes of the lookahead graph, and they are truncated after a given number of steps, while a terminal cost approximation is added to the heuristic cost to compensate for the resulting error; see Fig. 2.4.2. One possibility that works well for many problems, particularly when the combined lookahead for minimization and base heuristic simulation is long, is to simply set the terminal cost approximation to zero. Alternatively, the terminal cost function approximation can be obtained by problem approximation or by using some sophisticated off-line training process that may involve an approximation architecture such as a neural
network. Generally, the terminal cost approximation is especially important if a large portion of the total cost is incurred upon termination (this is true for example in games).

Note that the preceding algorithmic scheme can be viewed as multi-step approximation in value space, and it can be interpreted as a Newton step, with suitable starting point that is determined by the truncated rollout with the base heuristic, and the terminal cost approximation. This interpretation is possible once the discrete optimal control problem is reformulated to an equivalent infinite horizon SSP problem; cf. the discussion of Sections 1.6.2 and 2.1. Thus the algorithm inherits the fast convergence property of the Newton step, which we have discussed in the context of infinite horizon problems in Section 1.5; see also the book [Ber22a].

The architecture of Fig. 2.4.2 contains as a special case the general multistep approximation in value space scheme, where there is no rollout at all; i.e., the leaves of the multistep lookahead tree are evaluated with the function $\tilde{J}$. Figure 2.4.3 illustrates this special case, where for notational simplicity we have denoted the current state by $x_0$. The illustration involves an acyclic graph with a single root (the current state) and $\ell$ layers, with the $n$th layer consisting of the states $x_n$ that are reachable from $x_0$ with a feasible sequence of $n$ controls. In particular, there is an arc for every state $x_1$ of the 1st layer that can be reached from $x_0$ with a feasible control, and similarly an arc for every pair of states $(x_n, x_{n+1})$, of layers $n$ and $n + 1$. 

![Figure 2.4.2](image)

**Figure 2.4.2** Illustration of truncated rollout with two-step lookahead and a terminal cost approximation $\tilde{J}$. The base heuristic is used for a limited number of steps and the terminal cost is added to compensate for the remaining steps.
Sec. 2.4 Rollout and Approximation in Value Space with Multistep Lookahead

In $\ell$-step approximation in value space, we obtain a trajectory

$$\{x_0, x_1^*, \ldots, x_\ell^*\}$$

that minimizes the shortest distance from $x_0$ to $x_\ell$ plus $\hat{J}(x_\ell)$. We then use the control that corresponds to the first move $x_0 \rightarrow x_1^*$.

respectively, for which $x_{n+1}$ can be reached from $x_n$ with a feasible control. The cost of each of these arcs is the stage cost of the corresponding state-control pair, minimized over all possible controls that correspond to the same pair $(x_n, x_{n+1})$. Mathematically, the cost of the arc $(x_n, x_{n+1})$ is

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

For the states $x_\ell$ of the last layer there is also a given terminal cost approximation $\hat{J}(x_\ell)$, possibly obtained through off-line training and/or rollout with a base policy. It can be thought of as the cost of an artificial arc connecting $x_\ell$ to an artificial termination state.

Once we have computed all the shortest distances $D(x_\ell)$ from $x_0$ to all states $x_\ell$ of the last layer $\ell$, we obtain the $\ell$-step lookahead control to be applied at the current state $x_0$, by minimizing over $x_\ell$ the sum

$$D(x_\ell) + \hat{J}(x_\ell).$$
If \( x^* \) is the state that attains the minimum, we generate the corresponding trajectory \((x_0, x^*_1, \ldots, x^*_\ell)\), and then use the control that corresponds to the first move \( x_0 \rightarrow x^*_1 \); see Fig. 2.4.3. Note that the shortest path problems from \( x_0 \) to all states \( x_n \) of all the layers \( n = 1, \ldots, \ell \) can be solved simultaneously by backward DP (start from layer \( \ell \) and go back towards \( x_0 \)).

**Long Lookahead for Deterministic Problems**

The architecture of Figs. 2.4.2 and 2.4.3 is similar to the one we discussed in Section 1.1 for AlphaZero and related programs. However, because it is adapted to deterministic problems, it is much simpler to implement and to use. In particular, the truncated rollout portion does not involve expensive Monte Carlo simulation, while the multistep lookahead minimization portion involves a deterministic shortest path problem, which is much easier to solve than its stochastic counterpart. These favorable characteristics can be exploited to facilitate implementations that involve very long lookahead.

Generally speaking, longer lookahead is desirable because it typically results in improved performance. We will adopt this as a working hypothesis. It is typically true in practice, although it cannot be established analytically in the absence of additional assumptions.† On the other hand, the on-line computational cost of multistep lookahead increases, often exponentially, with the length of lookahead. We conclude that we should aim to use a lookahead that is as long as is allowed by the on-line computational budget (the amount of time that is available for calculating a control to apply at the current state).

**Long Lookahead by Using Truncated Rollout**

Our preceding discussion leads to the question of how to economize in computation in order to effectively increase the length of the multistep lookahead within a given on-line computational budget. One way to do this, which we have already discussed, is the use of truncated rollout that explores forward through a deterministic base policy at far less computational cost than lookahead minimization of equal length. As an example, let us consider the possibility of starting with a terminal cost function \( \tilde{J} \), possibly generated by off-line training, and use as base policy for rollout

† Indeed, there are examples where as the size \( \ell \) of the lookahead becomes longer, the performance of the multistep lookahead policy deteriorates (see [Ber17a], Section 6.1.2, or [Ber19a], Section 2.2.1). However, these examples are isolated and artificial. They are not representative of practical experience.
Sec. 2.4 Rollout and Approximation in Value Space with Multistep Lookahead

the one-step lookahead policy \( \tilde{\mu} \), defined by \( \tilde{J} \) using the equation\(^\dagger\)

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

Let us assume that the principal computation in the minimization of Eq. (2.23) is the calculation of \( \tilde{J}(f(x, u)) \), and compare two possibilities:

(a) Using \( \ell \)-step lookahead minimization with \( \tilde{J} \) as the terminal cost approximation without any rollout; cf. Fig. 2.4.3.

(b) Using one-step lookahead minimization, with \( (\ell - 1) \)-step truncated rollout and \( \tilde{J} \) as the terminal cost approximation.

Note that scheme (b) is the one used by the TD-Gammon program of Tesauto and Galperin [TeG96], out of necessity, because multistep lookahead is very expensive in backgammon, due to the rapid growth of the lookahead graph as \( \ell \) increases (cf. the discussion of Section 1.1).

Suppose that the control set \( U(x) \) has \( m \) elements for every \( x \). Then the \( \ell \)-step lookahead minimization scheme (a) requires the calculation of as many as \( m^\ell \) values of \( \tilde{J} \), because the number of leaves of the \( m \)-step lookahead graph are as many as \( m^\ell \). Let us now calculate the corresponding number of calculations of the value of \( \tilde{J} \) for scheme (b).

The first lookahead stage starting from the current state \( x_k \) requires \( m \) calculations corresponding to the \( m \) controls in \( U(x_k) \), and yields corresponding states \( x_{k+1} \), which are as many as \( m \). For each of these states \( x_{k+1} \), we must calculate a sequence of \( \ell - 1 \) controls using the base policy (2.23) for stages \( (k + 1) \) to \( (k + \ell) \). Each of these \( \ell - 1 \) controls requires \( m \) calculations of the value of \( \tilde{J} \). Thus, for the \( \ell - 1 \) stages of truncated rollout, there are \( m \cdot (\ell - 1) \) calculations of the value of \( \tilde{J} \) per state \( x_{k+1} \), for a total of as many as \( m^2 \cdot (\ell - 1) \) calculations. Adding the \( m \) calculations at state \( x_k \), we conclude that scheme (b) requires a total of as many as \( m^2 \cdot \ell \) calculations of the value of \( \tilde{J} \).

In conclusion, both schemes (a) and (b) above look forward for \( \ell \) stages, but their associated total computation grows exponentially and linearly with \( \ell \), respectively. Thus, for a given computational budget, short lookahead minimization with long truncated rollout, can increase the total amount of lookahead and improve the performance of approximation in value space schemes. This is particularly so since based on the Newton step interpretations of approximation in value space of Section 1.5, truncated rollout with a reasonably good (e.g., stable) base policy often works about as well as long lookahead minimization. Extensive computational practice, starting with the rollout/TD-Gammon scheme of [TeG96], is consistent with this assessment.

\(^\dagger\) For simplicity, we use stationary system notation, omitting the time subscripts of \( U \), \( g \), and \( f \).
In the following two sections, we will explore two alternative ways to speed up the lookahead minimization calculation, thereby allowing a larger number $\ell$ of computational stages for a given on-line computational budget. These are based on iterative deepening of the shortest path computation, and pruning of the lookahead minimization graph.

### 2.4.1 Iterative Deepening Using Forward Dynamic Programming

As noted earlier, the shortest path problems from $x_0$ to $x_\ell$ in Fig. 2.4.3 can be solved simultaneously by the familiar backward DP that starts from layer $\ell$ and goes towards $x_0$. An important alternative for solving these problems is the forward DP algorithm. This is the same as the backwards DP algorithm with the direction of the arcs reversed (start from $x_0$ and go towards layer $\ell$). In particular, the shortest distances $D_{n+1}(x_{n+1})$ to layer $n + 1$ states are obtained from the shortest distances $D_n(x_n)$ to layer $n$ states through the equation

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

In the following two sections, we will explore two alternative ways to speed up the lookahead minimization calculation, thereby allowing a larger number $\ell$ of computational stages for a given on-line computational budget. These are based on iterative deepening of the shortest path computation, and pruning of the lookahead minimization graph.

### 2.4.1 Iterative Deepening Using Forward Dynamic Programming

As noted earlier, the shortest path problems from $x_0$ to $x_\ell$ in Fig. 2.4.3 can be solved simultaneously by the familiar backward DP that starts from layer $\ell$ and goes towards $x_0$. An important alternative for solving these problems is the forward DP algorithm. This is the same as the backwards DP algorithm with the direction of the arcs reversed (start from $x_0$ and go towards layer $\ell$). In particular, the shortest distances $D_{n+1}(x_{n+1})$ to layer $n + 1$ states are obtained from the shortest distances $D_n(x_n)$ to layer $n$ states through the equation

$$D_{n+1}(x_{n+1}) = \min_{x_n} \left[ \hat{g}_n(x_n, x_{n+1}) + D_n(x_n) \right].$$
which is also illustrated in Fig. 2.4.4. Here \( \hat{g}_n(x_n, x_{n+1}) \) is the cost (or length) of the arc \((x_n, x_{n+1})\); cf. Eq. (2.22).

In particular, the solution of the \( \ell \)-step lookahead problem is obtained from the shortest path to the state \( x^*_\ell \) of layer \( \ell \) that minimizes \( D_\ell(x_\ell) + \hat{J}(x_\ell) \). The idea of iterative deepening is to progressively solve the \( n \)-step lookahead problem first for \( n = 1 \), then for \( n = 2 \), and so on, until our on-line computational budget is exhausted. In addition to fitting perfectly the mechanism of the forward DP algorithm, this scheme has the character of an “anytime” algorithm; i.e., it returns a feasible solution to a lookahead minimization of some depth, even if it is interrupted because the limit of our computational budget has been reached. In practice this is an important advantage, well known from chess programming, which allows us to keep on aiming for longer lookahead minimization, within the limit imposed by our computational budget constraint.

Iterative Deepening Combined with Pruning

A principal difficulty in approximation in value space with \( \ell \)-step lookahead stems from the rapid expansion of the lookahead graph as \( \ell \) increases. One way to mitigate this difficulty is to “prune” the lookahead minimization graph, i.e., to delete some of its arcs in order to expedite the shortest path computations from the current state to the states of subsequent layers; see Fig. 2.4.5. One possibility is to combine pruning with iterative deepening by eliminating from the computation states \( \hat{x}_n \) of layer \( n \) such that the

**Figure 2.4.5** Illustration of iterative deepening with pruning within the context of forward DP.
Figure 2.4.6 Illustration of the $\ell$-step lookahead minimization problem and its suboptimal solution with the IMR algorithm. The algorithm maintains a connected acyclic subgraph $S$ as shown. At each iteration it expands $S$ by selecting a leaf node of $S$ and by adding its neighbor nodes to $S$ (if not already in $S$). The leaf node, denoted $x^*$, is the one that minimizes over all leaf nodes $x$ of $S$ the sum of the shortest distance $D(x)$ from $x_0$ to $x$ and a “heuristic cost” $H(x)$.

$n$-step lookahead cost

$$D_n(\hat{x}_n) + \tilde{J}(\hat{x}_n)$$

is “far from the minimum” over $x_n$. This in turn prunes automatically some of the states of the next layer $n+1$. The rationale is that such states are “unlikely” to be part of the shortest path that we aim to compute. Note that this type of pruning is progressive, i.e., we prune states in layer $n$ before pruning states in layer $n+1$.

2.4.2 Incremental Multistep Rollout

We will now consider a more flexible form of the rollout scheme, which we call incremental multistep rollout (IMR). It applies a base heuristic and a forward DP computation to a sequence of subgraphs of a multistep lookahead graph, with the size of the subgraphs expanding iteratively. In particular, in incremental rollout a connected subgraph of multiple paths is iteratively extended starting from the current state going towards the end of the lookahead horizon, instead of extending a single path as in rollout. This is similar to what is done in Monte Carlo Tree Search (MCTS, to be discussed later), which is also designed to solve approximately general multistep lookahead minimization problems (including stochastic ones), and involves iterative expansion of an acyclic lookahead graph to new nodes, as well as backtracking to previously encountered nodes. However, incremental rollout seems to be more appropriate than MCTS for deterministic problems, where there are no random variables in the problem’s model and therefore Monte Carlo simulation does not make sense.
Sec. 2.5 Constrained Forms of Rollout Algorithms

The IMR algorithm starts with and maintains a connected acyclic subgraph \( S \) of the given multistep lookahead graph \( G \), which contains \( x_0 \). At each iteration it expands \( S \) by selecting a leaf node of \( S \) and by adding its neighbor nodes to \( S \) (if not already in \( S \)); see Fig. 2.4.6. The leaf node, denoted \( x^* \), is the one that minimizes (over all leaf nodes \( x \) of \( S \)) the sum

\[
D(x) + H(x),
\]

where

\( D(x) \) is the shortest distance from \( x_0 \) to the leaf node \( x \) using only arcs that belong to \( S \). This can be computed by forward DP.

\( H(x) \) is a “heuristic cost” corresponding to \( x \). This is defined as the sum of three terms:

(a) The cost of the base heuristic starting from node \( x \) and ending at one of the states \( x_\ell \) in the last layer \( \ell \).

(b) The terminal cost approximation \( \tilde{J}(x_\ell) \), where \( x_\ell \) is the state obtained via the base heuristic as in (a) above.

(c) An additional penalty \( P(x) \) that depends on the layer to which \( x \) belongs. As an example, we will assume here that

\[
P(x) = \delta \cdot (\text{the layer index of } x),
\]

where \( \delta \) is a positive parameter. Thus \( P(x) \) adds a cost of \( \delta \) for each extra arc to reach \( x \) from \( x_0 \), and penalizes nodes \( x \) that lie in more distant layers from the root \( x_0 \). It thus encourages the algorithm to “backtrack” and select nodes \( x^* \) that lie in layers closer to \( x_0 \).

The role of the parameter \( \delta \) is noteworthy and affects significantly the nature of the algorithm. When \( \delta = 0 \), the initial graph \( S \) consists of the single state \( x_0 \), and the base heuristic is sequentially improving, it can be seen that IMR performs exactly like the rollout algorithm for solving the \( \ell \)-step lookahead minimization problem. On the other hand when \( \delta \) is large enough, the algorithm operates like the forward DP algorithm. The reason is that a very large value of \( \delta \) forces the algorithm to expand all nodes of a given layer before proceeding to the next layer.

Generally, as \( \delta \) increases, the algorithm tends to backtrack more often, and to generate more paths through the graph, thereby visiting more nodes and increasing the number of applications of the base heuristic. Thus \( \delta \) may be viewed as an exploration parameter; when \( \delta \) is large the algorithm tends to explore more paths thereby improving the quality of the multistep lookahead minimization, at the expense of greater computational effort. In the absence of additional problem-specific information, favorable values of \( \delta \) should be obtained through experimentation. One may also consider alternative and more adaptive schemes; for example with a \( \delta \) that depends on \( x_0 \), and is adjusted in the course of the computation.
2.5 CONSTRAINED FORMS OF ROLLOUT ALGORITHMS

In this section we will discuss constrained deterministic DP problems, including challenging combinatorial optimization and integer programming problems. We introduce a rollout algorithm, which relies on a base heuristic and applies to problems with general trajectory constraints. Under suitable assumptions, we will show that if the base heuristic produces a feasible solution, the rollout algorithm has a cost improvement property: it produces a feasible solution, whose cost is no worse than the base heuristic’s cost.

Before going into formal descriptions of the constrained DP problem formulation and the corresponding algorithms, it is worth to revisit the broad outline of the rollout algorithm for deterministic DP:

(a) It constructs a sequence \{T_0, T_1, \ldots, T_N\} of complete system trajectories with monotonically nonincreasing cost (assuming a sequential improvement condition).

(b) The initial trajectory \(T_0\) is the one generated by the base heuristic starting from \(x_0\), and the final trajectory \(T_N\) is the one generated by the rollout algorithm.

(c) For each \(k\), the trajectories \(T_k, T_{k+1}, \ldots, T_N\) share the same initial portion \((x_0, \hat{u}_0, \ldots, \hat{u}_{k-1}, \hat{x}_k)\).

(d) For each \(k\), the base heuristic is used to generate a number of candidate trajectories, all of which share the initial portion with \(T_k\), up to state \(\hat{x}_k\). These candidate trajectories correspond to the controls \(u_k \in U_k(x_k)\). (In the case of fortified rollout, these trajectories include the current “tentative best” trajectory.)

(e) For each \(k\), the next trajectory \(T_{k+1}\) is the candidate trajectory that is best in terms of total cost.

In our constrained DP formulation, to be described shortly, we introduce a trajectory constraint \(T \in C\), where \(C\) is some subset of admissible trajectories. A consequence of this is that some of the candidate trajectories in (d) above, may be infeasible. Our modification to deal with this situation is simple: we discard all the candidate trajectories that violate the constraint, and we choose \(T_{k+1}\) to be the best of the remaining candidate trajectories, the ones that are feasible.

Of course, for this modification to be viable, we have to guarantee that at least one of the candidate trajectories will satisfy the constraint for every \(k\). For this we will rely on a sequential improvement condition that we will introduce shortly. For the case where this condition does not hold, we will introduce a fortified version of the algorithm, which requires only that the base heuristic generates a feasible trajectory \(T_0\) starting from the initial condition \(x_0\). Thus to apply reliably the constrained rollout algorithm, we only need to know a single feasible solution, i.e., a trajectory \(T_0\) that starts...
at \( x_0 \) and satisfies the constraint \( T_0 \in C \).

**Constrained Problem Formulation**

We assume that the state \( x_k \) takes values in some (possibly infinite) set and the control \( u_k \) takes values in some finite set. The finiteness of the control space is only needed for implementation purposes of the rollout algorithms to be described shortly. The algorithm can be defined without the finiteness condition, and makes sense, provided the implementation issues associated with infinite control spaces can be dealt with. A sequence of the form

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

(2.24)

where

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

(2.25)

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

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

(2.26)

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

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

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

(2.27)

which we have assumed so far. Thus, except for the finiteness of the control space, which is needed for implementation of rollout, this is a very general optimization problem. In fact, later we will simplify the problem further by eliminating the state transition structure of Eq. (2.25).†

Trajectory constraints can arise in a number of ways. A relatively simple example is the standard problem formulation for deterministic DP: an additive cost of the form (2.27), where the controls satisfy the time-uncoupled constraints \( u_k \in U_k(x_k) \) [so here \( C \) is the set of trajectories that are generated by the system equation with controls satisfying \( u_k \in U_k(x_k) \)]. In a more complicated constrained DP problem, there may be constraints that couple the controls of different stages such as

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

(2.28)

† Actually, similar to our discussion on model-free rollout in Section 2.3.6, it is not essential that we know the explicit form of the cost function \( G \) and the constraint set \( C \). For our constrained rollout algorithms, it is sufficient to have access to a human or software expert that can determine whether a given trajectory \( T \) is feasible, i.e., satisfies the constraint \( T \in C \), and also to be able to compare any two feasible trajectories \( T_1 \) and \( T_2 \), based on some internal process that is unknown to us, without assigning numerical values to them.
where \( g^m_k \) and \( b^m \) are given functions and scalars, respectively. Examples of this type include multiobjective or Pareto optimization problems, where there are multiple cost functions of interest, and all but one of the cost functions are treated through constraints (see e.g., [Ber17a], Ch. 2). Examples where difficult trajectory constraints arise also include situations where the control contains some discrete components, which once chosen must remain fixed for multiple time periods.

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

**Example 2.5.1 (A Constrained Form of the Traveling Salesman Problem)**

Let us consider a constrained version of the traveling salesman problem of Example 1.2.2. We want to find a minimum travel cost tour that additionally satisfies a safety constraint that the “safety cost” of the tour should be less than a certain threshold; see Fig. 2.5.1. This constraint need not have the
Sec. 2.5 Constrained Forms of Rollout Algorithms

additive structure of Eq. (2.28). We are simply given a safety cost for each tour (see the table at the bottom right), which is calculated in a way that is of no further concern to us. In this example, for a tour to be admissible, its safety cost must be less than or equal to 10. Note that the (unconstrained) minimum cost tour, ABDCA, does not satisfy the safety constraint.

Using a Base Heuristic for Constrained Rollout

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

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

can produce a (complementary) partial trajectory

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

that starts at \( x_k \) and satisfies the system equation

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

Thus, given \( y_k \) and any control \( u_k \), we can use the base heuristic to obtain a complete trajectory as follows:

(a) Generate the next state \( x_{k+1} = f_k(x_k, u_k) \).
(b) Extend \( y_k \) to obtain the partial trajectory

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

(c) Run the base heuristic from \( y_{k+1} \) to obtain the partial trajectory \( R(y_{k+1}) \).
(d) Join the two partial trajectories \( y_{k+1} \) and \( R(y_{k+1}) \) to obtain the complete trajectory \( (y_k, u_k, R(y_{k+1})) \), which is denoted by \( T_k(y_k, u_k) \):

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

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

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

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

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

\[ \tilde{y}_{k+1} = (\tilde{y}_k, \hat{u}_k, f_k(\tilde{x}_k, \hat{u}_k)), \quad k = 0, \ldots, N - 1, \tag{2.31} \]
Approximation in Value Space - Rollout Algorithms Chap. 2

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

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

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

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

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

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

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

(2.32)

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

\[
G(\tilde{y}_N) \leq G(R(\tilde{y}_0)).
\]

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


Constrained Rollout Algorithm
The algorithm starts at stage 0 and sequentially proceeds to the last stage. At the typical stage \( k \), it has constructed a partial trajectory
\[
\tilde{y}_k = (\tilde{x}_0, \tilde{u}_0, \tilde{x}_1, \ldots, \tilde{u}_{k-1}, \tilde{x}_k)
\] (2.33)
that starts at the given initial state \( \tilde{y}_0 = \tilde{x}_0 \), and is such that
\[
\tilde{x}_{t+1} = f_t(\tilde{x}_t, \tilde{u}_t), \quad t = 0, 1, \ldots, k-1.
\]
The algorithm then forms the set of controls
\[
U_k(\tilde{y}_k) = \{ u_k \mid T_k(\tilde{y}_k, u_k) \in C \}
\]
that is consistent with feasibility [cf. Eq. (2.30)], and chooses a control \( \tilde{u}_k \in U_k(\tilde{y}_k) \) according to the minimization
\[
\tilde{u}_k \in \arg \min_{u_k \in U_k(\tilde{y}_k)} G(T_k(\tilde{y}_k, u_k)),
\] (2.34)
[cf. Eq. (2.32)], where
\[
T_k(\tilde{y}_k, u_k) = \left( \tilde{y}_k, u_k, R(\tilde{y}_k, u_k, f_k(\tilde{x}_k, u_k)) \right);
\] [cf. Eq. (2.29)]. 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. (2.31)], thus obtaining the partial trajectory \( \tilde{y}_{k+1} \) to start the next stage.

