4

Sequential Problems

Mykel J. Kochenderfer

The previous chapter discussed problems in which a single decision is to be made, but many important problems require the decision maker to make a series of decisions. The same principle of maximum expected utility still applies, but optimal decision making requires reasoning about future sequences of actions and observations. This chapter will discuss sequential decision problems in stochastic environments.

4.1 Formulation

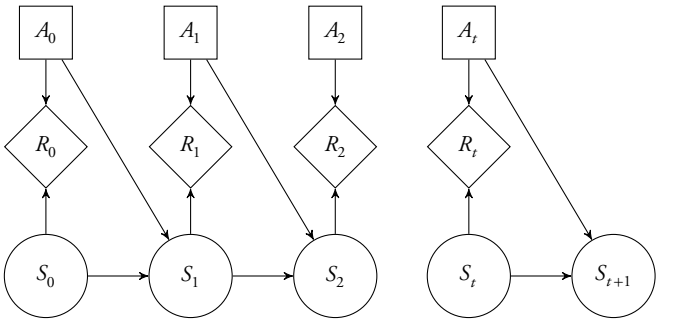
In this chapter, we will focus on a general formulation of sequential decision problems under the assumption that the model is known and that the environment is fully observable. Both of these assumptions will be relaxed in the next two chapters.

4.1.1 Markov Decision Processes

In a *Markov decision process* (MDP), an agent chooses action a_t at time t based on observing state s_t. The agent then receives a reward r_t. The state evolves probabilistically based on the current state and action taken by the agent. The assumption that the next state depends only on the current state and action and not on any prior state or action is known as the *Markov assumption*.

An MDP can be represented using a decision network as shown in Figure 4.1a. There are information edges (not shown in the figure) from $A_{0:t-1}$ and $S_{0:t}$ to A_t. The utility function is decomposed into rewards $R_{0:t}$.

We will focus on *stationary* MDPs in which $P(S_{t+1} \mid S_t, A_t)$ and $P(R_t \mid A_t, S_t)$ do not vary with time. Stationary MDPs can be compactly represented by a dynamic decision diagram as shown in Figure 4.1b. The *state transition function* $T(s' \mid s, a)$ represents the probability of transitioning from state s to s' after executing action a. The *reward function* $R(s, a)$ represents the expected reward received when executing



(a) General representation. (b) Stationary representation.

Figure 4.1 Markov decision process decision diagram.

action a from state s. We assume that the reward function is a deterministic function of s and a, but this need not be the case.

The problem of aircraft collision avoidance can be formulated as an MDP. The states represent the positions and velocities of our aircraft and the intruder aircraft, and the actions represent whether we climb, descend, or stay level. We receive a large negative reward for colliding with the other aircraft and a small negative reward for climbing or descending.

4.1.2 Utility and Reward

The rewards in an MDP are treated as components in an additively decomposed utility function (Section 3.1.6). In a *finite horizon* problem with n decisions, the utility associated with a sequence of rewards $r_{0:n-1}$ is simply

$$\sum_{t=0}^{n-1} r_t. \tag{4.1}$$

In an *infinite horizon* problem in which the number of decisions is unbounded, the sum of rewards can become infinite. Suppose strategy A results in a reward of 1 per time step and strategy B results in a reward of 100 per time step. Intuitively, a rational agent should prefer strategy B over strategy A, but both provide the same infinite expected utility.

There are several ways to define utility in terms of individual rewards in infinite horizon problems. One way is to impose a *discount factor* γ between 0 and 1. The utility

is given by

$$\sum_{t=0}^{\infty} \gamma^t r_t. \tag{4.2}$$

So long as $0 \leq \gamma < 1$ and the rewards are finite, the utility will be finite. The discount factor makes it so that rewards in the present is worth more than rewards in the future, a concept that also appears in economics.

Another way to define utility in infinite horizon problems is to use the *average reward* given by

$$\lim_{n \to \infty} \frac{1}{n} \sum_{t=0}^{n-1} r_t. \tag{4.3}$$

This book focuses primarily on optimizing with respect to discounted rewards over an infinite horizon.

4.2 Dynamic Programming

The optimal strategy can be found by using a computational technique called *dynamic programming*. Although we will focus on dynamic programming algorithms for MDPs, dynamic programming is a general technique that can be applied to a wide variety of other problems. For example, dynamic programming can be used in computing a Fibonacci sequence, finding the longest common subsequence between two strings, and finding the most likely sequence of states in a hidden Markov model. In general, algorithms that use dynamic programming for solving MDPs are much more efficient than brute force methods.

4.2.1 Policies and Utilities

A *policy* in an MDP determines what action to select given the past history of states and actions. The action to select at time t, given the history $h_t = (s_{0:t}, a_{0:t-1})$, is written $\pi_t(h_t)$. Because the future state sequence and rewards depend only on the current state and action (as made apparent in the conditional independence assumptions in Figure 4.1a), we can restrict our attention to policies that depend only on the current state.

In infinite horizon MDPs in which the transitions and rewards are stationary, we can further restrict our attention to stationary policies. We will write the action associated with stationary policy π in state s as $\pi(s)$, without the temporal subscript. In finite horizon problems, however, it may be beneficial to select a different action depending on how many time steps are remaining. For example, when playing basketball, it is generally not a good strategy to attempt a half-court shot unless there are just a couple seconds remaining in the game.

The expected utility of executing π from state s is denoted $U^\pi(s)$. In the context of MDPs, U^π is often referred to as the *value function*. An *optimal policy* π^* is a policy that maximizes expected utility:

$$\pi^*(s) = \arg\max_\pi U^\pi(s) \tag{4.4}$$

for all states s. Depending on the model, there may be multiple policies that are optimal.