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

The rollout algorithm just described is illustrated in Fig. 2.5.3 for our earlier traveling salesman Example 2.5.1. Here we want to find a minimum travel cost tour that additionally satisfies a safety constraint, namely that the “safety cost” of the tour should be less than a certain threshold.
Figure 2.5.3 The constrained traveling salesman problem; cf. Example 2.5.1, and its rollout solution using the base heuristic shown, which completes a partial tour as follows:

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

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

At city A, the rollout algorithm:

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

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

(c) Discards ADCBA as being infeasible.

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

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

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

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

**Sequential Consistency, Sequential Improvement, and the Cost Improvement Property**

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

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

while at the first step of the algorithm we have

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

It will then follow that the cost improvement property

\[
G(\tilde{y}_N) \leq G(R(\tilde{y}_0))
\]

holds.

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

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

starting from a partial trajectory \( y_k \), it also generates the partial trajectory

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

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

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

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

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

and

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

we have

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Proposition 2.5.1 establishes the fundamental cost improvement property for constrained rollout under the sequential improvement condition. On the other hand we may construct examples where the sequential improvement condition (2.35) is violated and the cost of the solution produced by rollout is larger than the cost of the solution produced by the...
Approximation in Value Space - Rollout Algorithms Chap. 2

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

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

The Fortified Rollout Algorithm and Other Variations

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**Tree-Based Constrained Rollout Algorithms**

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

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

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

### 2.5.1 Constrained Rollout for Discrete Optimization and Integer Programming

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

#### Example 2.5.2 (Facility Location)

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

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

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

(2.37)
Sec. 2.5 Constrained Forms of Rollout Algorithms

59

\[ C : \text{Set of } (u_0, \ldots, u_{N-1}) \text{ such that } u_k \in \{0, 1\} \]

and can satisfy the demand and other constraints (e.g., public policy constraints)

Cost Function:

\[ G(u_0, \ldots, u_{N-1}) = \min_{(y_{ik}, i, k) \in H(u_0, \ldots, u_{N-1})} \sum_{i=0}^{M} \sum_{k=0}^{N-1} a_{ik} y_{ik} + \sum_{k=0}^{N-1} b_k u_k \]

where

\[ H(u_0, \ldots, u_{N-1}) : \text{Set of feasible demand allocations, i.e. } \]

\[ \sum_{i=0}^{M} y_{ik} = d_i \text{ for all } i, \]

\[ \sum_{i=0}^{M} y_{ik} \leq u_k c_k \text{ for all } k \]

\[ \sum_{i=0}^{M} y_{ik} \geq 0 \]

\[ u_k \in \{0, 1\}, \quad i = 1, \ldots, M, \quad k = 0, \ldots, N - 1. \]  

(2.38)

(2.39)

We wish to minimize the cost

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

(2.40)

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

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

To address the problem suboptimally by rollout, we must define a base heuristic at a “state” \( (u_0, \ldots, u_k) \), where \( u_j = 1 \) or \( u_j = 0 \) specifies that a facility is or is not placed at location \( j \), respectively. A suitable base heuristic at that state is to place a facility at all of the remaining locations (i.e., \( u_j = 1 \) for \( j = k+1, \ldots, N-1 \)), and its cost is obtained by solving the corresponding linear transportation problem of minimizing the cost (2.40) subject to the constraints (2.37)-(2.39), with the variables \( u_j, j = 0, \ldots, k, \) fixed at the previously chosen values, and the variables \( u_j, j = k+1, \ldots, N, \) fixed at 1.
To illustrate, at the initial state where no placement decision has been made, we set \( u_0 = 1 \) (a facility is placed at location 0) or \( u_0 = 0 \) (a facility is not placed at location 0), we solve the two corresponding transportation problems, and we fix \( u_0 \), depending on which of the two resulting costs is smallest. Having fixed the status of location 0, we repeat with location 1: set the variable \( u_1 \) to 1 and to 0, solve the corresponding two transportation problems, and fix \( u_1 \), depending on which of the two resulting costs is smallest, etc.

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

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

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

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

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

**Example 2.5.3 (Constrained Shortest Paths and Directed Spanning Trees)**

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

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

with \( u_k \in U_k, k = 0, \ldots, N - 1 \). We require that these arcs do not form a cycle, so that \( u \) specifies a directed spanning tree that is rooted at node
Figure 2.5.5 Schematic illustration of a constrained shortest path problem with root node $N = 6$. Given the current feasible spanning tree solution (indicated with solid line arcs), the rollout algorithm, considers a node $k$ (in the figure $k = 0$) and the spanning tree arcs $\{u_i \mid i \neq k\}$ that are outgoing from the nodes $i \neq k$. It then considers the spanning trees that correspond to the outgoing arcs $u_k$ from $k$ that do not close a cycle with the set $\{u_i \mid i \neq k\}$ and are feasible [in the figure, these are the arcs indicated with broken lines, plus the arc $(0,4)$], and selects the arc that forms a spanning tree solution of minimum cost.

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

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

Suppose that we have an initial solution/directed spanning tree

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

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

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

...
i.e., a sequence of \( k \) arcs, each outgoing from one of the nodes \( 0, \ldots, k - 1 \), it generates the complete trajectory/directed spanning tree
\[
(u_0, \ldots, u_{k-1}, \bar{u}_k, \ldots, \bar{u}_{N-1}).
\]
Thus the rollout algorithm, given a partial trajectory
\[
\tilde{y}_k = (\tilde{u}_0, \ldots, \tilde{u}_{k-1}),
\]
considers the set \( \hat{U}_k(\tilde{y}_k) \) of all outgoing arcs \( u_k \) from node \( k \), such that the complete trajectory
\[
(\tilde{y}_k, u_k, \bar{u}_{k+1}, \ldots, \bar{u}_{N-1})
\]
is feasible. It then selects the arc \( u_k \in \hat{U}_k(\tilde{y}_k) \) that minimizes the cost
\[
G(\tilde{y}_k, u_k, \bar{u}_{k+1}, \ldots, \bar{u}_{N-1});
\]
see Fig. 2.5.5. It can be seen by induction, starting from \( \bar{u} \), that the set of arcs \( \hat{U}_k(\tilde{y}_k) \) is nonempty, and that the algorithm generates a sequence of feasible solutions/directed spanning trees, each with cost no worse than the preceding one.

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

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

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

Let us consider the deterministic one-step approximation in value space scheme
\[
\hat{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \left[ g_k(x_k, u_k) + \hat{J}_{k+1}(f_k(x_k, u_k)) \right]. \tag{2.41}
\]
In the context of rollout, \( \hat{J}_{k+1}(f_k(x_k, u_k)) \) is either the cost of the trajectory generated by the base heuristic starting from the next state \( f_k(x_k, u_k) \), or
Sec. 2.6  Small Stage Costs and Long Horizon - Continuous-Time Rollout

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

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

**Continuous-Time Optimal Control and Approximation in Value Space**

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

\[
\dot{x}(t) = h(x(t), u(t), t), \quad 0 \leq t \leq T,
\]

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

\[
G(x(T)) + \int_0^T g(x(t), u(t), t)dt,
\]

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

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

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

Here the function \( h_k \) is given by

\[
h_k(x_k, u_k) = h(x(k\delta), u(k\delta), k\delta),
\]

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

\[
x_k \approx x(k\delta), \quad u_k \approx u(k\delta).
\]
We also replace the cost function (2.43) by
\[ g_N(x_N) + \sum_{k=0}^{N-1} \delta \cdot g_k(x_k, u_k), \]
where
\[ g_N(x_N) = G(x(N\delta)), \quad g_k(x_k, u_k) = g(x(k\delta), u(k\delta), k\delta). \]

Thus the approximation in value space scheme with time discretization takes the form
\[ \tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U} \left[ \delta \cdot g_k(x_k, u_k) + \tilde{J}_{k+1}(x_k + \delta \cdot h_k(x_k, u_k)) \right], \tag{2.44} \]
where \( \tilde{J}_{k+1} \) is the function that approximates the cost-to-go starting from a state at time \( k+1 \). We note here that the ratio of the terms \( \delta \cdot g_k(x_k, u_k) \) and \( \tilde{J}_{k+1}(x_k + \delta \cdot h_k(x_k, u_k)) \) is likely to tend to 0 as \( \delta \to 0 \), since \( \tilde{J}_{k+1}(x_k + \delta \cdot h_k(x_k, u_k)) \) ordinarily stays roughly constant at a nonzero level as \( \delta \to 0 \). This suggests that the one-step lookahead minimization may be degraded substantially by discretization, and other errors, including rollout truncation and terminal cost approximation. Note that a similar sensitivity to errors may occur in other discrete-time models that involve frequent selection of decisions, with cost per stage that is very small relative to the cumulative cost over many stages and/or the terminal cost.

To deal with this difficulty, we subtract the constant \( \tilde{J}_k(x_k) \) in the one-step-lookahead minimization (2.44), and write
\[ \tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U} \left[ \delta \cdot g_k(x_k, u_k) + \tilde{J}_{k+1}(x_k + \delta \cdot h_k(x_k, u_k)) - \tilde{J}_k(x_k) \right], \tag{2.45} \]

since \( \tilde{J}_k(x_k) \) does not depend on \( u_k \), the results of the minimization are not affected. Assuming \( \tilde{J}_k \) is differentiable with respect to its argument, we can write
\[ \tilde{J}_{k+1}(x_k + \delta \cdot h_k(x_k, u_k)) - \tilde{J}_k(x_k) \approx \delta \cdot \nabla_x \tilde{J}_k(x_k)^t h_k(x_k, u_k), \]
where \( \nabla_x \tilde{J}_k \) denotes the gradient of \( \tilde{J}_k \) (a column vector), and prime denotes transposition. By dividing with \( \delta \), and taking informally the limit as \( \delta \to 0 \), we can write the one-step lookahead minimization (2.45) as
\[ \tilde{\mu}(t) \in \arg \min_{u(t) \in U} \left[ g(x(t), u(t), t) + \nabla_x \tilde{J}_t(x(t))^t h(x(t), u(t), t) \right], \tag{2.46} \]
where \( \tilde{J}_t(x) \) is the continuous-time cost function approximation and \( \nabla_x \tilde{J}_t(x) \) is its gradient with respect to \( x \). This is the correct analog of the approximation in value space scheme (2.41) for continuous-time problems.
Rollout for Continuous-Time Optimal Control

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

$$
\pi = \left\{ \mu_t(x(t)) \mid 0 \leq t \leq T \right\},
$$

(2.47)

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

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

(2.48)

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

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

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

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

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

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

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

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

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

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

In this case, in view of Eq. (2.46), the rollout policy is myopic. It turns out that the optimal policy in this example is also myopic, so that the rollout policy is optimal, even though the base policy is very poor.
Example 2.6.1 (A Calculus of Variations Problem)

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

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

\[
\sqrt{(dt)^2 + (\dot{x}(t)dt)^2},
\]

which is equal to

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

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

\[
\text{minimize } \int_0^T \sqrt{1 + (\dot{x}(t))^2} \, dt \quad \text{subject to } x(0) = 0.
\]

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

\[
\dot{x}(t) = u(t), \quad x(0) = 0.
\]

Our problem then takes the form

\[
\text{minimize } \int_0^T \sqrt{1 + (u(t))^2} \, dt.
\]
This is a problem that fits our continuous-time optimal control framework, with

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

Consider now a base policy \( \pi \) whereby the control depends only on \( t \) and not on \( x \). Such a policy has the form

\[ \mu_t(x(t)) = \beta(t), \quad \text{for all } x(t), \]

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

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

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

\[ \hat{\mu}_t(x(t)) \in \arg \min_{u(t) \in U} \sqrt{1 + (u(t))^2}, \]

and the rollout policy is

\[ \hat{\mu}_t(x(t)) \equiv 0. \]

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

**Rollout with General Base Heuristics - Sequential Improvement**

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

\[ \hat{\mu}(t) \in \arg \min_{u(t) \in U} \left[ g(x(t), u(t), t) + \nabla_x H_t(x(t)) \left( \frac{d}{dt} h(x(t), u(t), t) \right) \right]. \]

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

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

\[ H_T(\dot{x}(T)) = G(\dot{x}(T)), \quad (2.49) \]
and the assumption
\[
\min_{u(t) \in U} \left[ g(x(t), u(t), t) + \nabla_t H_t(x(t)) + \nabla_x H_t(x(t))' h(x(t), u(t), t) \right] \leq 0,
\]
(2.50)
for all \((x(t), t)\), where \(\nabla_x H_t\) denotes gradient with respect to \(x\), and \(\nabla_t H_t\) denotes gradient with respect to \(t\). This assumption is the continuous-time analog of the sequential improvement condition of Definition 2.3.2 [cf. Eq. (2.15)]. Under this assumption, we will show that
\[
J_{x,0}(x(0)) \leq H_0(x(0)),
\]
(2.51)
i.e., the cost of the rollout policy starting from the initial state \(x(0)\) is no worse than the base heuristic cost starting from the same initial state.

Indeed, let \(\{\tilde{x}(t) \mid t \in [0,T]\}\) and \(\{\tilde{u}(t) \mid t \in [0,T]\}\) be the state and control trajectories generated by the rollout policy starting from \(x(0)\). Then the sequential improvement condition (2.50) yields
\[
g(\tilde{x}(t), \tilde{u}(t), t) + \nabla_t H_t(\tilde{x}(t)) + \nabla_x H_t(\tilde{x}(t))' h(\tilde{x}(t), \tilde{u}(t), t) \leq 0
\]
for all \(t\), and by integration over \([0,T]\), we obtain
\[
\int_0^T g(\tilde{x}(t), \tilde{u}(t), t) dt + \int_0^T \left( \nabla_t H_t(\tilde{x}(t)) + \nabla_x H_t(\tilde{x}(t))' h(\tilde{x}(t), \tilde{u}(t), t) \right) dt \leq 0.
\]
(2.52)
The second integral above can be written as
\[
\int_0^T \left( \nabla_t H_t(\tilde{x}(t)) + \nabla_x H_t(\tilde{x}(t))' h(\tilde{x}(t), \tilde{u}(t), t) \right) dt
\]
\[
= \int_0^T \left( \nabla_t H_t(\tilde{x}(t)) + \nabla_x H_t(\tilde{x}(t)) \frac{d\tilde{x}(t)}{dt} \right) dt,
\]
and its integrand is the total differential with respect to time: \(\frac{d}{dt} \left( H_t(\tilde{x}(t)) \right)\).

Thus we obtain from Eq. (2.52)
\[
\int_0^T g(\tilde{x}(t), \tilde{u}(t), t) dt + \int_0^T \frac{d}{dt} \left( H_t(\tilde{x}(t)) \right) dt
\]
\[
= \int_0^T g(\tilde{x}(t), \tilde{u}(t), t) dt + H_T(\tilde{x}(T)) - H_0(\tilde{x}(0)) \leq 0.
\]
(2.53)
Since \(H_T(\tilde{x}(T)) = G(\tilde{x}(T))\) [cf. Eq. (2.49)] and \(\tilde{x}(0) = x(0)\), from Eq. (2.53) [which is a direct consequence of the sequential improvement condition (2.50)], it follows that
\[
J_{x,0}(x(0)) = \int_0^T g(\tilde{x}(t), \tilde{u}(t), t) dt + G(\tilde{x}(T)) \leq H_0(x(0)),
\]
thus proving the cost improvement property (2.51).

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

\[
0 = g(x(t), \mu_t(x(t)), t) + \nabla_t J_{\pi,t}(x(t)) + \nabla_x J_{\pi,t}(x(t)) h(x(t), \mu_t(x(t)), t).
\]

(2.54)

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

**Approximating Cost Function Differences**

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

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

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

\[
\hat{\mu}_k(x_k) = \arg \min_{u_k \in U_k(x_k)} \left[ g_k(x_k, u_k) + \hat{D}_k(x_k, u_k) \right],
\]

where \( \hat{D}_k \) is the approximation to \( D^*_k \).

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

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

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

2.7 STOCHASTIC ROLLOUT AND MONTE CARLO TREE SEARCH

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

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

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

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

where

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

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

$$J_{k,\hat{\pi}}(x_k) \leq J_{k,\pi}(x_k), \quad \text{for all } x_k \text{ and } k. \quad (2.55)$$

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

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

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

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

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

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

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

$$= J_{k,\pi}(x_k), \quad (2.56)$$

where:

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

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

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

(d) The final equality is the DP equation for the base policy $\pi$.

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

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

**Some Rollout Examples**

Similar to deterministic problems, it has been observed empirically that for stochastic problems the rollout policy not only does not deteriorate
the performance of the base policy, but also typically produces substantial cost improvement, thanks to its underlying Newton step; see also the case studies referenced at the end of the chapter. To emphasize this point, we provide here an example of an nontrivial optimal stopping problem where the rollout policy is actually optimal, despite the fact that the base policy is rather naive. Such behavior is of course special and nontypical, but highlights the nature of the cost improvement property of rollout.

**Example 2.7.1 (Optimal Stopping and Rollout Optimality)**

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

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

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

$$J^*_N(x_N) = C(x_N),$$

$$J^*_k(x_k) = \min \left\{ C(x_k), E\left\{ g(x_k, w_k) + J^*_{k+1}(f(x_k, w_k)) \right\} \right\},$$

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

$$S_k = \left\{ x \mid C(x) \leq E\left\{ g(x, w) + J^*_{k+1}(f(x, w)) \right\} \right\}.$$

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

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

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

$$C(x_k) \leq E\left\{ g(x_k, w_k) + C(f(x_k, w_k)) \right\},$$

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

The rollout policy also has an intuitive interpretation: it stops at the states for which it is better to stop rather than continue for one more stage.
and then stop. A policy of this type turns out to be optimal in several types of stopping applications. Let us provide a condition that guarantees its optimality.

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

\[ J_{N-1}^*(x) \leq J_N^*(x), \quad \text{for all } x, \]

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

\[ J_k^*(x) \leq J_{k+1}^*(x), \quad \text{for all } x \text{ and } k. \]

Using this fact and the definition of \( S_k \) we see that

\[ S_0 \subset \cdots \subset S_k \subset S_{k+1} \subset \cdots \subset S_{N-1}. \]

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

\[ f(x, w) \in S_{N-1}, \quad \text{for all } x \in S_{N-1}, w. \]

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

\[ S_k = S_{N-1} = \left\{ x \in S \mid C(x) \leq E\left\{ g(x, w) + C(f(x, w)) \right\} \right\}. \]

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

\[ J_{N-1}^*(x) = C(x), \quad \text{for all } x \in S_{N-1}, \]

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

\[ E\left\{ g(x, w) + J_{N-1}^*(f(x, w)) \right\} = E\left\{ g(x, w) + C(f(x, w)) \right\} \geq C(x). \]

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

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

We next discuss a special case of the preceding example. Again the one-step lookahead/rollout policy is optimal, despite the fact that the base policy is poor. Related examples can be found in Chapter 3 of the DP textbook [Ber17a].
Example 2.7.2 (The Rational Burglar)

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

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

\[
J^*_{N}(x_N) = x_N, \\
J^*_{k}(x_k) = \max \left[ x_k, (1 - p)E\{J^*_{k+1}(x_k + w_k)\} \right].
\]

The one-step stopping set is

\[
S_{N-1} = \{ x \mid x \geq (1 - p)(x + \overline{w}) \} = \left\{ x \mid x \geq \frac{(1 - p)\overline{w}}{p} \right\},
\]

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

2.7.1 Simplified Rollout and Policy Iteration

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

\[
\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} Q_{k,\pi}(x_k, u_k), \quad (2.61)
\]

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

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

with the inequality

\[
\min_{u_k \in \overline{U}_k(x_k)} Q_{k,\pi}(x_k, u_k) \leq Q_{k,\pi}(x_k, \mu_k(x_k)).
\]
The simplified rollout algorithm (2.61) may be implemented in a number of ways, including control constraint discretization/approximation, a random search algorithm, or a one-agent-at-a-time minimization process, as in multiagent rollout.

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

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

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

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

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

### 2.7.2 Certainty Equivalence Approximations

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

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

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

This approach, first proposed in the paper by Bertsekas and Castañón [BeC99], has an important advantage: it maintains the Newton step character of the approximation in value space scheme. In particular, the function
Approximation in Value Space - Rollout Algorithms Chap. 2

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

2.7.3 Simulation-Based Implementation of the Rollout Algorithm

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

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

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 \( \tilde{Q}_{k,\pi}(x_k, u_k) \). We then compute the (approximate) rollout control \( \tilde{\mu}_k(x_k) \) with the minimization

\[
\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \tilde{Q}_{k,\pi}(x_k, u_k). \tag{2.63}
\]
Figure 2.7.1 Illustration of multistep lookahead for stochastic problems with the CE approximation, applied at the states after the first layer of states of the multistep lookahead tree. The figure on the top (or the bottom) illustrates the lookahead tree without (or with, respectively) CE. It can be seen that with CE, the lookahead tree grows much faster (the layers contain more states). In particular, the “height” of the $\ell$-step lookahead graph without the CE approximation is the same as the “height” of a $\ell'$-step lookahead graph with the CE approximation, where $\ell' = 2\ell - 1$. Moreover, with a number $m$ of controls per state, and a number $n$ of disturbances per state-control pair, the number of leaves of the $\ell$-step lookahead tree is estimated as $O(mn\ell)$ without CE and $O\left(m(n + \ell)\right)$ with CE.

Example 2.7.3 (Backgammon)

The first impressive application of rollout was given for the ancient two-player game of backgammon, in the paper by Tesauro and Galperin [TeG96]; see Fig. 2.7.2. They implemented a rollout algorithm, which attained a level of play that was better than all computer backgammon programs, and eventually better than the best humans. Tesauro had proposed earlier the use of one-step and two-step lookahead with lookahead cost function approximation provided by a neural network, resulting in a backgammon program called TD-Gammon [Tes89a], [Tes89b], [Tes92], [Tes94], [Tes95], [Tes02]. TD-Gammon was trained with an approximate policy iteration method, and was used as
Figure 2.7.2 Illustration of rollout for backgammon. At a given position and roll of the dice, the set of all possible moves is generated, and the outcome of the game for each move is evaluated by “rolling out” (simulating to the end) many games using a suboptimal/heuristic backgammon player (the TD-Gammon player was used for this purpose in [TeG96]), and by Monte Carlo averaging the scores. The move that results in the best average score is selected for play.

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

An important issue to consider here is that backgammon is a two-player game and not an optimal control problem that involves a single decision maker. While there is a DP theory for sequential zero-sum games, this theory has not been covered in these notes. Thus how are we to interpret rollout algorithms in the context of two-player games, with both players using some base policy? The answer is to view the game as a (one-player) optimal control problem, where one of the two players passively uses the base policy exclusively (TD-Gammon in the present example). The other player takes the role of the optimizer, and actively tries to improve on his base policy (TD-Gammon) by using rollout. Thus “policy improvement” in the context of the present example means that when playing against a TD-Gammon opponent, the rollout player achieves a better score on the average than if he/she were to play with the TD-Gammon strategy. In particular, the theory does not guarantee that a rollout player that is trained using TD-Gammon for both players will do better than TD-Gammon would against a non-TD-Gammon opponent. While this is a plausible practical hypothesis, it is one that can only
be tested empirically. In fact relevant counterexamples have been constructed for the game of Go using “adversarial” optimization techniques; see Wang et al. [WGB22], and also our discussion on minimax problems in Section 2.12.

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

2.7.4 Variance Reduction in Rollout - Comparing Advantages

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

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

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

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

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

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

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

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

† The situation in backgammon is exacerbated by its high branching factor, i.e., for a given position, the number of possible successor positions is quite large, as compared for example with chess.
with \( \{ \mu_{k+1}, \ldots, \mu_{N-1} \} \) being the tail portion of the base policy. \( \dagger \)

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

The preceding example suggests a simulation scheme that is based on the difference (2.64) and involves a common disturbance \( w_k \) for \( u_k \) and \( \hat{u}_k \). In particular, it may be far more accurate than the one obtained by differencing samples of \( C_k(x_k, u_k, w_k) \) 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

\[
D_k(x_k, u_k, w_k) = C_k(x_k, u_k, w_k) - \tilde{Q}_{k,\pi}(x_k, u_k),
\]

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

\[
E_{w_k, \hat{w}_k} \left\{ \left| D_k(x_k, u_k, w_k) - D_k(x_k, \hat{u}_k, \hat{w}_k) \right|^2 \right\} \geq E_{w_k} \left\{ \left| D_k(x_k, u_k, w_k) - D_k(x_k, \hat{u}_k, w_k) \right|^2 \right\}.
\]

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

\[
E\{D_k(x_k, u_k, w_k)D_k(x_k, \hat{u}_k, w_k)\} \geq 0; \quad (2.65)
\]

i.e., the errors \( D_k(x_k, u_k, w_k) \) and \( D_k(x_k, \hat{u}_k, w_k) \) 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 (2.65) holds if changes in the value of \( u_k \) (at the first stage) have little effect on the value of the error \( D_k(x_k, u_k, w_k) \) 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

\[
E \left\{ \left| D_k(x_k, u_k, w_k) - D_k(x_k, \hat{u}_k, w_k) \right|^2 \right\} \leq \gamma E \left\{ \left| D_k(x_k, u_k, w_k) \right|^2 \right\}. \quad (2.66)
\]

\( \dagger \) For this to be possible, we need to assume that the probability distribution of each disturbance \( w_i \) does not depend on \( x_i \) and \( u_i \).
Then we have, by using the generic relation \( ab \geq a^2 - |a| \cdot |b - a| \) for two scalars \( a \) and \( b \),

\[
D_k(x_k, u_k, w_k)D_k(x_k, \hat{u}_k, w_k) \\
\geq |D_k(x_k, u_k, w_k)|^2 \\
- |D_k(x_k, u_k, w_k)| \cdot |D_k(x_k, \hat{u}_k, w_k) - D_k(x_k, u_k, w_k)|,
\]

from which we obtain

\[
E\{ D_k(x_k, u_k, w_k)D_k(x_k, \hat{u}_k, w_k) \} \\
\geq E\left\{ |D_k(x_k, u_k, w_k)|^2 \right\} \\
- E\left\{ |D_k(x_k, u_k, w_k)| \cdot |D_k(x_k, \hat{u}_k, w_k) - D_k(x_k, u_k, w_k)| \right\} \\
\geq E\left\{ |D_k(x_k, u_k, w_k)|^2 \right\} - \frac{1}{2} E\left\{ |D_k(x_k, u_k, w_k)|^2 \right\} \\
- \frac{1}{2} E\left\{ |D_k(x_k, \hat{u}_k, w_k) - D_k(x_k, u_k, w_k)|^2 \right\} \\
\geq \frac{1 - \gamma}{2} E\left\{ |D_k(x_k, u_k, w_k)|^2 \right\},
\]

where for the second inequality we use the generic relation

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

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

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

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

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

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

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

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

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

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

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

A simple remedy for (a) above is to use rollout trajectories of reasonably limited length, with some terminal cost approximation at the end (in an extreme case, the rollout may be skipped altogether for some states, i.e., rollout trajectories have zero length). The terminal cost function may be very simple (such as zero) or may be obtained through some auxiliary calculation. In fact the base policy used for rollout may be used to construct the terminal cost function approximation, as noted for the rollout-based backgammon algorithm of Example 2.7.3. In particular, an approximation to the cost function of the base policy may be obtained by training some approximation architecture, such as a neural network (see Chapter 3), and may be used as a terminal cost function.

A simple but less straightforward remedy for (b) is to use some heuristic or statistical test to discard some of the controls $u_k$, as soon as this is suggested by the early results of simulation. Similarly, to implement (c) one may use some heuristic to increase the length of lookahead selectively for some of the controls $u_k$. This is similar to the incremental multistep rollout scheme for deterministic problems that we discussed in Section 2.4.3; see Fig. 2.4.6.

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

In the simple case of one-step lookahead, with Q-factors calculated by Monte Carlo simulation, MCTS fundamentally aims to find efficiently the minimum of the expected values of a finite number of random variables. This is illustrated in the following example.

**Example 2.7.4 (Statistical Tests for Adaptive Sampling with One-Step Lookahead)**

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

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

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

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

where

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

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

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

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

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

of two indexes: an exploitation index $T_{i,n}$ and an exploration index $R_{i,n}$. Usually the exploitation index is chosen to be the empirical mean $Q_{i,n}$; see Fig. 2.7.3. The exploration index is based on a confidence interval formula and depends on the sample count

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

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

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

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

† The paper [ACF02] refers to the rule given here as UCB1 and credits its motivation to the paper by Agrawal [Agr95]. The book by Lattimore and Szepesvari [LaS20] provides an extensive discussion of the UCB rule and its generalizations.
Sec. 2.7  Stochastic Rollout and Monte Carlo Tree Search

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

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

A major success has been the use of MCTS in two-player game contexts, such as the AlphaGo program (Silver et al. [SHM16]), which performs better than the best humans in the game of Go. This program integrates several of the techniques discussed in these notes, including MCTS and rollout using a base policy that is trained off-line using a deep neural network. The AlphaZero program, which has performed spectacularly well against humans and other programs in the games of Go and chess (Silver et al. [SHS17]), bears some similarity with AlphaGo, and critically relies on MCTS, but does not use rollout in its on-line playing mode (it relies primarily on very long lookahead).

2.7.6 Randomized Policy Improvement by Monte Carlo Tree Search

We have described rollout and MCTS as schemes for policy improvement: start with a base policy, and compute an improved policy based on the results of one-step lookahead or multistep lookahead followed by simulation with the base policy. We have implicitly assumed that both the base policy and the rollout policy are deterministic in the sense that they map each state $x_k$ into a unique control $\tilde{\mu}_k(x_k)$ [cf. Eq. (2.63)]. In some (even nonstochastic) contexts, success has been achieved with randomized policies, which map a state $x_k$ to a probability distribution over the set of controls $U_k(x_k)$, rather than mapping onto a single control. In particular, the AlphaGo and AlphaZero programs use MCTS to generate and use for training purposes randomized policies, which specify at each board position the probabilities with which the various moves are selected.

A randomized policy can be used as a base policy in a rollout context in exactly the same way as a deterministic policy: for a given state $x_k$, we just generate sample trajectories and associated sample Q-factors, using probabilistically selected controls, starting from each leaf-state of the lookahead tree that is rooted at $x_k$. We then average the corresponding Q-factor samples. The rollout/improved policy, as described here, is a deterministic policy, i.e., it applies at $x_k$ the control $\tilde{\mu}_k(x_k)$ that is “best” according to the results of the rollout [cf. Eq. (2.63)]. Still, however, if we
wish to generate an improved policy that is randomized, we can simply change the probabilities of different controls in the direction of the deterministic rollout policy. This can be done by increasing by some amount the probability of the “best” control $\tilde{\mu}_k(x_k)$ from its base policy level, while proportionally decreasing the probabilities of the other controls.

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

2.8 ROLLOUT FOR INFINITE-SPACES PROBLEMS - OPTIMIZATION HEURISTICS

We have considered so far finite control space applications of rollout, so there is a finite number of relevant Q-factors at each state $x_k$, which are evaluated by simulation and are exhaustively compared. When the control constraint set is infinite, to implement this approach the constraint set must be replaced by a finite set, obtained by some form of discretization or random sampling, which can be inconvenient and ineffective. In this section we will discuss an alternative approach to deal with an infinite number of controls and Q-factors at $x_k$. The idea is to use a base heuristic that involves a continuous optimization, and to rely on a linear or nonlinear programming method to solve the corresponding lookahead optimization problem.

2.8.1 Rollout for Infinite-Spaces Deterministic Problems

To develop the basic idea of how to deal with infinite control spaces, we first consider deterministic problems, involving a system

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

and a cost per stage $g_k(x_k, u_k)$. The one-step lookahead rollout minimization is

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

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

$$\hat{Q}_k(x_k, u_k) = g_k(x_k, u_k) + H_{k+1}(f_k(x_k, u_k)),$$
with \( H_{k+1}(x_{k+1}) \) being the cost of the base heuristic starting from state \( x_{k+1} \) \([cf. Eq. (2.12)]\). Suppose that we have a differentiable closed-form expression for \( H_{k+1} \), and the functions \( g_k \) and \( f_k \) are known and are differentiable with respect to \( u_k \). Then the \( Q \)-factor \( Q_k(x_k, u_k) \) of Eq. (2.68) is also differentiable with respect to \( u_k \), and its minimization (2.67) may be addressed with one of the many gradient-based methods that are available for differentiable unconstrained and constrained optimization.

The preceding approach requires that the heuristic cost \( H_{k+1}(x_{k+1}) \) can be differentiated, so it should either be available in closed form, which is quite restrictive, or that it can be differentiated numerically, which may be inconvenient and/or unreliable. These difficulties can be circumvented by using a base heuristic that is itself based on multistep optimization. In particular, suppose that \( H_{k+1}(x_{k+1}) \) is the optimal cost of some \((\ell - 1)\)-stage deterministic optimal control problem that is related to the original problem. Then the rollout algorithm (2.67)-(2.68) can be implemented by solving the \( \ell \)-stage deterministic optimal control problem, which seamlessly concatenates the first stage minimization over \( u_k \) \([cf. Eq. (2.67)]\), with the \((\ell - 1)\)-stage minimization of the base heuristic; see Fig. 2.8.1. This \( \ell \)-stage problem may be solvable on-line by standard continuous spaces nonlinear programming or optimal control methods.† A major paradigm of methods of this type is model predictive control, which we have discussed in Chapter

† Note, however, that for this to be possible, it is necessary to have a mathematical model of the system; a simulator is not sufficient. Another difficulty
Figure 2.8.2. Illustration of a simple supply chain system for Example 2.8.1.

1 (cf. Section 1.6.7). In the present section we will discuss a few other possibilities. The following is a simple example of an important class of inventory storage and supply chain management processes.

**Example 2.8.1 (Supply Chain Management)**

Let us consider a supply chain system, where a certain item is produced at a production center and fulfilled at a retail center. Stock of the item is shipped from the production center to the retail center, where it arrives with a delay of \(\tau \geq 1\) time units, and is used to fulfill a known stream of demands \(d_k\) over an \(N\)-stage horizon; see Fig. 2.8.2. We denote:

- \(x^1_k\): 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

\[
x^1_{k+1} = x^1_k + u^1_k - u^2_k, \quad x^2_{k+1} = x^2_k + u^2_{k-\tau} - d_k,
\]

and involves the delayed control component \(u^2_{k-\tau}\). Thus the exact DP algorithm involves state augmentation as introduced in Section 1.6.3, and may thus be much more complicated than in the case where there are no delays.

occurs when the control space is the union of a discrete set and a continuous set. Then it may be necessary to use some type of mixed integer programming technique to solve the \(\ell\)-stage problem. Alternatively, it may be possible to handle the discrete part by brute force enumeration, followed by continuous optimization.

† Despite the fact that with large delays, the size of the augmented state space can become very large (cf. Section 1.6.3), the implementation of rollout schemes is not affected much by this increase in size. For this reason, rollout can be very well suited for problems involving delayed effects of past states and controls.
The cost at time \( k \) consists of three components: a production cost that depends on \( x^1_k \) and \( u^1_k \), a transportation cost that depends on \( u^2_k \), and a fulfillment cost that depends on \( x^2_k \) [which includes positive costs for both excess inventory (i.e., \( x^2_k > d_k \)) and for backordered demand (i.e., \( x^2_k < d_k \))]. The precise forms of these cost components are immaterial for the purposes of this example.

Here the control vector \( u_k \) is often continuous (or a mixture of discrete and continuous components), so it may be essential for the purposes of rollout to use the continuous optimization framework of this section. In particular, at the current stage \( k \), we know the current state, which includes \( x^1_k, x^2_k \), and the amounts of stock in transit together with their scheduled arrival times at the retail center. We then apply some heuristic optimization to determine the stream of future production and shipment levels over \( \ell \) steps, and use the first component of this stream as the control applied by rollout. As an example we may use as base policy one that brings the retail inventory to some target value \( \ell \) stages ahead, and possibly keep it at that value for a portion of the remaining periods. This is a nonlinear programming or mixed integer programming problem that may be solvable with available software far more efficiently than by a discretized form of DP.

A major benefit of rollout in the supply chain context is that it can readily incorporate on-line replanning. This is necessary when unexpected demand changes, production or transport equipment failures occur, or updated forecasts become available.

The following example deals with a common class of problems of resource allocation over time.

**Example 2.8.2 (Multistage Linear and Mixed Integer Programming)**

Let us consider a deterministic optimal control problem with linear system equation

\[
x_{k+1} = A_k x_k + B_k u_k + d_k, \quad k = 0, \ldots, N - 1,
\]

where \( A_k \) and \( B_k \) are known matrices of appropriate dimension, \( d_k \) is a known vector, and \( x_k \) and \( u_k \) are column vectors. The cost function is linear of the form

\[
c_N' x_N + \sum_{k=0}^{N-1} (c_k' x_k + d_k' u_k),
\]

where \( c_k \) and \( d_k \) are known column vectors of appropriate dimension, and a prime denotes transpose. The terminal state and state-control pairs \( (x_k, u_k) \) are constrained by

\[
x_N \in T, \quad (x_k, u_k) \in P_k, \quad k = 0, \ldots, N - 1,
\]

where \( T \) and \( P_k, k = 0, \ldots, N - 1 \), are given sets, which are specified by linear and possibly integer constraints.
As an example, consider a multi-item production system, where the state is $x_k = (x_1^k, \ldots, x_n^k)$ and $x_i^k$ represents stock of item $i$ available at the start of period $k$. The state evolves according to the system equation

$$x_{k+1}^i = \sum_{j=1}^{n} a_{kj}^i u_{kj}^i - d_k^i, \quad i = 1, \ldots, n,$$

where $u_{kj}^i$ is the amount of product $i$ that is used during time $k$ for the manufacture of product $j$, $a_{kj}^i$ are known scalars that are related to the underlying production process, and $d_k^i$ is a deterministic demand of product $i$ that is fulfilled at time $k$. One constraint here is that

$$\sum_{j=1}^{n} u_{kj}^i \leq x_k^i, \quad i = 1, \ldots, n,$$

and there are additional linear and integer constraints on $(x_k, u_k)$, which are collected in a general constraint of the form $(x_k, u_k) \in P_k$ (e.g., nonnegativity, production capacity, storage constraints, etc). Note that the problem may be further complicated by production delays, as in the preceding supply chain Example 2.8.1. Moreover, while in this section we focus on deterministic problems, we may envision a stochastic version of the problem where the demands $d_k^i$ are random with given probability distributions, which are subject to revisions based on randomly received forecasts.

The problem may be solved using a linear or mixed integer programming algorithm, but this may be very time-consuming when $N$ is large. Moreover, the problem will need to be resolved on-line if some of the problem data changes and replanning is necessary. A suboptimal alternative is to use truncated rollout with an $\ell$-stage mixed integer optimization, and a polyhedral terminal cost function $\tilde{J}_{k+\ell}$ to provide a terminal cost optimization. A simple possibility is no terminal cost $[\tilde{J}_{k+\ell}(x_{k+\ell}) \equiv 0]$, and another possibility is a polyhedral lower bound approximation that can be based on relaxing the integer constraints after stage $k + \ell$, or some kind of training approach that uses data.

We will next discuss how rollout can accommodate stochastic disturbances by using deterministic optimization ideas based on certainty equivalence (cf. Section 2.7.4) and the methodology of stochastic programming.

### 2.8.2 Rollout Based on Stochastic Programming

We have focused so far in this section on rollout that relies on deterministic continuous optimization. There is an important class of methods, known as stochastic programming, which can be used for stochastic optimal control, but bears a close connection to continuous spaces deterministic optimization. We will first describe this connection for two-stage problems, then discuss extensions to many-stage problems, and finally show how rollout can be brought to bear for their approximate solution.
Example 2.8.3 (Two-Stage Stochastic Programming)

Consider a stochastic problem of optimal decision making over two stages: In the first stage we will choose a finite-dimensional vector $u_0$ from a subset $U_0$ with cost $g_0(u_0)$. Then an uncertain event represented by a random variable $w_0$ will occur, whereby $w_0$ will take one of the values $w_1^i, \ldots, w_m^i$ with corresponding probabilities $p_1^i, \ldots, p_m^i$. Once $w_0$ occurs, we will know its value $w_i$, and we must then choose at the second stage a vector $u_1^i$ from a subset $U_1(u_0, w_i)$ at a cost $g_1(u_1^i, w_i)$. The objective is to minimize the expected cost

$$g_0(u_0) + \sum_{i=1}^m p_i^i g_1(u_1^i, w_i),$$

subject to

$$u_0 \in U_0, \quad u_1^i \in U_1(u_0, w_i), \quad i = 1, \ldots, m.$$
probabilities \( p^1, \ldots, p^m \), the cost of the first stage is \( g_0(u_0) \), the cost of the second stage is \( g_1(x_1, u_1) \), and the terminal cost is 0. The intuitive meaning is that since at time 0 we don’t know yet which of the \( m \) values \( w^i \) of \( w_0 \) will occur, we must calculate (in addition to \( u_0 \)) a separate second stage decision \( u_1^i \) for each \( i \), which will be used after we know that the value of \( w_0 \) is \( w^i \).

However, if \( u_0 \) and \( u_1 \) take values in a continuous space such as the Euclidean spaces \( \mathbb{R}^{d_0} \) and \( \mathbb{R}^{d_1} \), respectively, we can also equivalently view the problem as a nonlinear programming problem of dimension \((d_0 + md_1)\) (the optimization variables are \( u_0 \) and \( u_1^i, i = 1, \ldots, m \)).

For a generalization of the preceding example, consider the stochastic DP problem of Section 1.3 for the case where there are only two stages, and the disturbances \( w_0 \) and \( w_1 \) can independently take one of the \( m \) values \( w^1, \ldots, w^m \) with corresponding probabilities \( p_0^1, \ldots, p_0^m \) and \( p_1^1, \ldots, p_1^m \), respectively. The optimal cost function \( J_0(x_0) \) is given by the two-stage DP algorithm

\[
J_0(x_0) = \min_{u_0 \in U_0(x_0)} \left[ \sum_{i=1}^{m} p_0^i \left\{ g_0(x_0, u_0, w^i) \right. \right. \\
+ \left. \left. \min_{u_1^i \in U_1(f_0(x_0, u_0, w^i))} \left[ \sum_{j=1}^{m} p_1^j \left\{ g_1 \left( f_0(x_0, u_0, w^j), u_1^i, w^j \right) \right. \right. \right. \\
+ \left. \left. g_2 \left( f_1 \left( f_0(x_0, u_0, w^i), u_1^i, w^j \right) \right) \right\} \right\} \right]\].
\]

By bringing the inner minimization outside the inner brackets, we see that this DP algorithm is equivalent to solving the nonlinear programming problem

\[
\text{minimize} \quad \sum_{i=1}^{m} p_0^i \left\{ g_0(x_0, u_0, w^i) \right. \\
+ \left. \sum_{j=1}^{m} p_1^j \left\{ g_1 \left( f_0(x_0, u_0, w^j), u_1^i, w^j \right) \right. \right. \\
+ \left. \left. g_2 \left( f_1 \left( f_0(x_0, u_0, w^i), u_1^i, w^j \right) \right) \right\} \right\} \}
\]

subject to \( u_0 \in U_0(x_0), \quad u_1^i \in U_1(f_0(x_0, u_0, w^i)), \quad i = 1, \ldots, m. \)

If the controls \( u_0 \) and \( u_1^i \) are elements of \( \mathbb{R}^d \), this problem involves

\[
d(1 + m)
\]

scalar variables.

We can also consider an \( N \)-stage stochastic optimal control problem. A similar reformulation as a nonlinear programming problem is possible. It converts the \( N \)-stage stochastic problem into a deterministic optimization
problem of dimension that grows exponentially with the number of stages \(N\). In particular, for an \(N\)-stage problem, the number of control variables expands by a factor \(m\) with each additional stage. The total number of variables is bounded by

\[
d(1 + m + m^2 + \cdots + m^{N-1}),
\]

where \(m\) is the maximum number of values that a disturbance can take at each stage and \(d\) is the dimension of the control vector. An example is the multi-item production problem described in Example 2.8.2 in the case where the demands \(w_k^i\) and/or the production coefficients \(a_k^{ij}\) are stochastic.

### 2.8.3 Stochastic Rollout with Certainty Equivalence

The dimension of the preceding nonlinear programming formulation of the multistage stochastic optimal control problem with continuous control spaces can be very large. This motivates a variant of a rollout algorithm that relies on a stochastic optimization for the current stage, and a deterministic optimization that relies on (assumed) certainty equivalence for the remaining stages, where the base policy is used. In this way, the dimension of the nonlinear programming problem to be solved by rollout is drastically reduced.

This rollout algorithm operates as follows: Given a state \(x_k\) and control \(u_k \in U_k(x_k)\), we consider the next states \(x_{k+1}^i\) that correspond to the \(m\) possible values \(w_k^i, i = 1, \ldots, m\), which occur with the known probabilities \(p_k^i, i = 1, \ldots, m\). We then consider the approximate Q-factors

\[
\tilde{Q}_k(x_k, u_k) = \sum_{i=1}^{m} p_k^i \left( g_k(x_k, u_k, w_k^i) + \tilde{H}_{k+1}(x_{k+1}^i) \right), \tag{2.70}
\]

where \(\tilde{H}_{k+1}(x_{k+1}^i)\) is the cost of a base policy, which starting at stage \(k + 1\) from

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

optimizes the cost-to-go starting from \(x_{k+1}^i\), while assuming that the future disturbances \(w_{k+1}, \ldots, w_{N-1}\), will take some nominal (nonrandom) values \(\bar{w}_{k+1}, \ldots, \bar{w}_{N-1}\). The rollout control \(\tilde{\mu}_k(x_k)\) computed by this algorithm is

\[
\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k). \tag{2.71}
\]

Note that this rollout algorithm does not have the cost improvement property, because it involves an approximation: the cost \(\tilde{H}_{k+1}(x_{k+1}^i)\) used in Eq. (2.70) is an approximation to the cost of a policy. It is the cost of a policy applied to the certainty equivalent version of the original stochastic problem.
The key fact now is that the problem (2.71) can be viewed as a seamless \((N - k)\)-stage deterministic optimization, which involves the control \(u_0\), and for each value \(w^k\) of the disturbance \(w_k\), the sequence of controls \((u^k_{k+1}, \ldots, u^k_{N-1})\). If the controls are elements of \(\mathbb{R}^d\), this deterministic optimization involves a total of

\[
d(1 + (N - k - 1)m)
\]

scalar variables. Currently available deterministic optimization software can deal with quite large numbers of variables, particularly in the context of linear programming, so by using rollout in combination with certainty equivalence, very large problems with continuous state and control variables may be addressed.