4.2.2 Policy Evaluation

Computing the expected utility obtained from executing a policy is known as *policy evaluation*. We may use dynamic programming to evaluate the utility of a policy π for t steps. If we do not execute the policy at all, then $U_0^\pi(s) = 0$. If we execute the policy for one step, then $U_1^\pi(s) = R(s, \pi(s))$, which is simply the expected reward associated with the first step.

Suppose we know the utility associated with executing π for $t-1$ steps. Computing the utility associated with executing π for t steps can be computed as follows:

$$U_t^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s' \mid s, \pi(s)) U_{t-1}^\pi(s'), \tag{4.5}$$

where γ is the discount factor, which can be set to 1 if discounting is not desired. Algorithm 4.1 shows how to iteratively compute the expected utility of a policy up to an arbitrary horizon n.

Algorithm 4.1 Iterative policy evaluation

```
function IterativePolicyEvaluation(π, n)
    U_0^π(s) ← 0 for all s
    for t ← 1 to n
        U_t(s) ← R(s, π(s)) + γ Σ_{s'} T(s' | s, π(s)) U_{t-1}^π(s') for all s
    return U_n
```

For an infinite horizon with discounted rewards,

$$U^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s' \mid s, \pi(s)) U^\pi(s'). \tag{4.6}$$

We can compute U^π arbitrarily well with enough iterations of iterative policy evaluation. An alternative is to solve a system of n linear equations where n is the number of states. We can represent the system of equations in matrix form:

$$\mathbf{U}^\pi = \mathbf{R}^\pi + \gamma \mathbf{T}^\pi \mathbf{U}^\pi, \tag{4.7}$$

where $\mathbf{U}^\pi$ and $\mathbf{R}^\pi$ are the utility and reward functions represented as an n-dimensional vector. The $n \times n$ matrix $\mathbf{T}^\pi$ contains state transition probabilities. The probability of transitioning from the ith state to the jth state is given by $\mathbf{T}^\pi_{ij}$.

We can easily solve for U^π as follows:

$$\mathbf{U}^\pi - \gamma \mathbf{T}^\pi \mathbf{U}^\pi = \mathbf{R}^\pi \tag{4.8}$$

$$(\mathbf{I} - \gamma \mathbf{T}^\pi)\mathbf{U}^\pi = \mathbf{R}^\pi \tag{4.9}$$

$$\mathbf{U}^\pi = (\mathbf{I} - \gamma \mathbf{T}^\pi)^{-1}\mathbf{R}^\pi. \tag{4.10}$$

Solving for $\mathbf{U}^\pi$ in this way requires $O(n^3)$ time.

4.2.3 Policy Iteration

Policy evaluation can be used in a general process called *policy iteration* for computing an optimal policy π^* as outlined in Algorithm 4.2. Policy iteration starts with any policy π_0 and iterates the following two steps:

- *Policy evaluation.* Given the current policy π_k, compute U^{π_k}.
- *Policy improvement.* Using U^{π_k}, compute a new policy using the equation in Line 5.

The algorithm terminates when there is no more improvement. Because every step leads to improvement and there are finitely many policies, the algorithm terminates with an optimal solution.

Algorithm 4.2 Policy iteration

```
1: function PolicyIteration(π_0)
2:     k ← 0
3:     repeat
4:         Compute U^{π_k}
5:         π_{k+1}(s) = arg max_a (R(s,a) + γ Σ_{s'} T(s' | s,a) U^{π_k}(s')) for all states s
6:         k ← k + 1
7:     until π_k = π_{k−1}
8:     return π_k
```

There are many variants of policy iteration. One method known as *modified policy iteration* involves approximating U^{π_k} using only a few iterations of iterative policy evaluation instead of computing the utility function exactly.

4.2.4 Value Iteration

An alternative to policy iteration is *value iteration* (Algorithm 4.3), which is often used because it is simple and easy to implement. First, let us compute the optimal value

function U_n associated with a horizon of n and no discounting. If $n = 0$, then $U_0(s) = 0$ for all s. We can compute U_n recursively from this base case

$$U_n(s) = \max_a \left(R(s,a) + \sum_{s'} T(s' \mid s,a) U_{n-1}(s') \right). \tag{4.11}$$

For an infinite horizon problem with discount γ, it can be proven that the value of an optimal policy satisfies the *Bellman equation*:

$$U^*(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U^*(s') \right). \tag{4.12}$$

The optimal value function U^* appears on both sides of the equation. Value iteration approximates U^* by iteratively updating the estimate of U^* using Equation (4.12). Once we know U^*, we can extract an optimal policy by using

$$\pi(s) \leftarrow \arg\max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U^*(s') \right). \tag{4.13}$$

Algorithm 4.3 Value iteration

```
function ValueIteration
    k ← 0
    U_0(s) ← 0 for all states s
    repeat
        U_{k+1}(s) ← max_a [R(s,a) + γ Σ_{s'} T(s' | s,a) U_k(s')] for all states s
        k ← k + 1
    until convergence
    return U_k
```

Algorithm 4.3 shows U_0 being initialized to 0, but value iteration can be proven to converge with any bounded initialization (i.e., $|U_0(s)| < \infty$ for all s). It is common to initialize the utility function to a guess of the optimal value function in an attempt to speed convergence.