Another possibility is to use multistep lookahead that aims to represent better the stochastic character of the uncertainty. Here at state \(x_k\) we solve an \((N - k)\)-stage optimal control problem, where the uncertainty is fully taken into account in the first \(\ell\) stages, similar to stochastic programming, and in the remaining \(N - k - \ell\) stages, the uncertainty is dealt with by certainty equivalence, by fixing the disturbances \(w_{k+\ell}, \ldots, w_{N-1}\) at some nominal values (we assume here for simplicity that \(\ell < N - k\)). If the controls are elements of \(\mathbb{R}^d\), and the number of values that the disturbances \(w_0, \ldots, w_{N-1}\) can take is \(m\), the total number of control variables of this problem is

\[
d(1 + m + \cdots + m^{\ell-1} + (N - k - \ell)m^{\ell}),
\]

[this is the \(\ell\)-step lookahead generalization of the formula (2.72)]. Once the optimal policy \(\{\hat{u}_k, \hat{\mu}_{k+1}, \hat{\mu}_{k+2}, \ldots\}\) for this problem is obtained, the first control component \(\hat{u}_k\) is applied at \(x_k\) and the remaining components \(\{\hat{\mu}_{k+1}, \hat{\mu}_{k+2}, \ldots\}\) are discarded. Note also that this multistep lookahead approach may be combined with the ideas of multiagent rollout, which will be discussed in the next section.

### 2.9 MULTIAGENT ROLLOUT

We will now consider a special structure of the control space, whereby the control \(u_k\) consists of \(m\) components, \(u_k = (u^1_k, \ldots, u^m_k)\), with a separable control constraint structure \(u^\ell_k \in U^\ell_k(x_k), \ell = 1, \ldots, m\). The control constraint set is the Cartesian product

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

Conceptually, each component \(u^\ell_k, \ell = 1, \ldots, m\), is chosen at stage \(k\) by a separate "agent" (a decision making entity), and for the sake of the
following discussion, we assume that each set $U^\ell_k(x_k)$ is finite. We discussed this type of problem briefly in Section 1.6.5, and we will discuss it in this section in greater detail.

Thus the one-step lookahead minimization

$$\tilde{u}_k \in \arg\min_{u_k \in U_k(x_k)} E\left\{ g_k(x_k, u_k, w_k) + J_{k+1, \pi}(f_k(x_k, u_k, w_k)) \right\}, \quad (2.74)$$

where $\pi$ is a base policy, 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 the Cartesian product structure (2.73)]. As a result, the standard rollout algorithm requires an exponential [order $O(n^m)$] number of base policy cost computations per stage, which can be overwhelming even for moderate values of $m$.

This motivates an alternative and more efficient rollout algorithm, called \textit{multiagent rollout} also referred to as \textit{agent-by-agent rollout}, that still achieves the cost improvement property

$$J_{k, \tilde{\pi}}(x_k) \leq J_{k, \pi}(x_k), \quad \forall x_k, k, \quad (2.75)$$

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

A key idea here is that the computational requirements of the rollout one-step minimization (2.74) are proportional to the size of the control space and are independent of the size of the state space. We consequently reformulate the problem so that control space complexity is traded off with state space complexity, as discussed in Section 1.6.5. This is done by “unfolding” the control $u_k$ into its $m$ components $u^1_k, u^2_k, \ldots, u^m_k$. At the same time, between $x_k$ and the next state $x_{k+1} = f_k(x_k, u_k, w_k)$, we introduce artificial intermediate “states” and corresponding transitions; see Fig. 2.9.1.

It can be seen that this reformulated problem is equivalent to the original, since any control choice that is possible in one problem is also possible in the other problem, while the cost structure of the two problems is the same. In particular, each policy of the reformulated problem corresponds to a policy of the original problem, with the same cost function, and reversely.†

† Policies of the original problem involve functions of $x_k$, while policies of the reformulated problem involve functions of the choices of the preceding agents, as well as $x_k$. However, by successive substitution of the control functions of the preceding agents, we can view control functions of each agent as depending exclusively on $x_k$. It follows that the multi-transition structure of the reformulated problem cannot be exploited to reduce the cost function beyond what can be achieved with a single-transition structure.
controls trajectories up to stage \( N \) for a total of order \( O \). We average the costs of these trajectories, thereby obtaining the 

Each of the “states” sets \( U_k \). 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 \( U^\ell_k(x_k) \), the computation required at each stage \( k \) is of order \( O(n) \) for each of the “states”

\[
 (x_k, u^1_k), (x_k, u^2_k), \ldots, (x_k, u^{m-1}_k)
\]

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

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

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

(a) All agents have access to the current state \(x_k\) as well as the base policy (including the control functions \(\mu_n^\ell, \ell = 1, \ldots, m, n = 0, \ldots, N-1\) of all agents).

(b) There is an order in which agents compute and apply their local controls.

(c) The agents share their information, so agent \(\ell\) 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, while still maintaining the cost improvement property (2.75).

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

**Example 2.9.1 (Spiders and Flies)**

We have two spiders and two flies moving along integer locations on a straight line. For simplicity we assume that the flies’ positions are fixed at some integer locations, although the problem is qualitatively similar when the flies move randomly. The spiders have the option of moving either left or right by one unit; see Fig. 2.9.2. The objective is to minimize the time to capture both flies. The problem has essentially a finite horizon since the spiders can force the capture of the flies within a known number of steps.

The salient feature of the optimal policy here is to move the two spiders towards different flies. The minimal time to capture is the maximum of the initial distances of the two spider-fly pairs of the optimal policy.

Let us apply multiagent rollout with the base policy that directs each spider to move one unit towards the closest fly position (a tie is broken by moving towards the right-side fly). The base policy is poor because it may unnecessarily move both spiders in the same direction, when in fact only one is needed to capture the fly. This limitation is due to the lack of coordination between the spiders: each acts selfishly, ignoring the presence of the other. We will see that rollout restores a significant degree of coordination between
Multiagent rollout with the given base policy starts with spider 1 at location $n$, and calculates the two Q-factors of moving to locations $n-1$ and $n+1$, assuming that the remaining moves of the two spiders will be made using the go-towards-the-nearest-fly base policy. The Q-factor of going to $n-1$ is smallest because it saves in unnecessary moves of spider 1 towards fly 2, so spider 1 will move towards fly 1. The trajectory generated by multiagent rollout is to move spiders 1 and 2 towards flies 1 and 2, respectively, then spider 2 first captures fly 2, and then spider 1 captures fly 1.

Figure 2.9.2 Illustration of the two-spiders and two-flies problem. The spiders move along integer points of a line. The two flies stay still at some integer locations. The character of the optimal policy is to move the two spiders towards two different flies.

According to the multiagent rollout mechanism, the spiders choose their moves one-at-a-time, optimizing over the two Q-factors corresponding to the right and left moves, while assuming that future moves will be chosen according to the base policy. Let us consider a stage, where the two flies are alive, while both spiders are closest to fly 2, as in Fig. 2.9.2. Then the rollout algorithm will start with spider 1 and calculate two Q-factors corresponding to the right and left moves, while using the base heuristic to obtain the next move of spider 2, and the remaining moves of the two spiders. Depending on the values of the two Q-factors, spider 1 will move to the right or to the left, and it can be seen that it will choose to move away from spider 2 even if doing so increases its distance to its closest fly contrary to what the base heuristic will do. Then spider 2 will act similarly and the process will continue. Intuitively, at the state of Fig. 2.9.2, spider 1 moves away from spider 2 and fly 2, because it recognizes that spider 2 will capture earlier fly 2, so it might as well move towards the other fly.

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

The example illustrates how a poor base heuristic can produce an excellent rollout solution, something that can be observed frequently in many other problems. Intuitively, the key fact is that rollout is “farsighted” in the sense that it can benefit from control calculations that reach far into future stages.

A two-dimensional generalization of the example is also interesting. Here the flies are at two corners of a square in the plane. It can be shown that the two spiders, starting from the same position within the square, will
separate under the rollout policy, with each moving towards a different spider, while under the base policy, they will move in unison along the shortest path to the closest surviving fly. Again this will happen for both standard and multiagent rollout.

Example 2.9.2 (Multi-Vehicle Routing)

Consider the multi-vehicle routing problem of Example 1.2.3, whereby $m$ vehicles move along the arcs of a given graph, aiming to perform tasks located at the nodes of the graph; cf. Fig. 2.9.3.

For a large number of vehicles and a complicated graph, this is a non-trivial combinatorial problem. As we discussed in Example 1.2.3, the problem can be formulated as a discrete deterministic optimization problem, and addressed by approximate DP methods. The state at a given stage is the $m$-tuple of current positions of the vehicles together with the list of pending tasks, but the number of these states can be enormous (it increases exponentially with the number of nodes and the number of vehicles). Moreover the number of joint move choices by the vehicles also increases exponentially with the number of vehicles.

We are thus motivated to use a multiagent rollout approach. We define a base heuristic as follows: at a given stage and state (vehicle positions and pending tasks), it finds the closest pending task (in terms of number of moves needed to reach it) for each of the vehicles and moves each vehicle one step towards the corresponding closest pending task (this is a legitimate base heuristic: it assigns to each state a vehicle move for every vehicle).†

In the multiagent rollout algorithm, at a given stage and state, we take up each vehicle in the order 1, . . . , $n$, and we compare the Q-factors of the available moves to that vehicle while assuming that all the remaining moves will be made according to the base heuristic, and taking into account the moves that have been already made and the tasks that have already been performed; see the illustration of Fig. 2.9.3. In contrast to all-vehicles-at-once rollout, the one-vehicle-at-a-time rollout algorithm considers a polynomial (in $m$) number of moves and corresponding shortest path problems at each stage. In the example of Fig. 2.9.3, the one-vehicle-at-a-time rollout finds the optimal solution, while the base heuristic starting from the initial state does not.

† There is an alternative version of the base heuristic, which makes selections one-vehicle-at-a-time: at a given stage and state (vehicle positions and pending tasks), it finds the closest pending task (in terms of number of moves needed to reach it) for vehicle 1 and moves this vehicle one step towards this closest pending task. Then it finds the closest pending task for vehicle 2 (the pending status of the tasks, however, may have been affected by the move of vehicle 1) and moves this vehicle one step towards this closest pending task, and continues similarly for vehicles 3, . . . , $n$. There is a subtle difference between the two base heuristics: for example they may make different choices when vehicle 1 reaches a pending task in a single move, thereby changing the status of that task, and affecting the choice of the base heuristic for vehicle 2, etc.
Move each vehicle one step at a time towards its nearest pending task, until all tasks are performed.

**Figure 2.9.3** An instance of the vehicle routing problem of Example 2.9.2, and the multiagent rollout approach. The two vehicles aim to collectively perform the two tasks as fast as possible. Here, we should avoid sending both vehicles to node 4, towards the task at node 7; sending only vehicle 2 towards that task, while sending vehicle 1 towards the task at node 9 is clearly optimal. However, the base heuristic has “limited vision” and does not perceive this. By contrast the standard and the one-vehicle-at-a-time rollout algorithms look beyond the first move and avoid this inefficiency: they examine both moves of vehicle 1 to nodes 3 and 4, and use the base heuristic to explore the corresponding trajectories to the end of the horizon, and discover that vehicle 2 can reach quickly node 7, and that it is best to send vehicle 1 towards node 9.

In particular, the one-vehicle-at-a-time rollout algorithm will operate as follows: given the starting position pair (1,2) of the vehicles and the current pending tasks at nodes 7 and 9, we first compare the Q-factors of the two possible moves of vehicle 1 (to nodes 3 and 4), assuming that all the remaining moves will be selected by the base heuristic at the beginning of each stage. Thus vehicle 1 will choose to move to node 3. Then with knowledge of the move of vehicle 1 from 1 to 3, we select the move of vehicle 2 by comparing the Q-factors of its two possible moves (to nodes 4 and 5), taking also into account the fact that the remaining moves will be made according to the base heuristic. Thus vehicle 2 will choose to move to node 4.

We then continue at the next state [vehicle positions at (3,4) and pending tasks at nodes 7 and 9], select the base heuristic moves of vehicles 1 and 2 on the path to the closest pending tasks [(9 and 7), respectively], etc. Eventually the rollout finds the optimal solution (move vehicle 1 to node 9 in three moves and move vehicle 2 to node 7 in two moves), which has a total cost of 5. By contrast it can be seen that the base heuristic at the initial state will move both vehicles to node 4 (towards the closest pending task), and generate a trajectory that moves vehicle 1 along the path 1 → 4 → 7 and vehicle 2 along the path 2 → 4 → 7 → 10 → 12 → 9, while incurring a total cost of 7.
The Cost Improvement Property

Generally, it is unclear how the two rollout policies (standard/all-agents-at-once and agent-by-agent) perform relative to each other in terms of attained cost.† On the other hand, both rollout policies perform no worse than the base policy, since the performance of the base policy is identical for both the reformulated and the original problems. This cost improvement property can also be shown analytically as follows by induction, by modifying the standard rollout cost improvement proof; cf. Section 2.7.

Proposition 2.9.1: (Cost Improvement for Multiagent Rollout) The rollout policy \( \tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\} \) obtained by multiagent rollout satisfies

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

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

Proof: We will show the inequality (2.76) by induction, but for simplicity, we will give the proof for the case of just two agents, i.e., \( m = 2 \). Clearly the inequality holds for \( k = N \), since \( J_{N,\tilde{\pi}} = J_{N,\pi} = g_N \). Assuming that it holds for index \( k + 1 \), we have for all \( x_k \),

\[
J_{k,\tilde{\pi}}(x_k) = E \left\{ g_k(x_k, \tilde{\mu}_1^k(x_k), \tilde{\mu}_2^k(x_k), w_k) \\
+ J_{k+1,\tilde{\pi}} \left( f_k(x_k, \tilde{\mu}_1^k(x_k), \tilde{\mu}_2^k(x_k), w_k) \right) \right\}
\]

† For an example where the standard rollout algorithm works better, consider a single-stage problem, where the objective is to minimize the first stage cost \( g_0(u_0, \ldots, u_m) \). Let \( \pi_0 = (\pi_0^0, \ldots, \pi_0^m) \) 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 multiagent 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 no worse than the multiagent rollout algorithm.

The example just given is best seen within the framework of the classical coordinate descent method for minimizing a function of \( m \) components. This method can get stuck at a nonoptimal point in the absence of appropriate conditions on the cost function, such as differentiability and/or convexity. However, within our context of multistage rollout and possibly stochastic disturbances, it appears that the consequences of such a phenomenon may not be serious. In fact, one can construct multi-stage examples where multiagent rollout performs better than the standard rollout.
\[
\begin{align*}
\leq E\left\{ g_k(x_k, \tilde{\mu}_1^k(x_k), \tilde{\mu}_2^k(x_k), w_k) \\
+ J_{k+1, \pi}\left( f_k(x_k, \tilde{\mu}_1^k(x_k), \tilde{\mu}_2^k(x_k), w_k) \right) \right\}
= \min_{u_2^k \in U_2^k(x_k)} E\left\{ g_k(x_k, \tilde{\mu}_1^k(x_k), w_k) \\
+ J_{k+1, \pi}\left( f_k(x_k, \tilde{\mu}_1^k(x_k), w_k) \right) \right\}
\leq E\left\{ g_k(x_k, \tilde{\mu}_1^k(x_k), \tilde{\mu}_2^k(x_k), w_k) \\
+ J_{k+1, \pi}\left( f_k(x_k, \mu_2^k(x_k), w_k) \right) \right\}
= \min_{u_1^k \in U_1^k(x_k)} E\left\{ g_k(x_k, u_1^k, \mu_2^k(x_k), w_k) \\
+ J_{k+1, \pi}\left( f_k(x_k, u_1^k, \mu_2^k(x_k), w_k) \right) \right\}
\leq E\left\{ g_k(x_k, \mu_1^k(x_k), \mu_2^k(x_k), w_k) \\
+ J_{k+1, \pi}\left( f_k(x_k, \mu_1^k(x_k), \mu_2^k(x_k), w_k) \right) \right\}
= J_{k, \pi}(x_k),
\end{align*}
\]

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

**Optimizing the Agent Order in Agent-by-Agent Rollout - Multiagent Parallelization**

In the multiagent rollout algorithm described so far, the agents optimize the control components sequentially in a fixed order. It is possible to improve performance by trying to optimize at each stage \(k\) the order of the agents.

An efficient way to do this is to first optimize over all single agent Q-factors, by solving the \(m\) minimization problems that correspond to each of the agents \(\ell = 1, \ldots, m\) being first in the multiagent rollout order. If \(\ell_1\) is
Sec. 2.9 Multiagent Rollout

the agent that produces the minimal Q-factor, we fix $\ell_1$ to be the first agent in the multiagent rollout order. Then we optimize over all single agent Q-factors, by solving the $m - 1$ minimization problems that correspond to each of the agents $\ell \neq \ell_1$ being second in the multiagent rollout order. Let $\ell_2$ be the agent that produces the minimal Q-factor, fix $\ell_2$ to be the second agent in the multiagent rollout order, and continue in this manner. In the end, after

$$m + (m - 1) + \cdots + 1 = \frac{m(m + 1)}{2}$$

minimizations, we obtain an agent order $\ell_1, \ldots, \ell_m$ that produces a potentially much reduced Q-factor value, as well as the corresponding rollout control component selections.

The method just described likely produces substantially better performance, and eliminates the need for guessing a good agent order, but it increases the number of Q-factor calculations needed per stage roughly by a factor $(m + 1)/2$. Still this is much better than the all-agents-at-once approach, which requires an exponential number of Q-factor calculations. Moreover, the Q-factor minimizations of the above process can be parallelized, so with $m$ parallel processors, we can perform the number of $m(m + 1)/2$ minimizations derived above in just $m$ batches of parallel minimizations, which require about the same time as in the case where the agents are selected for Q-factor minimization in a fixed order. We finally note that our earlier cost improvement proof goes through again by induction, when the order of agent selection is variable at each stage $k$.

Multiagent Rollout Variants

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

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

(b) When the control constraint sets $U_k^\ell(x_k)$ are infinite, multiagent rollout still applies, based on the tradeoff between control and state space complexity, cf. Fig. 2.9.1. In particular, when the sets $U_k^\ell(x_k)$ are intervals of the real line, each agent’s lookahead minimization problem can be performed with the aid of one-dimensional search methods.

(c) When the problem is deterministic there are additional possible variants of the multiagent rollout algorithm. In particular, for deterministic problems, we may use a more general base policy, i.e., a heuristic that is not defined by an underlying policy; cf. Section 2.3.1. In this case, if the sequential improvement assumption for the modified problem of Fig. 2.9.1 is not satisfied, then the cost improvement property
may not hold. However, cost improvement may be restored by introducing fortification, as discussed in Section 2.3.2.

(d) The multiagent rollout algorithm can be simply modified to apply to infinite horizon problems. In this context, we may also consider policy iteration methods, which can be viewed as repeated rollout. These methods may involve agent-by-agent policy improvement, and value and policy approximations of intermediately generated policies (see the RL book [Ber19a], Section 5.7.3).

(e) The multiagent rollout algorithm can be simply modified to apply to deterministic continuous-time optimal control problems; cf. Section 2.6. The idea is again to simplify the minimization over $u(t)$ in the case where $u(t)$ consists of multiple components $u^1(t), \ldots, u^m(t)$.

(f) We can implement within the agent-by-agent rollout context the use of Q-factor differences. The motivation is similar: deal with the approximation errors that are inherent in the estimated cost of the base policy $\hat{J}_{k+1,\pi}(f_k(x_k, u_k))$, and may overwhelm the current stage cost term $g_k(x_k, u_k)$. As noted in Section 2.3.7, this may seriously degrade the quality of the rollout policy; see also the discussion of advantage updating and differential training in Chapter 3.

**Constrained Multiagent Rollout**

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

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

We refer to this as the *multiagent case*, motivated by the special case where each component $u^\ell_k$, $\ell = 1, \ldots, m$, is chosen by a separate agent $\ell$ at stage $k$.

Similar to the unconstrained case, we can introduce a modified but equivalent problem, involving one-at-a-time agent control selection. In particular, at the generic state $x_k$, we break down the control $u_k$ into the sequence of the $m$ controls $u^1_k, u^2_k, \ldots, u^m_k$, and between $x_k$ and the next state $x_{k+1} = f_k(x_k, u_k)$, we introduce artificial intermediate “states”

$$(x_k, u^1_k), (x_k, u^1_k, u^2_k), \ldots, (x_k, u^1_k, \ldots, u^{m-1}_k),$$

and corresponding transitions. The choice of the last control component $u^m_k$ at “state” $(x_k, u^1_k, \ldots, u^{m-1}_k)$ marks the transition at cost $g_k(x_k, u_k)$ to the next state $x_{k+1} = f_k(x_k, u_k)$ according to the system equation. It is evident that this reformulated problem is equivalent to the original, since
any control choice that is possible in one problem is also possible in the
other problem, with the same cost.

By working with the reformulated problem, we can consider a rollout
algorithm that requires a sequence of \( m \) minimizations per stage, one over
each of the control components \( u^1_k, \ldots, u^m_k \), with the past controls already
determined by the rollout algorithm, and the future controls determined
by running the base heuristic. Assuming a maximum of \( n \) elements in the
control component spaces \( U^\ell_k(x_k), \ell = 1, \ldots, m \), the computation required
for the \( m \) single control component minimizations is of order \( O(nm) \) per
stage. By contrast the standard rollout minimization (2.34) involves the
computation and comparison of as many as \( n^m \) terms \( G(T_k(\tilde{y}_k, u_k)) \) per
stage.

### 2.9.1 Asynchronous and Autonomous Multiagent Rollout

In this section we consider multiagent rollout algorithms that are dis-
tributed and asynchronous in the sense that the agents may compute their
rollout controls in parallel rather than in sequence, aiming at computa-
tional speedup. 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.

This algorithm may work well for some problems, but it does not
possess the cost improvement property, and may not work well for other
problems. In fact we can construct a simple example involving a single
state, two agents, and two controls per agent, where the second agent does
not take into account the control applied by the first agent, and as a result
the rollout policy performs worse than the base policy for some initial
states.

#### Example 2.9.3 (Cost Deterioration in the Absence of
Adequate Agent Coordination)

Consider a problem with two agents (\( m = 2 \)) and a single state. Thus the state
does not change and the costs of different stages are decoupled (the problem
is essentially static). Each of the two agents has two controls: \( u^1_k \in \{0, 1\} \)
and \( u^2_k \in \{0, 1\} \). The cost per stage \( g_k \) is equal to 0 if \( u^1_k \neq u^2_k \), is equal to
1 if \( u^1_k = u^2_k = 0 \), and is equal to 2 if \( u^1_k = u^2_k = 1 \). Suppose that the base
policy applies \( u^1_k = u^2_k = 0 \). Then it can be seen that when executing rollout,
the first agent applies \( u^1_k = 1 \), and in the absence of knowledge of this choice,
the second agent also applies \( u^2_k = 1 \) (thinking that the first agent will use
the base policy control \( u^1_k = 0 \)). Thus the cost of the rollout policy is 2 per
stage, while the cost of the base policy is 1 per stage. By contrast the rollout
algorithm that takes into account the first agent’s control when selecting the
second agent’s control applies \( u^1_k = 1 \) and \( u^2_k = 0 \), thus resulting in a rollout
policy with the optimal cost of 0 per stage.
The difficulty here is inadequate coordination between the two agents. In particular, each agent uses rollout to compute the local control, each thinking that the other will use the base policy control. If instead the two agents were to coordinate their control choices, they would have applied an optimal policy.

The simplicity of the preceding example raises serious questions as to whether the cost improvement property (2.76) can be easily maintained by a distributed rollout algorithm where the agents do not know the controls applied by the preceding agents in the given order of local control selection, and use instead the controls of the base policy. One may speculate that if the agents are naturally “weakly coupled” in the sense that their choice of control has little impact on the desirability of various controls of other agents, then a more flexible inter-agent communication pattern may be sufficient for cost improvement.†

An important question is to clarify the extent to which agent coordination is essential. In what follows in this section, we will discuss a distributed asynchronous multiagent rollout scheme, which is based on the use of a signaling policy that provides estimates of coordinating information once the current state is known.

**Autonomous Multiagent Rollout - Signaling Policies**

An interesting possibility for autonomous control selection by the agents is to use a distributed rollout algorithm, which is augmented by a precomputed signaling policy that embodies agent coordination.‡ The idea is to assume that the agents do not communicate their computed rollout control components to the subsequent agents in the given order of local control selection. Instead, once the agents know the state, they use precomputed (or easily computed) approximations to the control components of the preceding agents, and compute their own control components in parallel and asyn-

† In particular, one may divide the agents in “coupled” groups, and require coordination of control selection only within each group, while the computation of different groups may proceed in parallel. Note that the “coupled” group formations may change over time, depending on the current state. For example, in applications where the agents’ locations are distributed within some geographical area, it may make sense to form agent groups on the basis of geographic proximity, i.e., one may require that agents that are geographically near each other (and hence are more coupled) coordinate their control selections, while agents that are geographically far apart (and hence are less coupled) forego any coordination.

‡ The general idea of coordination by sharing information about the agents’ policies arises also in other multiagent algorithmic contexts, including some that involve forms of policy gradient methods and Q-learning; see the surveys of the relevant research cited earlier. The survey by Matignon, Laurent, and Le Fort-Piat [MLL12] focuses on coordination problems from an RL point of view.
Sec. 2.9 Multiagent Rollout

chronously. We call this algorithm autonomous multiagent rollout. While this type of algorithm involves a form of redundant computation, it allows for additional speedup through parallelization.

The algorithm at the $k$th stage uses a base policy $\mu_k = \{\mu_k^1, \ldots, \mu_k^{m-1}\}$, but also uses a second policy $\tilde{\mu}_k = \{\tilde{\mu}_k^1, \ldots, \tilde{\mu}_k^{m-1}\}$, called the signaling policy, which is computed off-line, is known to all the agents for on-line use, and is designed to play an agent coordination role. Intuitively, $\tilde{\mu}_k^m(x_k)$ provides an intelligent “guess” about what agent $\ell$ will do at state $x_k$. This is used in turn by all other agents $i \neq \ell$ to compute asynchronously their own rollout control components on-line.

More precisely, the autonomous multiagent rollout algorithm uses the base and signaling policies to generate a rollout policy $\tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$ as follows. At stage $k$ and state $x_k$, $\tilde{\mu}_k(x_k) = (\tilde{\mu}_k^1(x_k), \ldots, \tilde{\mu}_k^m(x_k))$, is obtained according to

$$\begin{align*}
\tilde{\mu}_k^1(x_k) &\in \arg \min_{u_k^1 \in U_k^1(x_k)} E \left\{ g_k(x_k, u_k^1, \mu_k^2(x_k), \ldots, \mu_k^m(x_k), w_k) \\
&\quad + J_{k+1,\pi} \left( f_k(x_k, u_k^1, \mu_k^2(x_k), \ldots, \mu_k^m(x_k), w_k) \right) \right\}, \\
\tilde{\mu}_k^2(x_k) &\in \arg \min_{u_k^2 \in U_k^2(x_k)} E \left\{ g_k(x_k, \tilde{\mu}_k^1(x_k), u_k^2, \ldots, \mu_k^m(x_k), w_k) \\
&\quad + J_{k+1,\pi} \left( f_k(x_k, \tilde{\mu}_k^1(x_k), u_k^2, \ldots, \mu_k^m(x_k), w_k) \right) \right\}, \\
&\quad \cdots \\
\tilde{\mu}_k^m(x_k) &\in \arg \min_{u_k^m \in U_k^m(x_k)} E \left\{ g_k(x_k, \tilde{\mu}_k^{m-1}(x_k), u_k^m, w_k) \\
&\quad + J_{k+1,\pi} \left( f_k(x_k, \tilde{\mu}_k^{m-1}(x_k), u_k^m, w_k) \right) \right\}.
\end{align*}$$

Note that the preceding computation of the controls $\tilde{\mu}_k^1(x_k), \ldots, \tilde{\mu}_k^m(x_k)$ can be done asynchronously and in parallel, and without direct agent coordination, since the signaling policy values $\tilde{\mu}_k^1(x_k), \ldots, \tilde{\mu}_k^{m-1}(x_k)$ are precomputed and are known to all the agents.

The simplest choice is to use as signaling policy $\tilde{\mu}$ the base policy $\mu$. However, this choice does not guarantee policy improvement as evidenced by Example 2.9.3. In fact performance deterioration with this choice is not uncommon, and can be observed in more complicated examples, including the following.

Example 2.9.4 (Spiders and Flies - Use of the Base Policy for Signaling)

Consider the problem of Example 2.9.1, which involves two spiders and two flies on a line, and the base policy $\mu$ that moves a spider towards the closest surviving fly (and in case where a spider starts at the midpoint between
the two flies, moves the spider to the right). Assume that we use as signaling policy \( \hat{\mu} \) the base policy \( \mu \). It can then be verified that if the spiders start from different positions, the rollout policy will be optimal (will move the spiders in opposite directions). If, however, the spiders start from the same position, a completely symmetric situation is created, whereby the rollout controls move both flies in the direction of the fly furthest away from the spiders’ position (or to the left in the case where the spiders start at the midpoint between the two flies). Thus, the flies end up oscillating around the middle of the interval between the flies and never catch the flies!

The preceding example is representative of a broad class of counterexamples that involve multiple identical agents. If the agents start at the same initial state, with a base policy that has identical components, and use the base policy for signaling, the agents will select identical controls under the corresponding multiagent rollout policy, ending up with a potentially serious cost deterioration.

This example also highlights an effect of the sequential choice of the control components \( u_{1k}, \ldots, u_{mk} \), based on the reformulated problem of Fig. 2.9.1: it tends to break symmetries and “group think” that guides the agents towards selecting the same controls under identical conditions. Generally, any sensible multiagent policy must be able to deal in some way with this “group think” issue. One simple possibility is for each agent \( \ell \) to randomize somehow the control choices of other agents \( j \neq \ell \) when choosing its own control, particularly in “tightly coupled” cases where the choice of agent \( \ell \) is “strongly” affected by the choices of the agents \( j \neq \ell \).

An alternative idea is to choose the signaling policy \( \hat{\mu}_k \) to approximate the sequential multiagent rollout policy (the one computed with each agent knowing the controls applied by the preceding agents), or some other policy that is known to embody coordination between the agents. In particular, we may obtain \( \hat{\mu}_k \) as the multiagent rollout policy for a related but simpler problem, such as a certainty equivalent version of the original problem, whereby the stochastic system is replaced by a deterministic one.

Another interesting possibility is to compute \( \hat{\mu}_k = (\hat{\mu}_k^1, \ldots, \hat{\mu}_k^m) \) by off-line training of a neural network (or \( m \) networks, one per agent) with training samples generated through the sequential multiagent rollout policy. We intuitively expect that if the neural network provides a signaling policy that approximates well the sequential multiagent rollout policy, we would obtain better performance than the base policy. This expectation was confirmed in a case study involving a large-scale multi-robot repair application (see [BKB20]).

The advantage of autonomous multiagent rollout with neural network or other type of approximations is that it may lead to approximate policy improvement, while at the same time allowing asynchronous distributed agent operation without on-line agent coordination through communication of their rollout control values (but still assuming knowledge of the exact
2.10 ROLLOUT FOR BAYESIAN OPTIMIZATION AND SEQUENTIAL ESTIMATION

In this section, we discuss a wide class of problems that has been studied intensively in statistics and related fields since the 1940s. Roughly speaking, in these problems we use observations and sampling for the purpose of inference, but the number and the type of observations are not fixed in advance. Instead, the outcomes of the observations are sequentially evaluated on-line with a view towards stopping or modifying the observation process. This involves sequential decision making, thus bringing to bear exact and approximate DP. A central issue here is to estimate an $m$-dimensional random vector $\theta$, using optimal sequential selection of observations, which are based on feedback from preceding observations; see Fig. 2.10.1.

For a simple illustrative example, let us consider a hypothesis testing problem whereby we can make observations, at a cost $C$ each, relating to two hypotheses. Given a new observation, we can either accept one of the hypotheses or delay the decision for one more period, pay the cost $C$, and obtain a new observation. At issue is trading off the cost of observation with the higher probability of accepting the wrong hypothesis. As an example, in a quality control setting, the two hypotheses may be that a certain product meets or does not meet a certain level of quality, while the observations may consist of quantitative tests of the quality of the product.

Intuitively, one expects that once the conditional probability of one of the hypotheses, given the observations thus far, gets sufficiently close to 1, we should stop the observations. Indeed classical DP analyses bear this out; see e.g., the books by Chernoff [Che72], DeGroot [DeG70], Whittle [Whi82], and the references quoted therein. In particular, the simple version of the hypothesis testing problem just described admits a simple and elegant optimal solution, known as the sequential probability ratio test. On the other hand more complex versions of the problem, involving for example multiple hypotheses and/or multiple types of observations, are computationally intractable, thus necessitating the use of suboptimal approaches.

An important distinction in sequential estimation problems is whether the current choice of observation affects the cost and the availability of future observations. If this is so, the problem can often be viewed most fruitfully as a combined estimation and control problem, and is related to a type of adaptive control problem that we will discuss in the next section. As an example we will consider there sequential decoding, whereby we search for a hidden code word by using a sequence of queries, in the spirit of the Wordle puzzle and the family of Mastermind games [see, e.g., the Wikipedia page for “Mastermind (board game)”].
If the observation choices are “independent” and do not affect the cost or availability of future observations, the problem is substantially simplified. We will discuss problems of this type in the present section, starting with the case of surrogate and Bayesian optimization.

**Surrogate Optimization**

Surrogate optimization refers to a collection of methods, which address suboptimally a broad range of minimization problems, beyond the realm of DP. The idea is to minimize approximately a function that is given as a “black box.” By this we mean a function whose analytical expression is unknown, and whose values at any one point may be hard-to-compute, e.g., may require costly simulation or experimentation cost functions with “surrogates” that are easier to obtain.

Here we introduce a model of the cost function that is parametrized by a parameter \( \theta \); see Fig. 2.10.2. We observe sequentially the cost function at a few observation points, construct a model of the cost function (the surrogate) by estimating \( \theta \) based on the results of the observations, and minimize the surrogate to obtain a suboptimal solution. The question is how to select observation points sequentially, using feedback from previous observations. This selection process often embodies an *exploration-exploitation tradeoff*: Observing at points likely to have near-optimal value vs observing at points in relatively unexplored areas of the search space.

**Bayesian Optimization**

Bayesian optimization (BO) has been used widely for the approximate optimization of functions whose values at given points can only be obtained
Figure 2.10.2 Illustration of construction of a surrogate for a “black box” function \( f \) whose values are hard-to-compute. We replace \( f \) with a parametric model that involves a parameter \( \theta \) to be estimated by using observations at some points. The points are selected sequentially, using the results of earlier observations. Eventually, the observation process is stopped (often when an observation/computation budget limit is reached), and the final estimate of \( \theta \) is used to construct the surrogate to be minimized in place of \( f \).

through time-consuming calculation, simulation, or experimentation. A classical application from geostatistical interpolation, pioneered by the statisticians Matheron and Krige, was to identify locations of high gold distribution in South Africa based on samples from a few boreholes (the name “kriging” is often used to refer to this type of application; see the review by Kleijnen [Kle09]). As another example, BO has been used to select the hyperparameters of machine-learning models, including the architectural parameters of the deep neural network of AlphaZero; see [SHS17].

In this section, we will focus on a relatively simple BO formulation that can be viewed as the special case of surrogate optimization. In particular, we will discuss the case where the surrogate function is parametrized by the collection of its values at the points where it is defined.† See the references cited later in this section. Formally, we want to minimize a real-valued function \( f \), defined over a set of \( m \) points, which we denote by \( 1, \ldots, m \). These \( m \) points lie in some space, which we leave unspecified for the moment.‡ The values of the function are not readily available, but

† More complex forms of surrogates are obtained through linear combinations of some basis functions, with the parameter vector \( \theta \) consisting of the weights of the basis functions.

‡ We restrict the domain of definition of \( f \) to be the finite set \( \{1, \ldots, m\} \) in order to facilitate the implementation of the rollout algorithm to be discussed in what follows. However, in a more general and sometimes more convenient formulation, the domain of \( f \) can be an infinite set, such as a subset of a finite-dimensional Euclidean space.
Figure 2.10.3 Illustration of a function $f$ that we wish to estimate. The function is defined at the points $u = 1, 2, 3, 4$, and is represented by a vector $\theta = (\theta_1, \theta_2, \theta_3, \theta_4) \in \mathbb{R}^4$, in the sense that $f(u) = \theta_u$ for all $u$. The prior distribution of $\theta$ is given, and is used to construct the posterior distribution of $\theta$ given noisy observations $z_u = \theta_u + w_u$ at some of the points $u$.

We denote the value of $f$ at a point $u$ by $\theta_u$:

$$\theta_u = f(u), \quad \text{for all } u = 1, \ldots, m.$$  

Thus the $m$-dimensional vector $\theta = (\theta_1, \ldots, \theta_m)$ belongs to $\mathbb{R}^m$ and represents the function $f$. We assume that we obtain sequentially noisy observations of values $f(u) = \theta_u$ at suitably selected points $u$. These values are used to estimate the vector $\theta$ (i.e., the function $f$), and to ultimately minimize (approximately) $f$ over the $m$ points $u = 1, \ldots, m$. The essence of the problem is to select points for observation based on an exploration-exploitation tradeoff (exploring the potential of relatively unexplored candidate solutions and improving the estimate of promising candidate solutions). The fundamental idea of the BO methodology is that the function value changes relatively slowly, so that observing the function value at some point provides information about the function values at neighboring points. Thus a limited number of strategically chosen observations can provide reasonable approximation to the true cost function over a large portion of the search space.

For a mathematical formulation of a BO framework, we assume that at each of $N$ successive times $k = 1, \ldots, N$, we select a single point $u_k \in \{1, \ldots, m\}$, and observe the corresponding component $\theta_{u_k}$ of $\theta$ (i.e., the
function value at $u_k$) with some noise $w_{uk}$, i.e.,

$$z_{uk} = \theta_{uk} + w_{uk};$$

(2.79)

see Fig. 2.10.3. We view the observation points $u_1, \ldots, u_N$ as the optimization variables (or controls/actions in a DP/RL context), and consider policies for selecting $u_k$ with knowledge of the preceding observations $z_{u_1}, \ldots, z_{u_k-1}$ that have resulted from the selections $u_1, \ldots, u_{k-1}$. We assume that the noise random variables $w_u, u \in \{1, \ldots, m\}$ are independent and that their distributions are given. Moreover, we assume that $\theta$ has a given a priori distribution on the space of $m$-dimensional vectors $\mathbb{R}^m$, which we denote by $b_0$. The posterior distribution of $\theta$, given any subset of observations

$$\{z_{u_1}, \ldots, z_{u_k}\},$$

is denoted by $b_k$.

An important special case arises when $b_0$ and the distributions of $w_u, u \in \{1, \ldots, m\}$, are Gaussian. In this case $b_0$ is a multidimensional Gaussian distribution, defined by its mean (based on prior knowledge, or an equal value for all $u = 1, \ldots, m$ in case of absence of such knowledge) and its covariance matrix [implying greater correlation for pairs $(u, u')$ that are “close” to each other in some problem-specific sense, e.g., exponentially decreasing with the Euclidean distance between $u$ and $u'$]. A key consequence of this assumption is that the posterior distribution $b_k$ is multidimensional Gaussian, and can be calculated in closed form by using well-known formulas.

More generally, $b_k$ evolves according to an equation of the form

$$b_{k+1} = B_k(b_k, u_{k+1}, z_{uk+1}), \quad k = 0, \ldots, N - 1.$$  (2.80)

Thus given the set of observations up to time $k$, and the next choice $u_{k+1}$, resulting in an observation value $z_{uk+1}$, the function $B_k$ gives the formula for updating $b_k$ to $b_{k+1}$, and may be viewed as a recursive estimator of $b_k$. In the Gaussian case, the function $B_k$ can be written in closed form, using standard formulas for Gaussian random vector estimation. In other cases where no closed form expression is possible, $B_k$ can be implemented through simulation that computes (approximately) the new posterior $b_{k+1}$ using samples generated from the current posterior $b_k$.

At the end of the sequential estimation process, after the complete observation set

$$\{z_{u_1}, \ldots, z_{u_N}\}$$

has been obtained, we have the posterior distribution $b_N$ of $\theta$, which we can use to compute a surrogate of $f$. As an example we may use as surrogate the posterior mean $\hat{\theta} = (\hat{\theta}_1, \ldots, \hat{\theta}_m)$, and declare as minimizer of $f$ over $u$ the point $u^*$ with minimum posterior mean:

$$u^* \in \arg\min\{\hat{\theta}_u \mid u = 1, \ldots, m\};$$

113
After 10 Observations

**Figure 2.10.4** Illustration of the true cost function $f$, defined over an interval of the real line, and the posterior distribution $b_{10}$ after noise-free measurements at 10 points. The shaded area represents the interval of the mean plus/minus the standard deviation of the posterior $b_{10}$ at the points $u$. The mean of the finally obtained posterior, as a function of $u$, may be viewed as a surrogate cost function that can be minimized in place of $f$. Note that since the observations are assumed noise-free, the mean of the posterior is exact at the observation points.

There is a large literature relating to the surrogate and Bayesian optimization methodology and its applications, particularly for the Gaussian case. We refer to the books by Rasmussen and Williams [RaW06], Powell and Ryzhov [PoR12], the highly cited papers by Saks et al. [SWM89], Jones, Schonlau, and Welch [JSW98], and Queipo et al. [QHS05], the reviews by Sasena [Sas02], Powell and Frazier [PoF08], Forrester and Keane [FoK09], Kleijnen [Kle09], Brochu, Cora, and De Freitas [BCD10], Ryzhov, Powell, and Frazier [RPF12], Ghavamzadeh, Mannor, Pineau, and Tamar [GMP15], Shahriari et al. [SSW16], and Frazier [Fra18], and the references quoted there. Our purpose here is to focus on the aspects of the subject that are most closely connected to exact and approximate DP.

**A Dynamic Programming Formulation**

The sequential estimation problem just described, viewed as a DP problem, involves a state at time $k$, which is the posterior (or belief state) $b_k$, and a control/action at time $k$, which is the point index $u_{k+1}$ selected for observation. The transition equation according to which the state evolves, is

$$b_{k+1} = B_k(b_k, u_{k+1}, z_{u_{k+1}}), \quad k = 0, \ldots, N - 1;$$

cf. Eq. (2.80). To complete the DP formulation, we need to introduce a cost structure. To this end, we assume that observing $\theta_u$, as per Eq. (2.79), incurs a cost $c(u)$, and that there is a terminal cost $G(b_N)$ that depends of the final posterior distribution; as an example, the function $G$ may involve the mean and covariance corresponding to $b_N$. 

114
The corresponding DP algorithm is given by

\[
J^*_k(b_k) = \min_{u_{k+1} \in \{1, \ldots, m\}} \left[ c(u_{k+1}) + E_{z_{uk+1}} \left\{ J^*_{k+1}(B_k(b_k, u_{k+1}, z_{u_{k+1}}) \mid b_k, u_{k+1}) \right\} \right],
\]

and proceeds backwards from the terminal condition

\[
J^*_N(b_N) = G(b_N).
\]

The expected value in the right side of the DP equation (2.81) is taken with respect to the conditional distribution of \( z_{uk+1} \), given \( b_k \) and the choice \( u_{k+1} \). The observation cost \( c(u) \) may be 0 or a constant for all \( u \), but it can also have a more complicated dependence on \( u \). The terminal cost \( G(b_N) \) may be a suitable measure of surrogate “fidelity” that depends on the posterior mean and covariance of \( \theta \) corresponding to \( b_N \).

Generally, executing the DP algorithm (2.81) is practically infeasible, because the space of posterior distributions is infinite-dimensional. In the Gaussian case where the a priori distribution \( b_0 \) is Gaussian and the noise variables \( w_u \) are Gaussian, the posterior \( b_k \) is \( m \)-dimensional Gaussian, so it is characterized by its mean and covariance, and can be specified by a finite set of numbers. Despite this simplification, the DP algorithm (2.81) is prohibitively time-consuming even under Gaussian assumptions, except for simple special cases. We consequently resort to approximation in value space, whereby the function \( J^*_{k+1} \) in the right side of Eq. (2.81) is replaced by an approximation \( \tilde{J}_{k+1} \).

**Approximation in Value Space**

The most popular BO methodology makes use of a myopic/greedy policy \( \mu_{k+1} \), which at each time \( k \) and given \( b_k \), selects a point \( \hat{u}_{k+1} = \mu_{k+1}(b_k) \) for the next observation, using some calculation involving an *acquisition function*. This function, denoted \( A_k(b_k, u_{k+1}) \), quantifies some form of “expected benefit” for an observation at \( u_{k+1} \), given the current posterior \( b_k \).† The myopic policy selects the next point at which to observe, \( \hat{u}_{k+1} \), by

† A common type of acquisition function is the *upper confidence bound*, which has the form

\[
A_k(b_k, u) = T_k(b_k, u) + \beta R_k(b_k, u),
\]

where \( T_k(b_k, u) \) is the negative of the mean of \( f(u) \) under the posterior distribution \( b_k \), \( R_k(b_k, u) \) is the standard deviation of \( f(u) \) under the posterior distribution \( b_k \), and \( \beta \) is a tunable positive scalar parameter. Thus \( T_k(b_k, u) \) can be
by maximizing the acquisition function:

$$\hat{u}_{k+1} \in \arg \max_{u_{k+1} \in \{1, \ldots, m\}} A_k(b_k, u_{k+1}).$$  \hfill (2.83)$$

Several ways to define suitable acquisition functions have been proposed, and an important issue is to be able to calculate economically its values $A_k(b_k, u_{k+1})$ for the purposes of the maximization in Eq. (2.83). Another important issue of course is to be able to calculate the posterior $b_k$ economically.

Approximation in value space is an alternative approach, which is based on the DP formulation of the preceding section. In particular, in this approach we approximate the DP algorithm (2.81) by replacing $J^*_{k+1}$ with an approximation $\tilde{J}_{k+1}$ in the minimization of the right side. Thus we select the next observation at point $\tilde{u}_{k+1}$ according to

$$\tilde{u}_{k+1} \in \arg \min_{u_{k+1} \in \{1, \ldots, m\}} Q_k(b_k, u_{k+1}), \quad k = 0, \ldots, N - 1, \quad (2.84)$$

where $Q_k(b_k, u_{k+1})$ is the Q-factor corresponding to the pair $(b_k, u_{k+1})$, given by

$$Q_k(b_k, u_{k+1}) = c(u_{k+1}) + \mathbb{E}_{z_{u_{k+1}}} \left\{ \tilde{J}_{k+1}(B_k(b_k, u_{k+1}, z_{u_{k+1}})) \mid b_k, u_{k+1} \right\}. \quad (2.85)$$

The expected value in the preceding equation is taken with respect to the conditional probability distribution of $z_{u_{k+1}}$ given $(b_k, u_{k+1})$, which can be computed using $b_k$ and the given distribution of the noise $w_{u_{k+1}}$. Thus if $b_k$ and $\tilde{J}_{k+1}$ are available, we may use Monte Carlo simulation to determine the Q-factors $Q_k(b_k, u_{k+1})$ for all $u_{k+1} \in \{1, \ldots, m\}$, and select as next point for observation the one that corresponds to the minimal Q-factor [cf. Eq. (2.84)].