A common termination condition for the loop in Algorithm 4.3 is when $||U_k - U_{k-1}|| < \delta$. In this context, $||\cdot||$ denotes the *max norm*, where $||U|| = \max_s |U(s)|$. The quantity $||U_k - U_{k-1}||$ is known as the *Bellman residual*.

If we want to guarantee that our estimate of the value function is within ϵ of U^* at all states, then we should choose δ to be $\epsilon(1-\gamma)/\gamma$. As γ approaches 1, the termination threshold becomes smaller, implying slower convergence. In general, the less future rewards are discounted, the more we have to iterate to look out to an acceptable horizon.

If we know that $||U_k - U^*|| < \epsilon$, then we can bound the *policy loss* of the policy extracted from U_k. If the extracted policy is π, then the policy loss is defined as $||U^\pi - U^*||$. It can be proven that $||U_k - U^*|| < \epsilon$ implies that the policy loss is less than $2\epsilon\gamma/(1-\gamma)$.

4.2.5 Grid World Example

To illustrate value iteration, we will use a 10×10 grid world problem. Each cell in the grid represents a state in an MDP. The available actions are up, down, left, and right. The effects of these actions are stochastic. We move one step in the specified direction with probability 0.7, and we move one step in one of the three other directions, each with probability 0.1. If we bump against the outer border of the grid, then we do not move at all.

We receive a cost of 1 for bumping against the outer border of the grid. There are four cells in which we receive rewards upon entering:

- $(8, 9)$ has a reward of $+10$
- $(3, 8)$ has a reward of $+3$
- $(5, 4)$ has a reward of -5
- $(8, 4)$ has a reward of -10

The coordinates are specified using the matrix convention in which the first coordinate is the row starting from the top and the second coordinate is the column starting from the left. The cells with rewards of $+10$ and $+3$ are absorbing states where no additional reward is ever received from that point onward.

Figure 4.2a shows the result of the first sweep of value iteration with a discount factor of 0.9. After this first sweep, the value function is simply the maximum expected immediate reward—i.e., $\max_a R(s, a)$. The gray pointers indicate the optimal actions from the cells as determined by Equation (4.13). As indicated in the figure, all actions are optimal for the interior cells. For the cells adjacent to the wall, the optimal actions are in the directions away from the wall.

Figure 4.2b shows the result of the second sweep. The values at the states with non-zero rewards remain the same, but the values are dispersed to adjacent cells. The value of the cells is based on expected discounted rewards after two time steps. Consequently, cells more than one step away from an absorbing cell or a cell bordering a wall have zero value. Cells within one step have had their optimal action set updated to direct us to positive rewards and away from negative rewards.

Figures 4.3a and 4.3b show the value function and policy after three and four sweeps, respectively. The value associated with the $+3$ and $+10$ cells spread outward over the grid. As the value is propagated further throughout the grid, there are fewer ties for optimal actions for the various cells.

Figures 4.4a and 4.4b show the value function and policy at convergence for $\gamma = 0.9$ and $\gamma = 0.5$, respectively. When $\gamma = 0.9$, even the cells on the left side of the grid have positive value. When the rewards are discounted more steeply with $\gamma = 0.5$, the +3 and +10 rewards do not propagate as far. The effect of steeper discounting can also be seen in the difference in policy at cell $(4, 8)$. With discounting at 0.5, the best strategy is to head straight to the +3 cell, whereas with discounting at 0.9, the best strategy is to head toward the +10 cell.

4.2.6 Asynchronous Value Iteration

The value iteration algorithm in Section 4.2.4 computes U_{k+1} based on U_k for *all* the states at each iteration. In *asynchronous value iteration*, only a subset of the states may be updated per iteration. It can be proven that, so long as the value function is updated at each state infinitely often, the value function is guaranteed to converge to the optimal value function.

Gauss-Seidel value iteration is a type of asynchronous value iteration. It sweeps through an ordering of the states and applies the following update:

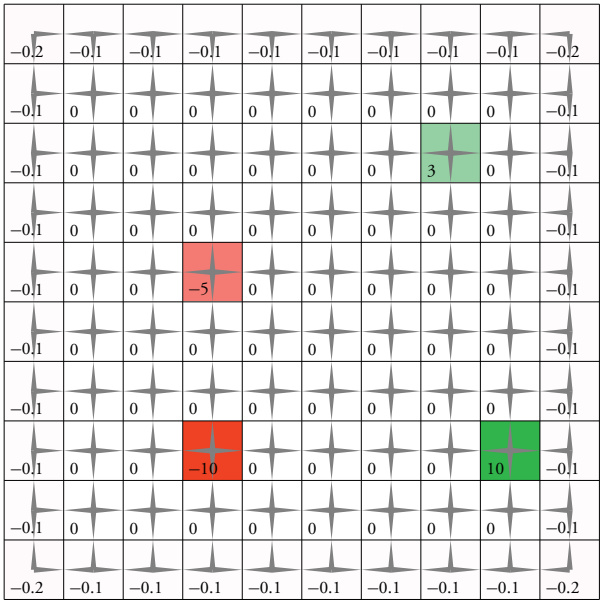
$$U(s) \leftarrow \max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U(s') \right). \tag{4.14}$$

With Gauss-Seidel value iteration, we only have to keep one copy of state values in memory instead of two because the values are updated *in place*. In addition, Gauss-Seidel can converge more quickly than standard value iteration can depending on the ordering chosen.