**Rollout Algorithms for Bayesian Optimization**

A special case of approximation in value space is the rollout algorithm, whereby the function $J^*_{k+1}$ in the right side of the DP Eq. (2.81) is replaced by the cost function of some base policy $\mu_{k+1}(b_k)$, $k = 0, \ldots, N - 1$. Thus, viewed as an exploitation index (encoding our desire to search within parts of the space where $f$ takes low value), while $R_k(b_k, u)$ can be viewed as an exploration index (encoding our desire to search within parts of the space that are relatively unexplored). There are several other popular acquisition functions, which directly or indirectly embody a tradeoff between exploitation and exploration. A popular example is the expected improvement acquisition function, which is equal to the expected value of the reduction of $f(u)$ relative to the minimal value of $f$ obtained up to time $k$ (under the posterior distribution $b_k$).
given a base policy the rollout algorithm uses the cost function of this policy as the function $\tilde{J}_{k+1}$ in the approximation in value space scheme (2.84)-(2.85). The values of $\tilde{J}_{k+1}$ needed for the Q-factor calculations in Eq. (2.85) can be computed or approximated by simulation. Greedy/myopic policies based on an acquisition function [cf. Eq. (2.83)] have been suggested as base policies in various rollout proposals.†

![Figure 2.10.5](chart.png)

**Figure 2.10.5** Illustration of rollout at the current posterior $b_k$. For each $u_{k+1} \in \{1, \ldots, m\}$, we compute the Q-factor $Q_k(b_k, u_{k+1})$ by using Monte-Carlo simulation with samples from $w_{u_{k+1}}$ and a base heuristic that uses an acquisition function starting from each possible posterior $b_{k+1}$. The rollout may extend to the end of the horizon $N$, or it may be truncated after a few steps.

In particular, given $b_k$, the rollout algorithm computes for each $u_{k+1} \in$...

† The rollout algorithm for BO was first proposed under Gaussian assumptions by Lam, Wilcox, and Wolpert [LWW16]. It was further discussed by Jiang et al. [JJB20], [JCG20], Lee et al. [LEC20], Lee [Lee20], Yue and Kontar [YuK20], Lee et al. [LEP21], Paulson, Sorouifar, and Chakrabarty [PSC22], where it is also referred to as “nonmyopic BO” or “nonmyopic sequential experimental design.” For related work, see Gerlach, Hoffmann, and Charlish [GHC21]. These papers also discuss various approximations to the rollout approach, and generally report encouraging computational results. Section 3.5 of the author’s book [Ber20a] focuses on rollout algorithms for surrogate and Bayesian optimization.
\{1, \ldots, m\}\) a Q-factor value \(Q_k(b_k, u_{k+1})\) by simulating the base policy for multiple time steps starting from all possible posteriors \(b_{k+1}\) that can be generated from \((b_k, u_k)\), and by accumulating the corresponding cost [including a terminal cost such as \(G(b_N)\)]; see Fig. 2.10.5. It then selects the next point \(\tilde{u}_{k+1}\) for observation by using the Q-factor minimization of Eq. (2.84).

Note that the equation

\[ b_{k+1} = B_k(b_k, u_{k+1}, z_{u_{k+1}}), \quad k = 0, \ldots, N - 1, \]

which governs the evolution of the posterior distribution (or belief state), is stochastic because \(z_{u_{k+1}}\) involves the stochastic noise \(w_{u_{k+1}}\). Thus some Monte Carlo simulation is unavoidable in the calculation of the Q-factors \(Q_k(b_k, u_{k+1})\). On the other hand, one may greatly reduce the Monte Carlo computational burden by employing a certainty equivalence approximation, which at stage \(k\), treats only the noise \(w_{u_{k+1}}\) as stochastic, and replaces the noise variables \(w_{u_{k+2}}, w_{u_{k+3}}, \ldots\) after the first stage of the calculation, by deterministic quantities such as their means \(\hat{w}_{u_{k+2}}, \hat{w}_{u_{k+3}}, \ldots\).

The simulation of the Q-factor values may also involve other approximations, some of which have been suggested in various proposals for rollout-based BO. For example, if the number of possible observations \(m\) is very large, we may compute and compare the Q-factors of only a subset. In particular, at a given time \(k\), we may rank the observations by using an acquisition function, select a subset \(U_{k+1}\) of most promising observations, compute their Q-factors \(Q_k(b_k, u_{k+1})\), \(u_{k+1} \in U_{k+1}\), and select the observation whose Q-factor is minimal; this idea has been used in the case of the Wordle puzzle in the paper by Bhambri, Bhattacharjee, and Bertsekas [BBB22], which will be discussed in the next section.

Multiagent Rollout for Bayesian Optimization

In some BO applications there arises the possibility of simultaneously performing multiple observations before receiving feedback about the corresponding observation outcomes. This occurs, among others, in two important contexts:

(a) In parallel computation settings, where multiple processors are used to perform simultaneously expensive evaluations of the function \(f\) at multiple points \(u\). These evaluations may involve some form of truncated simulation, so they yield evaluations of the form \(z_u = \theta_u + w_u\), where \(w_u\) is the simulation noise.

(b) In distributed sensor systems, where a number of sensors provide in parallel relevant information about the random vector \(\theta\) that we want to estimate; see e.g., the recent paper by Li, Krakow, and Gopalswamy [LKG21], which describes related multisensor estimation problems, based on the multiagent rollout methodology of Section 2.9.
Of course in such cases we may treat the entire set of simultaneous observations as a single observation within an enlarged Cartesian product space of observations, but there is a fundamental difficulty: the size of the observation space (and hence the number of Q-factors to be calculated by rollout at each time step) grows exponentially with the number of simultaneous observations. This in turn greatly increases the computational requirements of the rollout algorithm.

To address this difficulty, we may employ the methodology of multi-agent rollout whereby the policy improvement is done one-agent-at-a-time in a given order, with (possibly partial) knowledge of the choices of the preceding agents in the order. As a result, the amount of computation for each policy improvement grows linearly with the number of agents, as opposed to exponentially for the standard all-agents-at-once method. At the same time the theoretical cost improvement property of the rollout algorithm can be shown to be preserved, while the empirical evidence suggests that great computational savings are achieved with hardly any performance degradation.

**Generalization to Sequential Estimation of Random Vectors**

Aside from BO, there are several other types of simple sequential estimation problems, which involve “independent sampling,” i.e., problems where the choice of an observation type does not affect the quality, cost, or availability of observations of other types. A common class of problems that contains BO as a special case and admits a similar treatment, is to sequentially estimate an \( m \)-dimensional random vector \( \theta = (\theta_1, \ldots, \theta_m) \) by using \( N \) linear observations of \( \theta \) of the form

\[
z_u = a_u' \theta + w_u, \quad u \in \{1, \ldots, n\},
\]

where \( n \) is some integer. Here \( w_u \) are independent random variables with given probability distributions, the \( m \)-dimensional vectors \( a_u \) are known, and \( a_u' \theta \) denotes the inner product of \( a_u \) and \( \theta \). Similar to the case of BO, the problem simplifies if the given a priori distribution of \( \theta \) is Gaussian, and the random variables \( w_u \) are independent and Gaussian. Then, the posterior distribution of \( \theta \), given any subset of observations, is Gaussian (thanks to the linearity of the observations), and can be calculated in closed form.

Observations are generated sequentially at times 1, \ldots, \( N \), one at a time and with knowledge of the outcomes of the preceding observations, by choosing an index \( u_k \in \{1, \ldots, n\} \) at time \( k \), at a cost \( c(u_k) \). Thus \( u_k \) are the optimization variables, and affect both the quality of estimation of \( \theta \) and the observation cost. The objective, roughly speaking, is to select \( N \) observations to estimate \( \theta \) in a way that minimizes an appropriate cost function; for example, one that penalizes some form of estimation
error plus the cost of the observations. We can similarly formulate the corresponding optimization problem in terms of $N$-stage DP, and develop rollout algorithms for its approximate solution.

## 2.11 Adaptive Control by Rollout with a POMDP Formulation

In this section, we discuss various approaches for the approximate solution of Partially Observed Markovian Decision Problems (POMDP) with a special structure, which is well-suited for adaptive control, as well as other contexts that involve search for a hidden object.† It is well known that POMDP are among the most challenging DP problems, and nearly always require the use of approximations for (suboptimal) solution.

The application and implementation of rollout and approximate PI methods to general finite-state POMDP is described in the author’s RL book [Ber19a] (Section 5.7.3). Here we will focus attention on a special class of POMDP where the state consists of two components:

(a) A perfectly observed component $x_k$ that evolves over time according to a discrete-time equation.

(b) An unobserved component $\theta$ that stays constant and is estimated through the perfect observations of the component $x_k$.

We view $\theta$ as a parameter in the system equation that governs the evolution of $x_k$, hence the connection with adaptive control. Thus we have

$$x_{k+1} = f_k(x_k, \theta, u_k, w_k),$$  \hspace{1cm} (2.86)

where $u_k$ is the control at time $k$, selected from a set $U_k(x_k)$, and $w_k$ is a random disturbance with given probability distribution that depends on $(x_k, \theta, u_k)$. We will assume that $\theta$ can take one of $m$ known values $\theta^1, \ldots, \theta^m$:

$$\theta \in \{\theta^1, \ldots, \theta^m\}.$$  

see Fig. 2.11.1.

The a priori probability distribution of $\theta$ is given and is updated based on the observed values of the state components $x_k$ and the applied controls $u_k$. In particular, we assume that the information vector

$$I_k = \{x_0, \ldots, x_k, u_0, \ldots, u_{k-1}\}$$

† In Section 1.6.6, we discussed the indirect adaptive control approach, which enforces a separation of the controller into a system identification algorithm and a policy reoptimization algorithm. The POMDP approach of this section (also summarized in Section 1.6.6), does not assume such an a priori separation, and is thus founded on a more principled algorithmic framework.
is available at time $k$, and is used to compute the conditional probabilities

$$b_{k,i} = P\{\theta = \theta^i \mid I_k\}, \quad i = 1, \ldots, m.$$  

These probabilities form a vector

$$b_k = (b_{k,1}, \ldots, b_{k,m}),$$

which together with the perfectly observed state $x_k$, form the pair $(x_k, b_k)$ that is commonly called the belief state of the POMDP at time $k$.

Note that according to the classical methodology of POMDP (see e.g., [Ber17a], Chapter 4), the belief component $b_{k+1}$ is determined by the belief state $(x_k, b_k)$, the control $u_k$, and the observation obtained at time $k+1$, i.e., $x_{k+1}$. Thus $b_k$ can be updated according to an equation of the form

$$b_{k+1} = B_k(x_k, b_k, u_k, x_{k+1}),$$

where $B_k$ is an appropriate function, which can be viewed as a recursive estimator of $\theta$. There are several approaches to implement this estimator (perhaps with some approximation error), including the use of Bayes’ rule and the simulation-based method of particle filtering.

The preceding mathematical model forms the basis for a classical adaptive control formulation, where each $\theta^i$ represents a set of system parameters, and the computation of the belief probabilities $b_{k,i}$ can be viewed as the outcome of a system identification algorithm. In this context, the problem becomes one of dual control, a combined identification and control problem, whose optimal solution is notoriously difficult.

Another interesting context arises in search problems, where $\theta$ specifies the locations of one or more objects of interest within a given space. Some puzzles, including the popular Wordle game, fall within this category, as we will discuss briefly later in this section.
The Exact DP Algorithm - Approximation in Value Space

We will now describe an exact DP algorithm that operates in the space of information vectors \(I_k\). To describe this algorithm, let us denote by \(J_k(I_k)\) the optimal cost starting at information vector \(I_k\) at time \(k\). We can view \(I_k\) as a state of the POMDP, which evolves over time according to the equation

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

Viewing this as a system equation, whose right hand side involves the state \(I_k\), the control \(u_k\), and the disturbance \(w_k\), the DP algorithm takes the form

\[
J^*_k(I_k) = \min_{u_k \in U_k(x_k)} \mathbb{E}_{\theta, w_k} \left\{ g_k(x_k, \theta, u_k, w_k) + J^*_{k+1}(I_{k+1}) \mid I_k, u_k \right\}
\]

for \(k = 0, \ldots, N-1\), with \(J_N(I_N) = g_N(x_N)\); see e.g., the DP textbook [Ber17a], Section 4.1.

By using the law of iterated expectations,

\[
\mathbb{E}_{\theta, w_k} \left\{ \cdot \mid I_k, u_k \right\} = \mathbb{E}_\theta \{ \mathbb{E}_{w_k} \left\{ \cdot \mid I_k, \theta, u_k \right\} \mid I_k, u_k \},
\]

we can rewrite this DP algorithm as

\[
J^*_k(I_k) = \min_{u_k \in U_k(x_k)} \sum_{i=1}^m b_{k,i} \mathbb{E}_{w_k} \left\{ g_k(x_k, \theta^i, u_k, w_k) + J^*_{k+1}(I_{k+1}) \mid I_k, \theta^i, u_k \right\}
\]

for \(k = 0, \ldots, N-1\), with \(J_N(I_N) = g_N(x_N)\); see e.g., the DP textbook [Ber17a], Section 4.1.

The summation over \(i\) above represents the expected value of \(\theta\) conditioned on \(I_k\) and \(u_k\).

The algorithm (2.88) is typically very hard to implement, in part because of the dependence of \(J^*_{k+1}\) on the entire information vector \(I_{k+1}\), which expands in size according to

\[
I_{k+1} = (I_k, x_{k+1}, u_k).
\]

To address this implementation difficulty, we may use approximation in value space, based on replacing \(J^*_{k+1}(I_{k+1})\) in the DP algorithms (2.87) and (2.88) with some function \(\tilde{J}_{k+1}(I_{k+1})\) such that the expected value

\[
\mathbb{E}_{w_k} \left\{ \tilde{J}_{k+1}(I_{k+1}) \mid I_k, \theta^i, u_k \right\}
\]

122
can be obtained (either off-line or on-line) with a tractable computation for any \((I_k, \theta^i, u_k)\).

Here we will focus on functions \(\hat{J}_{k+1}\) with a special structure that facilitates the implementation of the corresponding approximation in value space scheme. One such possibility is to use the optimal cost functions corresponding to the \(m\) parameters \(\theta^i\),

\[
\hat{J}^i_{k+1}(x_{k+1}), \quad i = 1, \ldots, m. \quad (2.89)
\]

In particular, \(\hat{J}^i_{k+1}(x_{k+1})\) is the optimal cost that would be obtained starting from state \(x_{k+1}\) under the assumption that \(\theta = \theta^i\); this corresponds to a perfect state information problem. Then an approximation in value space scheme with one-step lookahead minimization is given by

\[
\hat{u}_k \in \arg\min_{u_k \in U_k(x_k)} \sum_{i=1}^{m} b_{k,i} E_{k} \left\{ g_k(x_k, \theta^i, u_k, w_k) + \hat{J}^i_{k+1}(f_k(x_k, \theta^i, u_k, w_k)) \mid x_k, \theta^i, u_k \right\}. \quad (2.90)
\]

In particular, instead of the optimal control, which minimizes the optimal Q-factor of \((I_k, u_k)\) appearing in the right side of Eq. (2.87), we apply control \(\hat{u}_k\) that minimizes the expected value over \(\theta\) of the optimal Q-factors that correspond to fixed values of \(\theta\).

In the case where the horizon is infinite, it is reasonable to expect that the estimate of the parameter \(\theta\) improves over time, and that with a suitable estimation scheme, it converges asymptotically to the correct value of \(\theta\), call it \(\theta^*\), i.e.,

\[
\lim_{k \to \infty} b_{k,i} = \begin{cases} 
1 & \text{if } \theta^i = \theta^*, \\
0 & \text{if } \theta^i \neq \theta^*.
\end{cases}
\]

Then it can be seen that the generated one-step lookahead controls \(\hat{u}_k\) are asymptotically obtained from the Bellman equation that corresponds to the correct parameter \(\theta^*\), and are typically optimal in some asymptotic sense. Schemes of this type have been extensively discussed in the adaptive control literature since the 70s; see the end-of-chapter references.

Generally, the optimal costs \(\hat{J}_{k+1}(x_{k+1})\) that correspond to the different parameter values \(\theta^i\) [cf. Eq. (2.89)] may be hard to compute, despite their perfect state information structure. An alternative possibility is to use off-line trained feature-based or neural network-based approximations to \(\hat{J}_{k+1}(x_{k+1})\). Another possibility, described next, is to use a rollout approach.

† In favorable special cases, such as linear quadratic problems, the optimal costs \(\hat{J}_{k+1}(x_{k+1})\) may be easily calculated in closed form. Still, however, even in such cases the calculation of the belief probabilities \(b_{k,i}\) may not be simple, and may require the use of a system identification algorithm.
Rollout and Cost Improvement

A simpler possibility for approximation in value space is to use the costs of given policies $\pi_i$ in place of the optimal costs $\hat{J}_{i+1}^{k}(x_{k+1})$. In this case the one-step lookahead scheme (2.90) takes the form

$$\tilde{u}_k \in \arg \min_{u_k \in \mathcal{U}_k(x_k)} \sum_{i=1}^{m} b_{k,i} E_{w_k} \left\{ g_k(x_k, \theta^i, u_k, w_k) + \hat{J}_{k+1, \pi}^i \left( f_k(x_k, \theta^i, u_k, w_k) \right) \mid x_k, \theta^i, u_k \right\},$$

(2.91)

with $\pi^i = \{\mu_{0}^i, \ldots, \mu_{N-1}^i\}, i = 1, \ldots, m$, being known policies, with components $\mu_k^i$ that depend on $x_k$. Here, the term

$$\hat{J}_{k+1, \pi}^i \left( f_k(x_k, \theta^i, u_k, w_k) \right)$$

in Eq. (2.91) is the cost of the base policy $\pi^i$, calculated starting from the next state

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

under the assumption that $\theta$ will stay fixed at the value $\theta = \theta^i$ until the end of the horizon. Note that the cost function of $\pi^i$, conditioned on $\theta = \theta^i$, $x_k$, and $u_k$, which is needed in Eq. (2.91), can be calculated by Monte Carlo simulation. This is made possible by the fact that the components $\mu_k^i$ of $\pi^i$ depend only on $x_k$ [rather than $I_k$ or the belief state $(x_k, b_k)$].

The preceding scheme has the character of a rollout algorithm, but strictly speaking, it does not qualify as a rollout algorithm because the policy components $\mu_k^i$ involve a dependence on $i$ in addition to the dependence on $x_k$. On the other hand if we restrict all the policies $\pi^i$ to be the same for all $i$, the corresponding functions $\mu_k$ depend only on $x_k$ and not on $i$, thus defining a legitimate base policy. It is then possible to view the rollout policy as being generated from the base policy through a policy iteration scheme. As a result, a cost improvement property can be shown.

Within our rollout context, a policy $\pi$ such that $\pi^i = \pi$ for all $i$ must be a robust policy, in the sense that it should work adequately well for all parameter values $\theta^i$. The choice of such a policy is likely problem-dependent. On the other hand robust policies have a long history in the context of adaptive control, and have been discussed widely (see e.g., the book by Jiang and Jiang [JiJ17], and the references quoted therein).

The Case of a Deterministic System

Let us now consider the case where the system (2.86) is deterministic of the form

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

(2.92)
Then, while the problem still has a stochastic character due to the uncertainty about the value of $\theta$, the DP algorithm (2.88) and its approximation in value space counterparts are greatly simplified because there is no expectation over $w_k$ to contend with. Indeed, given a state $x_k$, a parameter $\theta_i$, and a control $u_k$, the on-line computation of the control of the rollout-like algorithm (2.91), takes the form

$$\tilde{u}_k \in \arg \min_{u_k \in U_k(x_k)} \sum_{i=1}^m b_{k,i} \left( g_k(x_k, \theta_i, u_k) + \tilde{J}_{k+1, \pi^i}^i(f_k(x_k, \theta_i, u_k)) \right).$$ (2.93)

The computation of $\tilde{J}_{k+1, \pi^i}^i(f_k(x_k, \theta_i, u_k))$ involves a deterministic propagation from the state $x_{k+1}$ of Eq. (2.92) up to the end of the horizon, using the base policy $\pi^i$, while assuming that $\theta$ is fixed at the value $\theta_i$.

In particular, the term

$$Q_k(x_k, u_k, \theta^i) = g_k(x_k, \theta^i, u_k) + \tilde{J}_{k+1, \pi^i}^i(f_k(x_k, \theta^i, u_k))$$ (2.94)

appearing on the right side of Eq. (2.93) is viewed as a Q-factor that must be computed for every pair $(u_k, \theta^i)$, $u_k \in U_k(x_k)$, $i = 1, \ldots, m$, using the base policy $\pi^i$. The expected value of this Q-factor,

$$\tilde{Q}_k(x_k, u_k) = \sum_{i=1}^m b_{k,i} Q_k(u_k, \theta^i),$$

must then be calculated for every $u_k \in U_k(x_k)$, and the computation of the rollout control $\tilde{u}_k$ is obtained from the minimization

$$\tilde{u}_k \in \arg \min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k);$$ (2.95)

cf. Eq. (2.93). This computation is illustrated in Fig. 2.11.2.

The case of a deterministic system is particularly interesting because we can typically expect that the true parameter $\theta^*$ is identified in a finite number of stages, since at each stage $k$, we are receiving a noiseless measurement relating to $\theta$, namely the state $x_k$. Once this happens, the problem becomes one of perfect state information.

An illustration similar to the one of Fig. 2.11.2 applies to the rollout scheme (2.91) for the case of a stochastic system. In this case, a Q-factor

$$Q_k(x_k, u_k, \theta^i, w_k) = g_k(x_k, \theta^i, u_k, w_k) + \tilde{J}_{k+1, \pi^i}^i(f_k(x_k, \theta^i, u_k, w_k))$$

must be calculated for every triplet $(u_k, \theta^i, w_k)$, using the base policy $\pi^i$. The rollout control $\tilde{u}_k$ is obtained by minimizing the expected value of this Q-factor [averaged using the distribution of $(\theta, w_k)$]; cf. Eq. (2.91).

An interesting and intuitive example that demonstrates the deterministic system case is the popular Worlde puzzle.
Figure 2.11.2 Schematic illustration of adaptive control by rollout for deterministic systems; cf. Eqs. (2.94) and (2.95). The Q-factors $Q(x_k, u_k, \theta_i)$ are averaged over $\theta_i$, using the current belief distribution $b_k$, and the control applied is the one that minimizes the averaged Q-factor

$$\hat{Q}_k(x_k, u_k) = \sum_{i=1}^{m} b_{k,i} Q(x_k, u_k, \theta_i)$$

over $u_k \in U_k(x_k)$.

Example 2.11.1 (The Wordle Puzzle)

In the classical form of this puzzle, we try to guess a mystery word $\theta^*$ out of a known finite collection of 5-letter words. This is done with sequential guesses each of which provides additional information on the correct word $\theta^*$, by using certain given rules to shrink the current mystery list (the smallest list that contains $\theta^*$, based on the currently available information). The objective is to minimize the number of guesses to find $\theta^*$ (using more than 6 guesses is considered to be a loss). This type of puzzle descends from the classical family of Mastermind puzzles that centers around decoding a secret sequence of objects (e.g., letters or colors) using partial observations.

The rules for shrinking the mystery list relate to the common letters between the word guesses and the mystery word $\theta^*$, and they will not be described here (there is a large literature regarding the Wordle puzzle). Moreover, $\theta^*$ is assumed to be chosen from the initial collection of 5-letter words according to a uniform distribution. Under this assumption, it can be shown that the belief distribution $b_k$ at stage $k$ continues to be uniform over the mystery list. As a result, we may use as state $x_k$ the mystery list at stage $k$, which evolves deterministically according to an equation of the form (2.92), where $u_k$ is the guess word at stage $k$. There are several base policies to
The system function \( f_k \) does not depend on the current state \( x_k \), so the system provides a decision-dependent noisy observation of \( \theta \).

The rollout approach also applies to several variations of the Wordle puzzle. Such variations may include for example a larger length \( \ell > 5 \) of mystery words, and/or a known nonuniform distribution over the initial collection of \( \ell \)-letter words; see [BBB22].

The Case of Sequential Estimation - Alternative Base Policies

We finally note that the adaptive control framework of this section contains as a special case the sequential estimation framework of the preceding section. This special case corresponds to a dynamic system of the form

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

where the state \( x_{k+1} \) is the observation at time \( k+1 \) and exhibits no explicit dependence on the preceding observation \( x_k \), but depends on the stochastic disturbance \( w_k \), and on the decision \( u_k \); cf. Figs. 2.11.1 and 2.11.3. This decision may involve a cost and determines the type of next observation out of a collection of possible types.

While the rollout methodology of the present section applies to sequential estimation problems, other rollout algorithms may also be used, depending on the problem’s detailed structure. In particular, the rollout algorithms for Bayesian optimization of the works noted in Section 2.10 involve base policies that depend on the current belief state \( b_k \), rather than the current state \( x_k \). Another example of rollout for adaptive control, which uses a base policy that depends on the current belief state is given.
in Section 6.7 of the book [Ber22a]. For work on related stochastic optimal
control problems that involve observation costs and the rollout approach,
see Antunes and Heemels [AnH14], and Khashooei, Antunes, and Heemels
[KAH15].

2.12 ROLLOUT FOR MINIMAX CONTROL

The problem of optimal control of uncertain systems is usually treated
within a stochastic framework, whereby all disturbances \( w_0, \ldots, w_{N-1} \) are
described by probability distributions, and the expected value of the cost is
minimized. However, in many practical situations a stochastic description
of the disturbances may not be available, but one may have information
with less detailed structure, such as bounds on their magnitude. In other
words, one may know a set within which the disturbances are known to lie,
but may not know the corresponding probability distribution. Under these
circumstances one may use a minimax approach, whereby the worst possi-
ble values of the disturbances within the given set are assumed to occur.
Within this context, we take the view that the disturbances are chosen by
an antagonistic opponent. The minimax approach is also connected with
two-player games, when in lack of information about the opponent, we
adopt a worst case viewpoint during on-line play, as well as with contexts
where we wish to guard against adversarial attacks.\footnote{The minimax approach to decision and control has its origins in the 50s
and 60s. It is also referred to by other names, depending on the underlying
context, such as robust control, robust optimization, control with a set membership
description of the uncertainty, and games against nature. In this book, we will
be using the minimax control name.}

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

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

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

\[
J_N^*(x_N) = g_N(x_N), \tag{2.96}
\]
\[ J^*_k(x_k) = \min_{u_k \in U(x_k)} \max_{w_k \in W_k(x_k, u_k)} \left[ g_k(x_k, u_k, w_k) + J^*_{k+1} \left( f_k(x_k, u_k, w_k) \right) \right]. \]

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”

\[ \max_{w_t \in W_t(x_t, \mu_t(x_t))} \left[ g_N(x_N) + \sum_{t=k}^{N-1} g_t(x_t, \mu_t(x_t), w_t) \right]. \]

We argue that if \( \pi^* = \{\mu_0^*, \mu_1^*, \ldots, \mu_{N-1}^*\} \) is an optimal policy for the min-max problem, then the tail of the policy \( \{\mu_k^*, \mu_{k+1}^*, \ldots, \mu_{N-1}^*\} \) is optimal for the tail subproblem. The optimal cost of this subproblem is \( J_k^* (x_k) \), as given by the DP algorithm (2.96)-(2.97). The algorithm expresses the intuitive fact that when at state \( x_k \) at time \( k \), then regardless of what happened in the past, we should choose \( u_k \) that minimizes the worst/maximum value over \( w_k \) of the sum of the current stage cost plus the optimal cost of the tail subproblem that starts from the next state. This argument requires a mathematical proof, which turns out to involve a few fine points. For a detailed mathematical derivation, we refer to the author’s textbook [Ber17a], Section 1.6. However, the DP algorithm (2.96)-(2.97) is correct assuming finite state and control spaces, among other cases.

**Approximation in Value Space and Minimax Rollout**

The approximation ideas for stochastic optimal control are also relevant within the minimax context. In particular, approximation in value space with one-step lookahead applies at state \( x_k \) a control

\[ \bar{u}_k \in \arg \min_{u_k \in U(x_k)} \max_{w_k \in W_k(x_k, u_k)} \left[ g_k(x_k, u_k, w_k) + J_{k+1} (x_{k+1}) \right], \]

where \( J_{k+1} (x_{k+1}) \) is an approximation to the optimal cost-to-go \( J^*_{k+1} (x_{k+1}) \) from state \( x_{k+1} \).

Rollout is obtained when this approximation is the tail cost of some base policy \( \pi = \{\mu_0, \ldots, \mu_{N-1}\} \):

\[ J_{k+1} (x_{k+1}) = J_{k+1, \pi} (x_{k+1}). \]

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 longest path problem defined on an acyclic graph, since the control variables \( u_{k+1}, \ldots, u_{N-1} \) are determined by the base policy. It is then straightforward to implement rollout: at \( x_k \) we generate all next states of the form

\[ x_{k+1} = f_k(x_k, u_k, w_k) \]
corresponding to all possible values of $u_k \in U_k(x_k)$ and $w_k \in W_k(x_k, u_k)$. We then run the maximization/longest path problem described above to compute $\tilde{J}_{k+1}(x_{k+1})$ from each of these possible next states $x_{k+1}$. Finally, we obtain the rollout control $\tilde{u}_k$ by solving the minimax problem in Eq. (2.98). Moreover, it is possible to use truncated rollout to approximate the tail cost of the base policy.†

Note that like all rollout algorithms, the minimax rollout algorithm is well-suited for on-line replanning in problems where data may be changing or may be revealed during the process of control selection.

We mentioned earlier that deterministic problems allow a more general form of rollout, whereby we may use a base heuristic that need not be a legitimate policy, i.e., it need not be sequentially consistent. For cost improvement it is sufficient that the heuristic be sequentially improving. A similarly more general view of rollout is not easily constructed for stochastic problems, but is possible for minimax control.

In particular, suppose that at any state $x_k$ there is a heuristic that generates a sequence of feasible controls and disturbances, and corresponding states,

$$\{u_k, w_k, x_{k+1}, u_{k+1}, w_{k+1}, x_{k+2}, \ldots, u_{N-1}, w_{N-1}, x_N\},$$

with corresponding cost

$$H_k(x_k) = g_k(x_k, u_k, w_k) + \cdots + g_{N-1}(x_{N-1}, u_{N-1}, w_{N-1}) + g_N(x_N).$$

Then the rollout algorithm applies at state $x_k$ a control

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

This does not preclude the possibility that the disturbances $w_k, \ldots, w_{N-1}$ are chosen by an antagonistic opponent, but allows more general choices of disturbances, obtained for example, by some form of approximate maximization. For example, when the disturbance involves multiple components, $w_k = (w_k^1, \ldots, w_k^m)$, corresponding to multiple opponent agents, the heuristic may involve an agent-by-agent maximization strategy.

The sequential improvement condition, similar to the deterministic case, is that for all $x_k$ and $k$,

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

It guarantees cost improvement, i.e., that for all $x_k$ and $k$, the rollout policy

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

† For a more detailed discussion of this implementation, see the author’s paper [Ber19b] (Section 5.4).
satisfies

\[ J_{k,\hat{\pi}}(x_k) \leq H_k(x_k). \]

Thus, generally speaking, minimax rollout is fairly similar to rollout for deterministic as well as stochastic DP problems. The main difference with deterministic (or stochastic) problems is that to compute the Q-factor of a control \( u_k \), we need to solve a maximization problem, rather than carry out a deterministic (or Monte-Carlo, respectively) simulation with the given base policy.

**Example 2.12.1 (Pursuit-Evasion Problems)**

Consider a pursuit-evasion problem with state \( x_k = (x^1_k, x^2_k) \), where \( x^1_k \) is the location of the minimizer/pursuer and \( x^2_k \) is the location of the maximizer/evader, at stage \( k \), in a (finite node) graph defined in two- or three-dimensional space. There is also a cost-free and absorbing termination state that consists of a subset of pairs \( (x^1, x^2) \) that includes all pairs with \( x^1 = x^2 \).

The pursuer chooses one out of a finite number of actions \( u_k \in U_k(x_k) \) at each stage \( k \), when at state \( x_k \), and if the state is \( x_k \) and the pursuer selects \( u_k \), the evader may choose from a known set \( X_{k+1}(x_k, u_k) \) of next states \( x_{k+1} \), which depends on \( (x_k, u_k) \). The objective of the pursuer is to minimize a nonnegative terminal cost \( g(x^1_N, x^2_N) \) at the end of \( N \) stages (or reach the termination state, which has cost 0 by assumption). A reasonable base policy for the pursuer can be precomputed by DP as follows: given the current (nontermination) state \( x_k = (x^1_k, x^2_k) \), make a move along the path that starts from \( x^1_k \) and minimizes the terminal cost after \( N - k \) stages, under the assumption that the evader will stay motionless at his current location \( x^2_k \). (In a variation of this policy, the DP computation is done under the assumption that the evader will follow some nominal sequence of moves.)

For the on-line computation of the rollout control, we need the maximal value of the terminal cost that the evader can achieve starting from every \( x_{k+1} \in X_{k+1}(x_k, u_k) \), assuming that the pursuer will follow the base policy (which has already been computed). We denote this maximal value by \( \tilde{J}_{k+1}(x_{k+1}) \). The required values \( \tilde{J}_{k+1}(x_{k+1}) \) can be computed by an \((N - k)\)-stage DP computation involving the optimal choices of the evader, while assuming the pursuer uses the (already computed) base policy. Then the rollout control for the pursuer is obtained from the minimization

\[
\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \max_{x_{k+1} \in X_{k+1}(x_k, u_k)} \tilde{J}_{k+1}(x_{k+1}).
\]

Note that the preceding algorithm can be adapted for the imperfect information case where the pursuer knows \( x^2_k \) imperfectly. This is possible by using a form of assumed certainty equivalence: the pursuer’s base policy and the evader’s maximization can be computed by using an estimate of the current location \( x^2_k \) instead of the unknown true location.

In the preceding pursuit-evasion example, the choice of the base policy was facilitated by the special structure of the problem. Generally, however,
finding a suitable base policy that can be conveniently implemented is an important problem-dependent issue.

**Variants of Minimax Rollout**

Several of the variants of rollout discussed earlier have analogs in the min-max context, e.g., truncation with terminal cost approximation, multistep and selective step lookahead, and multiagent rollout. In particular, in the \( \ell \)-step lookahead variant, we solve the \( \ell \)-stage problem

\[
\min_{u_k, \mu_{k+1}, \ldots, \mu_{k+\ell-1}} \max_{w_k \in W_k(u_k)} \left\{ g_k(x_k, u_k, w_k) + \sum_{t=k+1}^{k+\ell-1} g_t(x_t, \mu_t(x_t), w_t) + H_{k+\ell}(x_{k+\ell}) \right\},
\]

we find an optimal solution \( \tilde{u}_k, \tilde{\mu}_{k+1}, \ldots, \tilde{\mu}_{k+\ell-1} \), and we apply the first component \( \tilde{u}_k \) of that solution. As an example, this type of problem is solved at each move of chess programs like AlphaZero, where the terminal cost function is encoded through a position evaluator. In fact when multi-step lookahead is used, special techniques such as alpha-beta pruning may be used to accelerate the computations by eliminating unnecessary portions of the lookahead graph. These techniques are well-known in the context of the two-person computer game methodology, and are used widely in games such as chess.

It is interesting to note that, contrary to the case of stochastic optimal control, there is an on-line constrained form of rollout for minimax control. Here there are some additional trajectory constraints of the form

\[
(x_0, u_0, \ldots, u_{N-1}, x_N) \in C,
\]

where \( C \) is an arbitrary set. The modification needed is similar to the one of Section 6.6: at partial trajectory

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

generated by rollout, we use a heuristic with cost function \( H_{k+1} \) to compute the Q-factor

\[
\hat{Q}_k(\tilde{x}_k, u_k) = \max_{w_k, \ldots, w_{N-1}} \left[ g_k(\tilde{x}_k, u_k, w_k) + H_{k+1}(f_k(\tilde{x}_k, u_k, w_k), w_{k+1}, \ldots, w_{N-1}) \right]
\]

for each \( u_k \) in the set \( \hat{U}_k(y_k) \) that guarantee feasibility [we can check feasibility here by running some algorithm that verifies whether the future
disturbances \( w_k, \ldots, w_{N-1} \) can be chosen to violate the constraint under the base policy, starting from \((\tilde{y}_k, u_k)\). Once the set of “feasible controls” \( \tilde{U}_k(\tilde{y}_k) \) is computed, we can obtain the rollout control by the Q-factor minimization:

\[
\tilde{u}_k \in \arg \min_{u_k \in \tilde{U}_k(\tilde{y}_k)} \tilde{Q}_k(\tilde{x}_k, u_k).
\]

We may also use fortified versions of the unconstrained and constrained rollout algorithms, which guarantee a feasible cost-improved rollout policy. This requires the assumption that the base heuristic at the initial state produces a trajectory that is feasible for all possible disturbance sequences. Similar to the deterministic case, there are also truncated and multiagent versions of the minimax rollout algorithm.

**Example 2.12.2 (Multiagent Minimax Rollout)**

Let us consider a minimax problem where the minimizer’s choice involves the collective decision of \( m \) agents, \( u = (u^1, \ldots, u^m) \), with \( u^\ell \) corresponding to agent \( \ell \), and constrained to lie within a finite set \( U^\ell \). Thus \( u \) must be chosen from within the set

\[
U = U^1 \times \ldots \times U^m,
\]

which is finite but grows exponentially in size with \( m \). The maximizer’s choice \( w \) is constrained to belong to a finite set \( W \). We consider multiagent rollout for the minimizer, and for simplicity, we focus on a two-stage problem. However, there are straightforward extensions to a more general multistage framework.

In particular, we assume that the minimizer knowing an initial state \( x_0 \), chooses \( u = (u^1, \ldots, u^m) \), with \( u^\ell \in U^\ell, \ell = 1, \ldots, m \), and a state transition

\[
x_1 = f_0(x_0, u)
\]

occurs with cost \( g_0(x_0, u) \). Then the maximizer, knowing \( x_1 \), chooses \( w \in W \), and a terminal state

\[
x_2 = f_1(x_1, w)
\]

is generated with cost

\[
g_1(x_1, w) + g_2(x_2).
\]

The problem is to select \( u \in U \), to minimize

\[
g_0(x_0, u) + \max_{w \in W} \left[ g(x_1, w) + g_2(x_2) \right].
\]

The exact DP algorithm for this problem is given by

\[
J^*_1(x_1) = \max_{w \in W} \left[ g_1(x_1, w) + g_2(f_1(x_1, w)) \right],
\]

\[
J^*_0(x_0) = \min_{u \in U} \left[ g_0(x_0, u) + J^*_1(f_0(x_0, u)) \right].
\]
This DP algorithm is computationally intractable for large \( m \). The reason is that the set of possible minimizer choices \( u \) grows exponentially with \( m \), and for each of these choices the value of \( J^*_1(f_0(x_0, u)) \) must be computed.

However, the problem can be solved approximately with multiagent rollout, using a base policy \( \mu = (\mu^1, \ldots, \mu^m) \). Then the number of times \( J^*_1(f_0(x_0, u)) \) needs to be computed is dramatically reduced. This computation is done sequentially, one-agent-at-a-time, as follows:

\[
\tilde{u}^1 \in \arg \min_{u^1 \in U^1} \left[ g_0(x_0, u^1, \mu^2(x_0), \ldots, \mu^m(x_0)) \right.
\]
\[
+ J^*_1 \left( f_0(x_0, u^1, \mu^2(x_0), \ldots, \mu^m(x_0)) \right) \bigg],
\]

\[
\tilde{u}^2 \in \arg \min_{u^2 \in U^2} \left[ g_0(x, \tilde{u}^1, u^2, \mu^3(x_0), \ldots, \mu^m(x_0)) \right.
\]
\[
+ J^*_1 \left( f_0(x, \tilde{u}^1, u^2, \mu^3(x_0), \ldots, \mu^m(x_0)) \right) \bigg],
\]

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

\[
\tilde{u}^m \in \arg \min_{u^m \in U^m} \left[ g_0(x_0, \tilde{u}^1, \tilde{u}^2, \ldots, \tilde{u}^{m-1}, u^m) \right.
\]
\[
+ J^*_1 \left( f_0(x_0, \tilde{u}^1, \tilde{u}^2, \ldots, \tilde{u}^{m-1}, u^m) \right) \bigg].
\]

In this algorithm, the number of times for which \( J^*_1(f_0(x_0, u)) \) must be computed grows linearly with \( m \).

When the number of stages is larger than two, a similar algorithm can be used. Essentially, the one-stage maximizer’s cost function \( J^*_1 \) must be replaced by the optimal cost function of a multistage maximization problem, where the minimizer is constrained to use the base policy (see also the paper [Ber19b], Section 5.4).

An interesting question is how do various algorithms work when approximations are used in the min-max and max-min problems? We can certainly improve the minimizer’s policy assuming a fixed policy for the maximizer. However, it is unclear how to improve both the minimizer’s and the maximizer’s policies simultaneously. In practice, in symmetric games, like chess, a common policy is trained for both players. In particular, in the AlphaZero and TD-Gammon programs this strategy is computationally expedient and has worked well. However, there is no reliable theory to guide the simultaneous training of policies for both maximizer and minimizer, and it is quite plausible that unusual behavior may arise in exceptional cases.†

† Even exact policy iteration methods for Markov games

† Indeed such exceptional cases have been reported for the AlphaGo program in late 2022, when humans defeated an AlphaGo look-alike, KataGo, “by using adversarial techniques that take advantage of KataGo’s blind spots” (according to the reports); see Wang et al. [WGB22].
encounter serious convergence difficulties, and need to be modified for reliable behavior. The author’s paper [Ber21c] and book [Ber22b] (Chapter 5) address these convergence issues with modified versions of the policy iteration method, and give many earlier references.

We finally note another source of difficulty in minimax control: Newton’s method applied to solution of the Bellman equation for minimax problems exhibits more complex behavior than its expected value counterpart. The reason is that the Bellman operator $T$ for infinite horizon problems, given by

$$(TJ)(x) = \min_{u \in U(x)} \max_{w \in W(x,u)} \left[ g(x, u, w) + \alpha J(f(x, u, w)) \right], \quad \text{for all } x,$$

is neither convex nor concave as a function of $J$. To see this, note that the function

$$\max_{w \in W(x,u)} \left[ g(x, u, w) + \alpha J(f(x, u, w)) \right],$$

viewed as a function of $J$ [for fixed $(x, u)$], is convex, and when minimized over $u \in U(x)$, it becomes neither convex nor concave. As a result there are special difficulties in connection with convergence of Newton’s method and the natural form of policy iteration, given by Pollatschek and Avi-Itzhak [PoA69]; see also Chapter 5 of the author’s abstract DP book [Ber22a].

Minimax Control and Zero-Sum Game Theory

Zero-sum game problems are viewed as fundamental in the field of economics, and there is an extensive and time-honored theory around them. In the case where the game involves a dynamic system

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

and a cost function

$$g_k(x_k, u_k, w_k),$$

there are two players, the minimizer choosing $u_k \in U_k(x_k)$, and the maximizer choosing $w_k \in W_k(x_k)$, at each stage $k$. Such zero-sum games involve two minimax control problems:

(a) The min-max problem, where the minimizer chooses a policy first and the maximizer chooses a policy second with knowledge of the minimizer’s policy. The DP algorithm for this problem has the form

$$J^*_N(x_N) = g_N(x_N),$$

$$J^*_k(x_k) = \min_{u_k \in U_k(x_k)} \max_{w_k \in W_k(x_k)} \left[ g_k(x_k, u_k, w_k) + J^*_{k+1}(f_k(x_k, u_k, w_k)) \right].$$
(b) The max-min problem, where the maximizer chooses policy first and the minimizer chooses policy second with knowledge of the maximizer’s policy. The DP algorithm for this problem has the form

\[ \hat{J}_N(x_N) = g_N(x_N), \]

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

A basic and easily seen fact is that

\[
\text{Max-Min optimal value} \leq \text{Min-Max optimal value}.
\]

Game theory is particularly interested on conditions that guarantee that

\[
\text{Max-Min optimal value} = \text{Min-Max optimal value}. \quad (2.99)
\]

However, this question is of limited interest in engineering contexts that involve worst case design. Moreover, the validity of the minimax equality (2.99) is beyond the range of practical RL. This is so primarily because once approximations are introduced, the delicate assumptions that guarantee this equality are disrupted.

### 2.13 NOTES, SOURCES, AND EXERCISES

**Section 2.1:** In this chapter, we have placed emphasis on finite horizon problems, possibly involving a nonstationary system and cost per stage. However, the insights that can be obtained from the infinite horizon/stationary context fully apply. These include the interpretation of approximation in value space as a Newton step, and of rollout as a single step of the policy iteration method. The reason is that an \( N \)-step finite horizon/nonstationary problem can be converted to an infinite horizon/stationary problem with a termination state to which the system moves at the \( N \)th stage; see Section 1.6.2.

**Section 2.2:** Approximation in value space has been considered in an ad hoc manner since the early days of DP, motivated by the curse of dimensionality. Moreover, the idea of \( \ell \)-step lookahead minimization with horizon truncation beyond the \( \ell \) steps has a long history and is often referred to as “rolling horizon” or “receding horizon” optimization. Approximation in value space was reframed in the late 80s and was coupled with model-free simulation methods that originated in artificial intelligence.

**Section 2.3:** The main idea of rollout algorithms, obtaining an improved policy starting from some other suboptimal policy, has appeared in several DP contexts, including games; see e.g., Abramson [Abr90], and Tesauro
and Galperin [TeG96]. The name “rollout” was coined by Tesauro [TeG96] in the context of backgammon; see Example 2.7.3. The use of the name “rollout” has gradually expanded beyond its original context; for example, the samples collected through trajectory simulation are referred to as “rollouts” by some authors.

In the present notes, we will adopt the original intended meaning: rollout is an algorithm that provides policy improvement starting from a base policy, which is evaluated with some form of Monte Carlo simulation, perhaps augmented by some other calculation that may include a terminal cost function approximation. The author’s rollout book [Ber20a] provides a more extensive discussion.

There has been a lot of research on rollout algorithms, which we list selectively in chronological order: Christodouleas [Chr97], Bertsekas and Castaño [BeC99], Duin and Voss [DuV99], Secomandi [Sec00], [Sec01], [Sec03], Ferris and Voelker [FeV02], [FeV04], McGovern, Moss, and Barto [MMB02], Savagaonkar, Givan, and Chong [SGC02], Wu, Chong, and Givan [WCG02], [WCG03], Bertsimas and Popescu [BeP03], Guerriero and Mancini [GuM03], Tu and Pattipati [TuP03], Meloni, Pacciarelli, and Pranzo [MP04], Yan et al. [YDR04], Han, Lai, and Spivakovsky [HLS06], Besse and Chaib-draa [BeC08], Sun et al. [SZL08], Mishra et al. [MCT10], Bertazzi et al. [BBG13], Sun et al. [SLJ13], Tesauro et al. [TGL13], Antunes and Heemels [AnH14], Beyme and Leung [BeL14], Goodson, Thomas, and Ohlmann [GTO15], [GTO17], Khashoeei, Antunes, and Heemels [KAH15], Li and Womer [LiW15], Mastin and Jaillet [MaJ15], Huang, Jia, and Guan [HJG16], Simroth, Holfeld, and Brunsch [SHB15], Lan, Guan, and Wu [LGW16], Lam, Willcox, and Wolpert [LWW16], Gommans et al. [GTA17], Lam and Willcox [LaW17], Ulmer [Ulm17], Bertazzi and Secomandi [BeS18], Zhang, Ohlmann, and Thomas [ZOT18], Sarkale et al. [SNC18], Ulmer et al. [UGM18], Arcari, Hewing, and Zeilinger [AHZ19], Chu, Xu, and Li [CXL19], Goodson, Bertazzi, and Levary [GBL19], Guerriero, Di Puglia, and Macrina [GDM19], Ho, Liu, and Zabinsky [HLZ19], Liu et al. [LLL19], Nozhati et al. [NSE19], Singh and Kumar [SiK19], Yu et al. [YYM19], Yuanhong [Yua19], Andersen, Stidsen, and Reinhardt [ASR20], Durasevic and Jakobovic [DuJ20], Issakkmuthu, Fern, and Tadepalli [IFT20], Lee et al. [LEC20], Li et al. [LZS20], Lee [Lee20], Montenegro et al. [MLM20], Meshram and Kaza [MeK20], Schope, Driessen, and Yarovoy [SDY20], Yan, Wang, and Xu [YWX20], Yue and Kontar [YuK20], Zhang, Kafouros, and Yu [ZKY20], Hoffman et al. [HCR21], Houy and Flagg [HoF21], Li, Krakow, and Gopalswamy [LKG21], Liu et al. [LPS21], Nozhati [Noz21], Rimelé et al. [RGG21], Tuncel et al. [TBP21], Xie, Li, and Xu [XLX21], Bertsekas [Ber22d], Paulson, Sonoufar, and Chakrabarty [PSC22], Bai et al. [BLJ23], Rusmevichientong et al. [RST23].