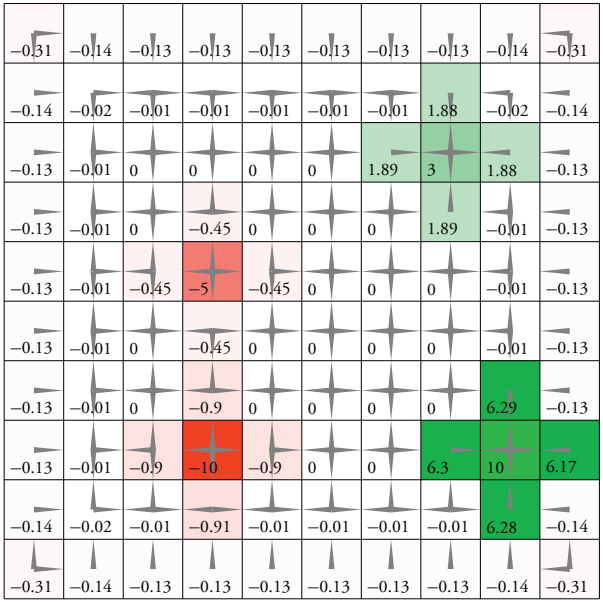
4.2.7 Closed- and Open-Loop Planning

The process of using a model to choose an action in a sequential problem is called *planning*. There are two general approaches to planning:

- *Closed-loop planning* accounts for future state information. The dynamic programming algorithms discussed in this chapter fall within this category. They involve developing a *reactive* plan (or policy) that can react to the different outcomes of the actions over time.
- *Open-loop planning* does not account for future state information. Many *path planning* algorithms fall within this category. They involve developing a static sequence of actions.

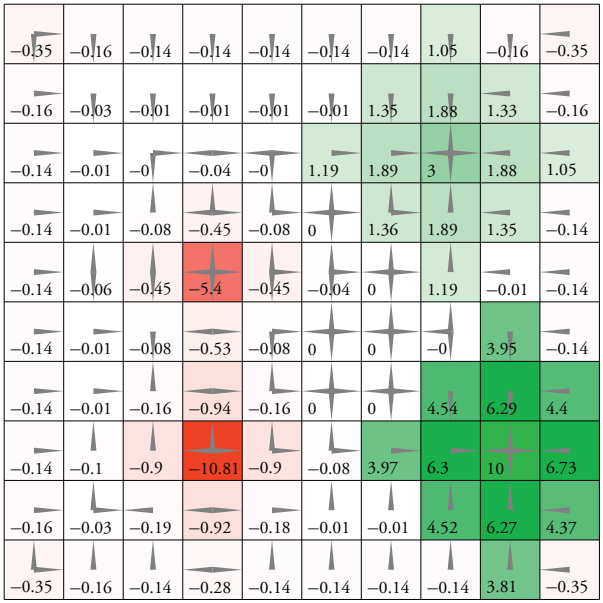


(a) Sweep 1.

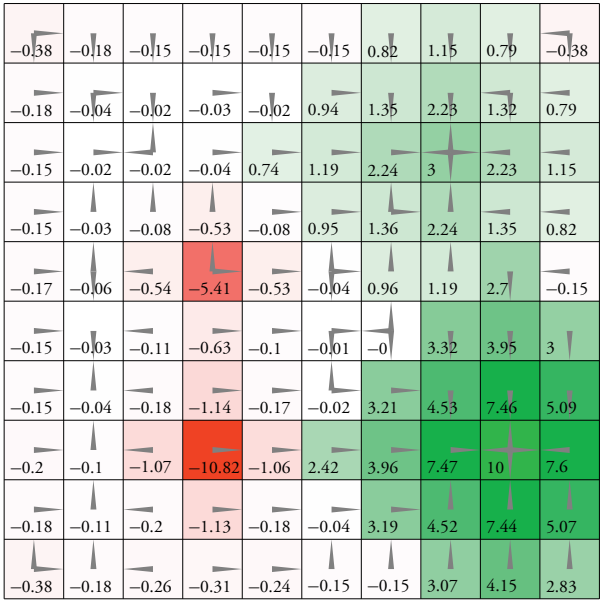


(b) Sweep 2.

Figure 4.2 Value iteration with $\gamma = 0.9$.

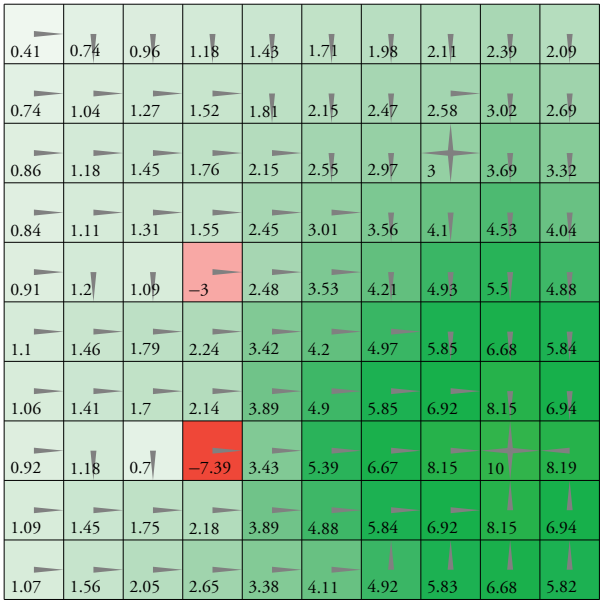


(a) Sweep 3.

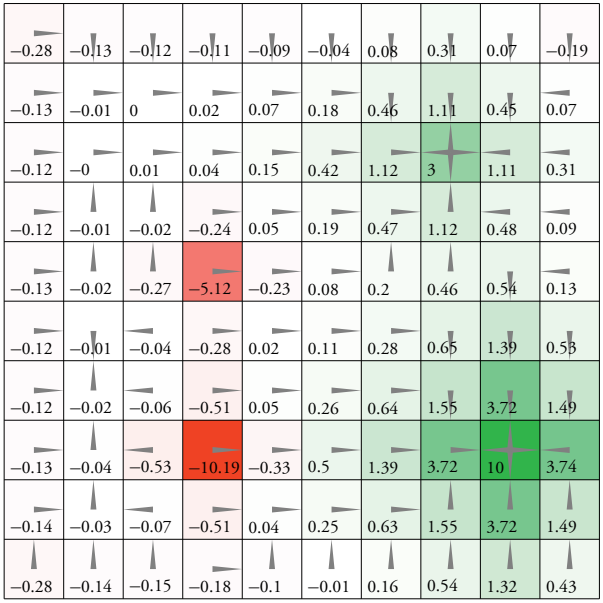


(b) Sweep 4.

Figure 4.3 Value iteration with $\gamma = 0.9$.



(a) $\gamma = 0.9$.



(b) $\gamma = 0.5$.

Figure 4.4 Value iteration at convergence.

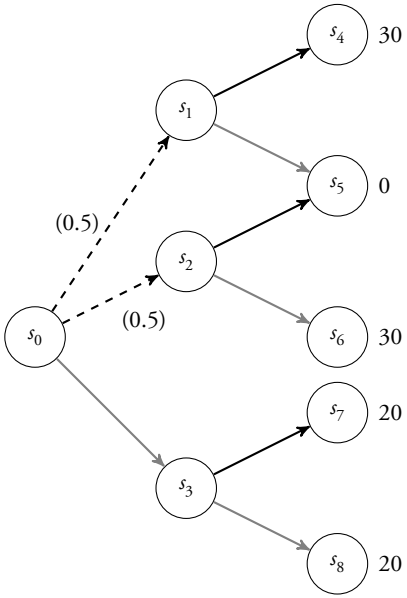


Figure 4.5 Example illustrating suboptimality of open-loop planning.

The advantage of closed-loop planning can be illustrated with the example in Figure 4.5. There are nine states, and we start in state s_0. There are two decision steps, where we must decide between going up (black arrows) or going down (gray arrows). The effects of the actions are deterministic, except that if we go up from s_0, then we end up in state s_1 half the time and in state s_2 half the time. We receive a reward of 30 in states s_4 and s_6 and a reward of 20 in states s_7 and s_8, as indicated in the figure.

There are exactly four open-loop plans: (up, up), (up, down), (down, up), and (down, down). In this simple example, it is easy to compute their expected utilities:

- $U(\text{up, up}) = 0.5 \times 30 + 0.5 \times 0 = 15$
- $U(\text{up, down}) = 0.5 \times 0 + 0.5 \times 30 = 15$
- $U(\text{down, up}) = 20$
- $U(\text{down, down}) = 20$

According to the set of open-loop plans, it is best to choose down from s_0 because our expected reward is 20 instead of 15.

Closed-loop planning, in contrast, takes into account the fact that we can base our next decision on the observed outcome of our first action. If we choose to go up from s_0, then we can choose to go down or up depending on whether we end up in s_1 or s_2, thereby guaranteeing a reward of 30.

In sequential problems where the effects of actions are uncertain, closed-loop planning can provide a significant benefit over open-loop planning. However, in some domains, the size of the state space can make the application of closed-loop planning methods, such as value iteration, infeasible. Open-loop planning algorithms, although suboptimal in principle, can provide satisfactory performance. There are many open-loop planning algorithms, but this book will focus on closed-loop methods and ways for addressing large problems without sacrificing the ability to account for the availability of future information.

4.3 Structured Representations

The dynamic programming algorithms described earlier in this chapter have assumed that the state space is discrete. If the state space is determined by n binary variables, then the number of discrete states is 2^n. This exponential growth of discrete states restricts the direct application of algorithms such as value iteration and policy iteration to problems with only a limited number of state variables. This section discusses methods that can help solve higher dimensional problems by leveraging their structure.

4.3.1 Factored Markov Decision Processes

A *factored Markov decision process* compactly represents the transition and reward functions using a dynamic decision network. The actions, rewards, and states may be factored into multiple nodes. Figure 4.6 shows an example of a factored MDP with two decision variables (A and F), three state variables (B, D, and G), and two reward variables (C and E).

We can use decision trees to compactly represent the conditional probability distributions and reward functions. For example, the conditional probability distribution $P(G_{t+1} \mid D_t, F_t, G_t)$ shown in tabular form in Figure 4.7a can be represented using the decision tree in Figure 4.7b.

Additional efficiency can be gained using *decision diagrams* instead of decision trees. In trees, all nodes (except for the root) have exactly one parent, but nodes in decision diagrams can have multiple parents. Figure 4.8 shows an example of a decision tree and an equivalent decision diagram. Instead of requiring four leaf nodes as in the decision tree, the decision diagram only requires two leaf nodes.