These references collectively include a large number of computational studies, discuss variants and problem-specific adaptations of rollout algorithms for a broad variety of practical problems, and consistently report
favorable computational experience. The size of the cost improvement over
the base policy is often impressive, evidently owing to the fast convergence
rate of Newton’s method that underlies rollout. Moreover these works
illustrate some of the other important advantages of rollout: reliability,
simplicity, suitability for on-line replanning, and the ability to interface
with other RL techniques, such as neural network training, which can be
used to provide suitable base policies and/or approximations to their cost
functions.

The adaptation of rollout algorithms to discrete deterministic opti-
mization problems, the notions of sequential consistency, sequential im-
provement, fortified rollout, and the use of multiple heuristics for parallel
rollout were first given in the paper by Bertsekas, Tsitsiklis, and Wu
[BTW97], and were also discussed in the neuro-dynamic programming book
[BeT96]. Rollout algorithms for stochastic problems were further formal-
ized in the papers by Bertsekas [Ber97b], and Bertsekas and Castaño
[BeC99]. Extensions to constrained rollout were first given in the author’s
papers [Ber05a], [Ber05b]. A survey of rollout in discrete optimization was
given by the author in [Ber13a].

The model-free rollout algorithm, in the form given here, was first dis-
cussed in the RL book [Ber19a]. It is inspired by the method of comparison
training, proposed by Tesauro [Tes89a], [Tes89b], [Tes01], and subsequently
used by several other authors (see [DNW16], [TCW19]). This is a general
method for training an approximation architecture to choose between two
alternatives, using a dataset of expert choices in place of an explicit cost
function.

Section 2.4: Our discussion of rollout, iterative deepening, and pruning
in the context of multistep approximation in value space for deterministic
problems contains some original ideas particularly in connection with the
incremental multistep rollout algorithms.

Section 2.5: Constrained forms of rollout were introduced in the author’s
papers [Ber05a] and [Ber05b]. The paper [Ber05a] also discusses rollout
and approximation in value space for stochastic problems in the context of
so-called restricted structure policies. The idea here is to simplify the prob-
lem by selectively restricting the information and/or the controls available
to the controller, thereby obtaining a restricted but more tractable problem
structure, which can be used conveniently in a one-step lookahead context.
An example of such a structure is one where fewer observations are ob-
tained, or one where the control constraint set is restricted to a single or a
small number of given controls at each state.

Section 2.6: Rollout for continuous-time optimal control was first dis-
cussed in the author’s rollout book [Ber20a]. A related discussion of policy
iteration, including the motivation for approximating the gradient of the
optimal cost-to-go $\nabla_x J_t$ rather than the optimal cost-to-go $J_t$, has been
given in Section 6.11 of the neuro-dynamic programming book [BeT96].
This discussion also includes the use of value and policy networks for approximate policy evaluation and policy improvement for continuous-time optimal control. The underlying ideas have long historical roots, which are recounted in detail in the book [BeT96].

**Section 2.7:** The idea of the certainty equivalence approximation in the context of rollout for stochastic systems (Section 2.7.3) was proposed in the paper by Bertsekas and Castaño [BeC99], together with extensive empirical justification. However, the associated theoretical insight into this idea was established more recently, through the interpretation of approximation in value space as a Newton step, which suggests that the lookahead minimization after the first step can be approximated with small degradation of performance.

The idea of variance reduction in the context of rollout (Section 2.7.4) was proposed by the author in the paper [Ber97b]; see also the DP textbook [Ber17a], Section 6.5.2. The paper by Chang, Hu, Fu, and Marcus [CHF05], and the 2007 first edition of their monograph proposed and analyzed adaptive sampling in connection with DP, as well as early forms of Monte Carlo tree search, including statistical tests to control the sampling process (a second edition, [CHF13], appeared in 2013). The name “Monte Carlo tree search” has become popular, and in its current use, it encompasses a variety of methods that involve adaptive sampling, rollout, and extensions to sequential games. We refer to the papers by Coulom [Cou06], and Chang et al. [CHF13], the discussion by Fu [Fu17], and the survey by Browne et al. [BPW12].

Statistical tests for adaptive sampling has been inspired by works on multiarmed bandit problems; see Lai and Robbins [LaR85], Agrawal [Agr95], Burnetas and Katehakis [BuK97], Meuleau and Bourgine [MeB99], Auer, Cesa-Bianchi, and Fischer [ACF02], Kocsis and Szepesvari [KoS06], Dimitrakakis and Lagoudakis [DiL08], Audibert, Munos, and Szepesvari [AMS09], and Munos [Mun14]. The book by Lattimore and Szepesvari [LaS20] focuses on multiarmed bandit methods, and provides an extensive account of the UCB rule.

Adaptive sampling and MCTS may be viewed within the context of a broader class of on-line lookahead minimization techniques, sometimes called on-line search methods. These techniques are based on a variety of ideas, such as random search and intelligent pruning of the lookahead tree. One may naturally combine them with approximation in value space and (possibly) rollout, although it is not necessary to do so (the multistep minimization horizon may extend to the terminal time $N$). For representative works, some of which apply to continuous spaces problems, including POMDP, see Hansen and Zilberstein [HaZ01], Kearns, Mansour, and Ng [KMN02], Peret and Garcia [PeG04], Ross et al. [RPP08], Silver and Veness [SiV10], Hostetler, Fern, and Dietterich [HFD17], and Ye et al. [YSH17].
The multistep lookahead approximation ideas of Section 2.4 may also be viewed within the context of on-line search methods.

Another rollout idea for stochastic problems, which we have not discussed in these notes, is the open-loop feedback controller (OLFC), a suboptimal control scheme that dates to the 60s; see Dreyfus [Dre65]. The OLFC applies to POMDP as well, and uses an open-loop optimization over the future evolution of the system. In particular, it uses the current information vector $I_k$ to determine the belief state $b_k$. It then solves the open-loop problem of minimizing

$$E \left\{ g_N(x_N) + \sum_{i=k}^{N-1} g_i(x_i, u_i, w_i) \mid I_k \right\}$$

subject to the constraints

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

and applies the first control $\pi_k$ in the optimal open-loop control sequence $\{\pi_k, \pi_{k+1}, \ldots, \pi_{N-1}\}$. It is easily seen that the OLFC is a rollout algorithm that uses as base policy the optimal open-loop policy for the problem (the one that ignores any state or observation feedback).

For a detailed discussion of the OLFC, we refer to the author’s survey paper [Ber05a] (Section 4) and DP textbook [Ber17a] (Section 6.4.4). The survey [Ber05a] discusses also a generalization of the OLFC, called partial open-loop-feedback-control, which calculates the control input on the basis that some (but not necessarily all) of the observations will in fact be taken in the future, and the remaining observations will not be taken. This method often allows one to deal with those observations that are troublesome and complicate the solution, while taking into account the future availability of other observations that can be reasonably dealt with.

A computational case study for hydrothermal power system scheduling is given by Martinez and Soares [MaS02]. A variant of the OLFC, which also applies to minimax control problems, is given in the author’s paper [Ber72b], together with a proof of a cost improvement property over the optimal open-loop policy.

Section 2.8: The role of stochastic programming in providing a link between stochastic DP and continuous spaces deterministic optimization (cf. Section 2.8) is well known; see the texts by Birge and Louveaux [BiL97], Kall and Wallace [KaW94], and Prekopa [Pre95], and the survey by Ruszczyński and Shapiro [RuS03]. Stochastic programming has been applied widely, and there is much to be gained from its combination with RL. The material of this section comes from the author’s rollout book [Ber20a].

Section 2.9: The multiagent rollout algorithm was proposed in the author’s papers [Ber19c], [Ber20b]. The paper [Ber21a] provides an extensive overview of this research. See also the notes and sources for Chapter 1.
Section 2.10: The material on rollout for Bayesian optimization and sequential estimation comes from a recent paper by the author [Ber22d]. This paper is also the basis for the adaptive control material of Section 2.11, and has been included in the book [Ber22a]. The paper by Bhambr, Bhattacharjee, and Bertsekas [BBB22] discusses this material for the case of a deterministic system, applies rollout to sequential decoding in the context of the challenging Wordle puzzle, and provides an implementation using some popular base heuristics, with performance that is very close to optimal. For related work see Loxley and Cheung [LoC23].

Section 2.11: The POMDP framework for adaptive control dates to the 60s, and has stimulated substantial theoretical investigations; see Mandl [Man74], Doshi and Shreve [DoS80], Kumar and Lin [KuL82], and the survey by Kumar [Kum85]. Some of the pitfalls of performing parameter identification while simultaneously applying adaptive control have been described by Borkar and Varaiya [BoV79], and by Kumar [Kum83]; see [Ber17a], Section 6.8 for a related discussion.

Section 2.12: The treatment of sequential minimax problems by DP (cf. Section 2.12) has a long history. For some early influential works, see Blackwell and Girshick [BlG54], Shapley [Sha53], and Witsenhausen [Wit66]. In minimax control problems, the maximizer is assumed to make choices with perfect knowledge of the minimizer’s policy. If the roles of maximizer and minimizer are reversed, i.e., the maximizer has a policy (a sequence of functions of the current state) and the minimizer makes choices with perfect knowledge of that policy, the minimizer gains an advantage, the problem may genuinely change, and the optimal value may be reduced. Thus “min-max” and “max-min” are generally two different problems. In classical two-person zero-sum game theory, however, the main focus is on situations where the min-max and max-min are equal. By contrast, in engineering worst case design contexts, the min-max and max-min values are typically unequal.

There is substantial literature on sequential zero-sum games in the context of DP, often called Markov games. The classical paper by Shapley [Sha53] addresses discounted infinite horizon games. A PI algorithm for finite-state Markov games was proposed by Pollatschek and Avi-Itzhak [PoA69], and was interpreted as a Newton method for solving the associated Bellman equation. They have also shown that the algorithm may not converge to the optimal cost function. Computational studies have verified that the Pollatschek and Avi-Itzhak algorithm converges much faster than its competitors, when it converges (see Breton et al. [BFH86], and also Filar and Tolwinski [FiT91], who proposed a modification of the algorithm). Related methods have been discussed for Markov games by van der Wal [Van78] and Tolwinski [Tol89]. The paper by Raghavan and Filar [RaF91], and the textbook by Filar and Vrieze [FiV96] provide extensive surveys of the research up to that time.
The paper by Yu [Yu14] provides an analysis of stochastic shortest path games, where the termination state may not be reachable under some policies, following the earlier paper by Patek and Bertsekas [PaB99]. The paper [Yu14] also includes a rigorous analysis of the Q-learning algorithm for stochastic shortest path games (without any cost function approximation). The papers by Perolat et al. [PSP15], [PPG16], and the survey by Zhang, Yang, and Basar [ZYB21] discuss alternative RL methods for games. The author’s paper [Ber19b] develops VI, PI, and Dijkstra-like finitely terminating algorithms for exact solution of shortest path minimax problems. It also discusses related rollout algorithms for approximate solution.

The author’s paper [Ber21b] has explained the reason behind the unreliable behavior of the Pollatschek and Avi-Itzhak algorithm, based on the Newton step interpretation of PI given in Chapter 1: in the case of Markov games, the Bellman operator does not have the concavity property that is typical of one-player games. This paper has also provided a modified algorithm with solid convergence properties under a totally asynchronous implementation, which applies to very general types of sequential zero-sum games and minimax control. Related aggregation-based RL algorithms were also given. The algorithms, variations, and analysis of the paper [Ber21b] were incorporated as Chapter 5 in the 3rd edition of the abstract DP book [Ber22b].
2.1 (A Traveling Salesman Rollout Example with a Sequentially Improving Heuristic)

Consider the traveling salesman problem of Example 1.2.3 and Fig. 1.2.11, and the rollout algorithm starting from city A.

(a) Assume that the base heuristic is chosen to be the farthest neighbor heuristic, which completes a partial tour by successively moving to the farthest neighbor city not visited thus far. Show that this base heuristic is sequentially consistent. What are the tours produced by this base heuristic and the corresponding rollout algorithm? \textit{Answer:} The base heuristic will produce the tour $A \rightarrow AD \rightarrow ADB \rightarrow ADBC \rightarrow A$ with cost 45. The rollout algorithm will produce the tour $A \rightarrow AB \rightarrow ABD \rightarrow ABDC \rightarrow A$ with cost 13.

(b) Assume that the base heuristic at city A is the nearest neighbor heuristic, while at the partial tours AB, AC, and AD it is the farthest neighbor heuristic. Show that this base heuristic is sequentially improving but not sequentially consistent. Compute the final tour generated by rollout.

\textbf{Solution of part (b):} Clearly the base heuristic is not sequentially consistent, since from A it generates

$A \rightarrow AC \rightarrow ACD \rightarrow ACDB \rightarrow A$,

but from AC it generates

$AC \rightarrow ACB \rightarrow ACBD \rightarrow A$.

However, it is seen that the sequential improvement criterion (2.15) holds at each of the states A, AB, AC, and AD (and also trivially for the remaining states).

The base heuristic at A is the nearest neighbor heuristic so it generates

$A \rightarrow AC \rightarrow ACD \rightarrow ACDB \rightarrow A$ with cost 28.

The rollout algorithm at state A looks at the three successor states AB, AC, and AD, and runs the farthest neighbor heuristic from each, and generates:

$A \rightarrow AB \rightarrow ABD \rightarrow ABDC \rightarrow A$ with cost 13,

$A \rightarrow AC \rightarrow ACB \rightarrow ACBD \rightarrow A$ with cost 45,

$A \rightarrow AD \rightarrow ADB \rightarrow ADBC \rightarrow A$ with cost 45,

so the rollout algorithm will move from A to AB.

Then the rollout algorithm looks at the two successor states ABC, ABD, and runs the base heuristic (whatever that may be; it does not matter) from each. The paths generated are:

$AB \rightarrow ABC \rightarrow ABCD \rightarrow A$ with cost 26.
AB→ABD→ABDC→A with cost 13,
so the rollout algorithm will move from AB to ABD.
Thus the final tour generated by the rollout algorithm is
A→AB→ABD→ABDC→A, with cost 13.

2.2 (A Generic Example of a Base Heuristic that is not Sequentially Improving)

Consider a rollout algorithm for a deterministic problem with a base heuristic
that produces an optimal control sequence at the initial state \( x_0 \), and uses the
(optimal) first control \( u_0 \) of this sequence to move to the (optimal) next state
\( x_1 \). Suppose that the base heuristic produces a strictly suboptimal sequence from
every successor state \( x_2 = f_1(x_1, u_1), u_1 \in U_1(x_1) \), so that the rollout yields a
control \( u_1 \) that is strictly suboptimal. Show that the trajectory produced by the
rollout algorithm starting from the initial state \( x_0 \) is strictly inferior to the one
produced by the base heuristic starting from \( x_0 \), while the sequential improvement
condition does not hold.

2.3 (Computational Exercise - Parking with Problem Approximation and Rollout)

In this computational exercise we consider a more complex, imperfect state informa-
tion version of the one-directional parking problem of Example 1.6.1. Recall
that in this problem a driver is looking for a free parking space in an area con-
sisting of \( N \) spaces arranged in a line, with a garage at the end of the line (space \( N \)). The driver starts at space 0 and traverses the parking spaces sequentially,
i.e., from each space he/she goes to the next space, up to when he/she decides to
park in space \( k \) at cost \( c(k) \), if space \( k \) is free. Upon reaching the garage, parking
is mandatory at cost \( C \).

In Example 1.6.1, we assumed that the driver knows the probabilities \( p(k + 1), \ldots, p(N - 1) \) of the parking spaces \( (k + 1), \ldots, (N - 1) \), respectively, being
free. Under this assumption, the state at stage \( k \) is either the termination state \( t \)
(if already parked), or it is \( F \) (location \( k \) free), or it is \( \overline{F} \) (location \( k \) taken), and the
DP algorithm has the form

\[
J^*_k(F) = \begin{cases} 
\min \left[ c(k), p(k + 1)J^*_{k+1}(F) + (1 - p(k + 1))J^*_{k+1}(\overline{F}) \right] & \text{if } k < N - 1, \\
\min \left[ c(N - 1), C \right] & \text{if } k = N - 1,
\end{cases}
\]

(2.100)

\[
J^*_k(\overline{F}) = \begin{cases} 
p(k + 1)J^*_{k+1}(F) + (1 - p(k + 1))J^*_{k+1}(\overline{F}) & \text{if } k < N - 1, \\
C & \text{if } k = N - 1,
\end{cases}
\]

(2.101)

for the states other than the termination state \( t \), while for \( t \) we have \( J^*_k(t) = 0 \)
for all \( k \).

We will now consider the more complex variant of the problem where the
probabilities \( p(0), \ldots, p(N - 1) \) do not change over time, but are unknown to
the driver, so that he/she cannot use the exact DP algorithm (2.100)-(2.101). Instead, the driver considers a one-step lookahead approximation in value space scheme, which uses empirical estimates of these probabilities that are based on the ratio $\frac{f_k}{k+1}$, where $f_k$ is the number of free spaces seen up to space $k$, after the free/taken status of spaces $0, \ldots, k$ has been observed. In particular, these empirical estimates are given by

$$b_k(m, f_k) = \gamma \bar{p}(m) + (1 - \gamma) \frac{f_k}{k+1}, \quad m = k + 1, \ldots, N - 1,$$  \hspace{1cm} (2.102)

where $f_k$ is the number of free spaces seen up to space $k$, and $\gamma$ and $\bar{p}(m)$ are fixed numbers between 0 and 1. Of course the values $f_k$ observed by the driver evolve according to the true (and unknown) probabilities $p(0), \ldots, p(N - 1)$ according to $f_{k+1} = \begin{cases} f_k + 1 & \text{with probability } p(k + 1), \\ f_k & \text{with probability } 1 - p(k + 1). \end{cases}$ \hspace{1cm} (2.103)

For the solution of this exercise you may assume any reasonable values you wish for $N$, $p(m)$, $\bar{p}(m)$, and $\gamma$. Recommended values are $N \geq 100$, and probabilities $p(m)$ and $\bar{p}(m)$ that are nonincreasing with $m$.

The decision made by the approximation in value space scheme is to park at space $k$ if and only if it is free and in addition

$$c(k) \leq b_k(k + 1, f_k) \hat{J}_{k+1}(F) + (1 - b_k(k + 1, f_k)) \bar{J}_{k+1}(\overline{F}),$$  \hspace{1cm} (2.104)

where $\hat{J}_{k+1}(F)$ and $\bar{J}_{k+1}(\overline{F})$ are the cost-to-go approximations from stage $k+1$. Consider the following two different methods to compute $\hat{J}_{k+1}(F)$ and $\bar{J}_{k+1}(\overline{F})$ for use in Eq. (2.104):

(1) Here the approximate cost function values $\hat{J}_{k+1}(F)$ and $\bar{J}_{k+1}(\overline{F})$ are obtained by using problem approximation, whereby at time $k$ it is assumed that the probabilities of free/taken status at the future spaces $m = k + 1, \ldots, N - 1$ are $b_k(m, f_k)$, $m = k + 1, \ldots, N - 1$, as given by Eq. (2.102).

More specifically, $\hat{J}_{k+1}(F)$ and $\bar{J}_{k+1}(\overline{F})$ are obtained by solving optimally the problem whereby we use the probabilities $b_k(m, f_k)$ of Eq. (2.102) in place of the unknown $p(m)$ in the DP algorithm (2.100)-(2.101):

$$\hat{J}_{k+1}(F) = \hat{J}_{k+1}(F), \quad \hat{J}_{k+1}(\overline{F}) = \hat{J}_{k+1}(\overline{F}),$$

where $\hat{J}_{k+1}(F)$ and $\hat{J}_{k+1}(\overline{F})$ are given at the last step of the DP algorithm

$$\hat{J}_{N-1}(F) = \min \left[c(N - 1), C\right], \quad \hat{J}_{N-1}(\overline{F}) = C,$$

$$\hat{J}_m(F) = \min \left[c(m), b_k(m + 1, f_k) \hat{J}_{m+1}(F) + (1 - b_k(m + 1, f_k)) \bar{J}_{m+1}(\overline{F}) \right],$$

if $k < m < N - 1$,

$$\hat{J}_m(\overline{F}) = b_k(m + 1, f_k) \hat{J}_{m+1}(F) + (1 - b_k(m + 1, f_k)) \bar{J}_{m+1}(\overline{F}),$$

if $k < m < N - 1$. 

145
Here for each \( k \), the approximate cost function values \( \tilde{J}_{k+1}(F) \) and \( \tilde{J}_{k+1}(\overline{F}) \)
are obtained by using rollout with a greedy base heuristic (park as soon as possible), and Monte Carlo simulation. In particular, according to this greedy heuristic, we have \( \tilde{J}_{k+1}(F) = c(k+1) \). To compute \( \tilde{J}_{k+1}(\overline{F}) \) we generate many random trajectories by running the greedy heuristic forward from space \( k+1 \) assuming the probabilities \( b_k(m+1, f_k) \) of Eq. (2.102) in place of the unknown \( p(m+1) \), \( m = k+1, \ldots, N-1 \), and we average the cost results obtained.

(a) Use Monte Carlo simulation to compute the expected cost from spaces \( 0, \ldots, N-1 \), when using each of the two schemes (1) and (2).

(b) Compare the performance of the schemes of part (a) with the following:

(i) The optimal expected costs \( J^*_k(F) \) and \( J^*_k(\overline{F}) \) from \( k = 0, \ldots, N-1 \), using the DP algorithm (2.100)-(2.101), and the probabilities \( p(m) \), \( m = 0, \ldots, N-1 \), that you used for the random generation of the numbers of free spaces \( f_k \) [cf. Eq. (2.103)].

(ii) The expected costs \( \hat{J}_k(F) \) and \( \hat{J}_k(\overline{F}) \) from \( k = 0, \ldots, N-1 \) that are attained by using the greedy base heuristic. Argue that these are given by

\[
\hat{J}_k(F) = c(k), \quad k = 0, \ldots, N-1,
\]

\[
\hat{J}_k(\overline{F}) = p(k+1)c(k+1) + (1-p(k+1))\hat{J}_{k+1}(\overline{F}), \quad k = 0, \ldots, N-2,
\]

\[
\hat{J}_{N-1}(\overline{F}) = C.
\]

(c) Argue that scheme (1) becomes superior to scheme (2) in terms of cost attained as \( \gamma \approx 1 \) and \( \bar{p}(m) \approx p(m) \). Are your computational results in rough agreement with this assertion?

(d) Argue that as \( \gamma \approx 0 \) and \( N >> 1 \), scheme (1) becomes superior to scheme (2) in terms of cost attained from parking spaces \( k >> 1 \).

(e) What happens if the probabilities \( p(m) \) do not change much with \( m \)?