4.3.2 Structured Dynamic Programming

Several dynamic programming algorithms exist for finding policies for factored MDPs. Algorithms such as *structured value iteration* and *structured policy iteration* perform updates on the leaves of the decision trees instead of all the states. These algorithms improve efficiency by *aggregating states* and leveraging the additive decomposition of

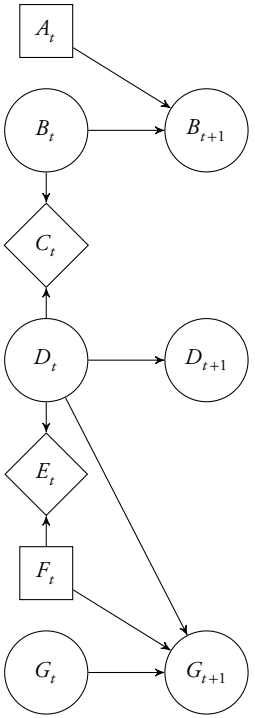
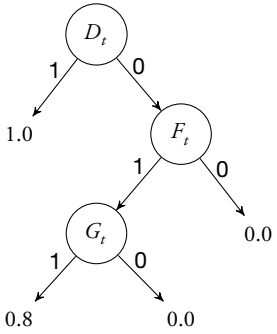


Figure 4.6 Factored Markov decision process.

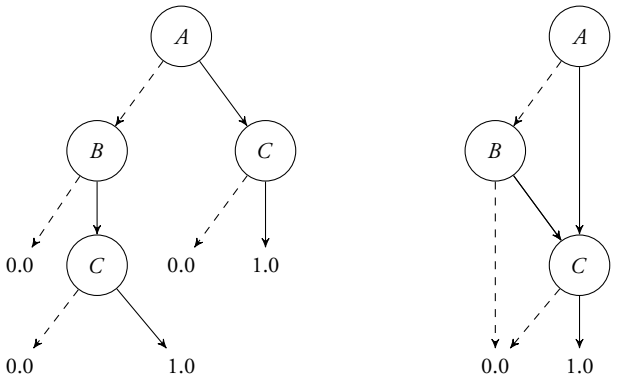
D_t	F_t	G_t	$P(g^1_{t+1} \mid D_t, F_t, G_t)$
1	1	1	1.0
1	1	0	1.0
1	0	1	1.0
1	0	0	1.0
0	1	1	0.8
0	1	0	0.0
0	0	1	0.0
0	0	0	0.0

(a) Tabular form.



(b) Decision tree form.

Figure 4.7 Conditional distribution as a decision tree.



(a) Decision tree. (b) Decision diagram.

Figure 4.8 Tree- and graph-based representations of a conditional probability table. Dashed lines are followed from nodes when the outcome of the variable test is false.

the reward and value functions. The resulting policy is represented as a decision tree in which the interior nodes correspond to tests of the state variables and the leaf nodes correspond to actions.

4.4 Linear Representations

The methods presented in this chapter so far have required that the problems be discrete. Of course, we may discretize a problem that is naturally continuous, but doing so may not be feasible if the state or action spaces are large. This section presents a method for finding exact optimal policies for problems with continuous state and action spaces that meet certain criteria:

- *Dynamics are linear Gaussian.* The state transition function has the form

$$T(\mathbf{z} \mid \mathbf{s}, \mathbf{a}) = \mathcal{N}(\mathbf{z} \mid \mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a}, \boldsymbol{\Sigma}), \tag{4.15}$$

  where $\mathbf{T}_s$ and $\mathbf{T}_a$ are matrices that determine the mean of the next state $\mathbf{z}$ based on $\mathbf{s}$ and $\mathbf{a}$, and $\boldsymbol{\Sigma}$ is a covariance matrix that controls the amount of noise in the dynamics.
- *Reward is quadratic.* The reward function has the following form

$$R(\mathbf{s}, \mathbf{a}) = \mathbf{s}^\top \mathbf{R}_s \mathbf{s} + \mathbf{a}^\top \mathbf{R}_a \mathbf{a}, \tag{4.16}$$

  where $\mathbf{R}_s = \mathbf{R}_s^\top \leq 0$ and $\mathbf{R}_a = \mathbf{R}_a^\top < 0$.

For simplicity, we will assume a finite horizon undiscounted reward problem, but the approach can be generalized to average reward and discounted infinite horizon problems as well. We may generalize Equation (4.11) for continuous state spaces by replacing the summation with an integral and making $T(s' \mid s, a)$ represent a probability density rather than a probability mass:

$$U_n(\mathbf{s}) = \max_{\mathbf{a}} \left(R(\mathbf{s}, \mathbf{a}) + \int T(\mathbf{z} \mid \mathbf{s}, \mathbf{a}) U_{n-1}(\mathbf{z}) d\mathbf{z} \right). \tag{4.17}$$

From our assumptions about T and R, we can rewrite the equation above:

$$U_n(\mathbf{s}) = \max_{\mathbf{a}} \left(\mathbf{s}^\top \mathbf{R}_s \mathbf{s} + \mathbf{a}^\top \mathbf{R}_a \mathbf{a} + \int \mathcal{N}(\mathbf{z} \mid \mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a}, \Sigma) U_{n-1}(\mathbf{z}) d\mathbf{z} \right). \tag{4.18}$$

It can be proven inductively that $U_n(\mathbf{s})$ can be written in the form $\mathbf{s}^\top \mathbf{V}_n \mathbf{s} + q_n$. We can rewrite the equation above as

$$U_n(\mathbf{s}) = \max_{\mathbf{a}} \left(\mathbf{s}^\top \mathbf{R}_s \mathbf{s} + \mathbf{a}^\top \mathbf{R}_a \mathbf{a} + \int \mathcal{N}(\mathbf{z} \mid \mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a}, \Sigma)(\mathbf{z}^\top V_{n-1} \mathbf{z} + q_{n-1}) d\mathbf{z} \right). \tag{4.19}$$

Simplifying, we get

$$U_n(\mathbf{s}) = q_{n-1} + \mathbf{s}^\top \mathbf{R}_s \mathbf{s} + \max_{\mathbf{a}} \left(\mathbf{a}^\top \mathbf{R}_a \mathbf{a} + \int \mathcal{N}(\mathbf{z} \mid \mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a}, \Sigma) \mathbf{z}^\top \mathbf{V}_{n-1} \mathbf{z} d\mathbf{z} \right). \tag{4.20}$$

The integral in the equation above evaluates to

$$\text{Tr}(\Sigma \mathbf{V}_{n-1}) + (\mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a})^\top \mathbf{V}_{n-1} (\mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a}), \tag{4.21}$$

where Tr represents the *trace* of a matrix, which is simply the sum of the main diagonal elements. We now have

$$U_n(\mathbf{s}) = q_{n-1} + \mathbf{s}^\top \mathbf{R}_s \mathbf{s} + \text{Tr}(\Sigma \mathbf{V}_{n-1}) + \max_{\mathbf{a}} \left(\mathbf{a}^\top \mathbf{R}_a \mathbf{a} + (\mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a})^\top \mathbf{V}_{n-1} (\mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a}) \right). \tag{4.22}$$

We can determine the $\mathbf{a}$ that maximizes the last term in the equation above by computing the derivative with respect to $\mathbf{a}$, setting it to 0, and solving for $\mathbf{a}$:

$$2\mathbf{a}^\top \mathbf{R}_a + 2(\mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a})^\top \mathbf{V}_{n-1} \mathbf{T}_a = 0 \tag{4.23}$$

$$\mathbf{a} = -(\mathbf{T}_a^\top \mathbf{V}_{n-1} \mathbf{T}_a + \mathbf{R}_a)^{-1} \mathbf{T}_a \mathbf{V}_{n-1} \mathbf{T}_s \mathbf{s}. \tag{4.24}$$

Substituting Equation (4.24) into Equation (4.22) and simplifying, we get $U_n(\mathbf{s}) = \mathbf{s}^\top \mathbf{V}_n \mathbf{s} + q_n$ with

$$\mathbf{V}_n = \mathbf{T}_s^\top \mathbf{V}_{n-1} \mathbf{T}_s - \mathbf{T}_s^\top \mathbf{V}_{n-1} \mathbf{T}_a (\mathbf{T}_a^\top \mathbf{T}_a + \mathbf{R}_a)^{-1} \mathbf{T}_s^\top \mathbf{V}_{n-1} \mathbf{T}_s + \mathbf{R}_s \tag{4.25}$$

$$q_n = q_{n-1} + \text{Tr}(\Sigma \mathbf{V}_{n-1}). \tag{4.26}$$

To compute $\mathbf{V}_n$ and q_n for arbitrary n, we first set $\mathbf{V}_0 = 0$ and $q_0 = 0$ and iterate using the equations above. Once we know $\mathbf{V}_{n-1}$ and q_{n-1}, we can extract the optimal n-step policy

$$\pi_n(\mathbf{s}) = -(\mathbf{T}_a^\top \mathbf{V}_{n-1} \mathbf{T}_a + \mathbf{R}_a)^{-1} \mathbf{T}_a \mathbf{V}_{n-1} \mathbf{T}_s \mathbf{s}. \tag{4.27}$$

Interestingly, $\pi_n(\mathbf{s})$ does not depend on the covariance of the noise Σ, although the optimal cost does depend on the noise. A linear system with quadratic cost that has no noise in the dynamics is known in control theory as a *linear quadratic regulator* and has been well studied.

4.5 Approximate Dynamic Programming

The field of *approximate dynamic programming* is concerned with finding approximately optimal policies for problems with large or continuous spaces. Approximate dynamic programming is an active area of research that shares ideas with reinforcement learning. In reinforcement learning, we try to quickly accrue as much reward as possible without a known model. Many of the algorithms for reinforcement learning (discussed in the next chapter) can be applied directly to approximate dynamic programming. This section focuses on several local and global approximation strategies for efficiently finding value functions and policies for known models.

4.5.1 Local Approximation

Local approximation relies on the assumption that states close to each other have similar values. If we know the value associated with a finite set of states $s_{1:n}$, then we can approximate the value of arbitrary states by using the equation

$$U(s) = \sum_{i=1}^{n} \lambda_i \beta_i(s) = \boldsymbol{\lambda}^\top \boldsymbol{\beta}(s), \tag{4.28}$$

where $\beta_{1:n}$ are weighting functions, such that $\sum_{i=1}^{n} \beta_i(s) = 1$. The value λ_i is the value of state s_i. In general, $\beta_i(s)$ should assign greater weight to states that are closer (in some sense) to s_i. A weighting function is often referred to as a *kernel*.

Algorithm 4.4 shows how to compute an approximation of the optimal value function by iteratively updating $\boldsymbol{\lambda}$. The loop is continued until convergence. Once an approximate value function is known, an approximately optimal policy can be extracted as follows:

$$\pi(s) \leftarrow \arg\max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) \boldsymbol{\lambda}^\top \boldsymbol{\beta}(s') \right). \tag{4.29}$$

Algorithm 4.4 Local approximation value iteration

```
function LocalApproximationValueIteration
    λ ← 0
    loop
        for i ← 1 to n
            u_i ← max_a [R(s_i, a) + γ Σ_{s'} T(s' | s_i, a) λ^⊤ β(s')]
        λ ← u
    return λ
```

A simple approach to local approximation, called *nearest neighbor*, is to assign all weight to the closest discrete state, resulting in a piecewise constant value function. A smoother approximation can be achieved using *k-nearest neighbor*, where a weight of $1/k$ is assigned to each of the k nearest discrete states of s.

If we define a neighborhood function $N(s)$ that returns a set of states from $s_{1:n}$, then we can use *linear interpolation*. If the state space is one dimensional and $N(s) = \{s_1, s_2\}$, then the interpolated value is given by

$$U(s) = \lambda_1 \underbrace{\left(1 - \frac{s - s_1}{s_2 - s_1}\right)}_{\beta_1(s)} + \lambda_2 \underbrace{\left(1 - \frac{s_2 - s}{s_2 - s_1}\right)}_{\beta_2(s)}. \tag{4.30}$$

The equation above can be generalized to d-dimensional state spaces and is often called *bilinear interpolation* in two dimensions and *multilinear interpolation* in arbitrary dimensions.

If the state space has been discretized using a multidimensional grid and the vertices of the grid correspond to the discrete states, then $N(s)$ could be defined to be the set of vertices of the rectangular cell that encloses s. In d dimensions, there may be as many as 2^d neighbors.

Figure 4.9 shows an example grid-based discretization of a two-dimensional state space. To determine the interpolated value of state s (shown as a black circle in the figure), we look at the discrete states in $N(s) = \{s_{12}, s_{13}, s_{17}, s_{18}\}$ (shown as white circles). Using the neighboring values and weights shown in the figure, we compute the value at

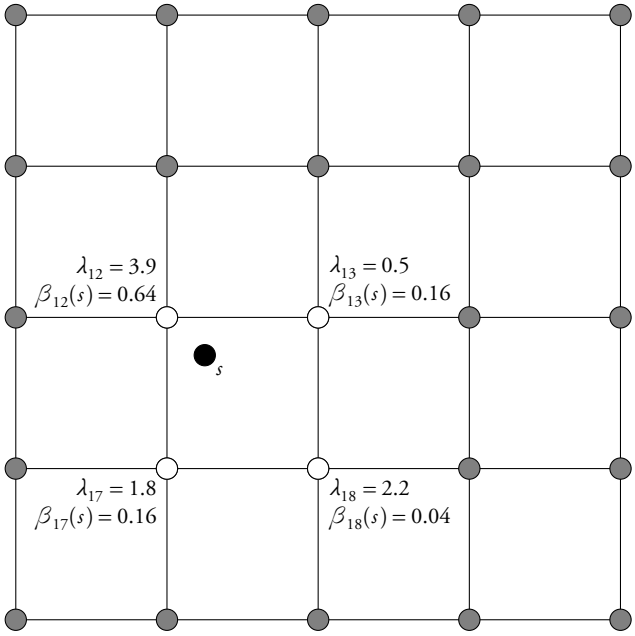


Figure 4.9 Multilinear interpolation in two dimensions.

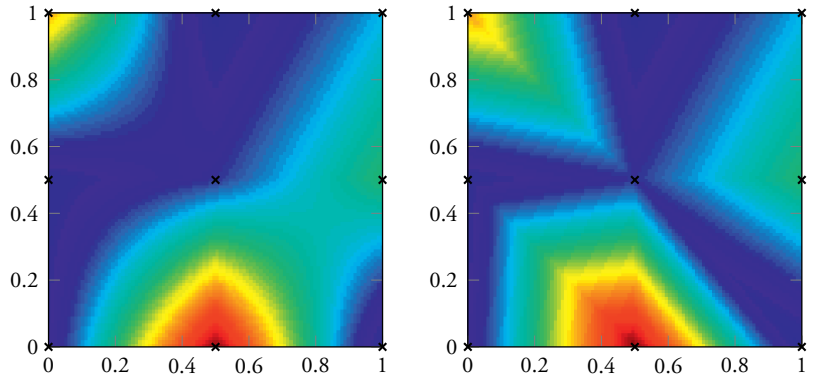
s as follows:

$$U(s) = \lambda_{12}\beta_{12}(s) + \lambda_{13}\beta_{13}(s) + \lambda_{17}\beta_{17}(s) + \lambda_{18}\beta_{18}(s) \tag{4.31}$$

$$= 3.9 \times 0.64 + 0.5 \times 0.16 + 1.8 \times 0.16 + 2.2 \times 0.04 \tag{4.32}$$

$$= 2.952. \tag{4.33}$$

When the dimensionality of the problem is high, it may be prohibitive to interpolate over the 2^d vertices in the enclosing rectangular cell. An alternative is to use *simplex-based interpolation*. In the simplex method, the rectangular cells are broken into $d!$ multidimensional triangles, called *simplexes*. Instead of interpolating over rectangular cells, we interpolate over a simplex defined by up to $d + 1$ vertices. Hence, interpolating over the simplex scales linearly instead of exponentially with the dimensionality of the state space. However, rectangular interpolation can provide higher quality estimates that can lead to better policies for the same grid resolution. Figure 4.10 shows an example of two-dimensional rectangular and simplex interpolation.



(a) Rectangular interpolation. (b) Simplex interpolation.

Figure 4.10 Interpolation methods. Data points are indicated by crosses.

4.5.2 Global Approximation

Global approximation uses a fixed set of parameters $\lambda_{1:m}$ to approximate the value function over the entire state space $\mathcal{S}$. One of the most commonly used global approximation methods is based on *linear regression*. We define a set of *basis functions* $\beta_{1:m}$, where $\beta_i : \mathcal{S} \to \mathbb{R}$. Sometimes these basis functions are called *features*. The approximation of $U(s)$ is a linear combination of the parameters and output of the basis functions:

$$U(s) = \sum_{i=1}^{m} \lambda_i \beta_i(s) = \boldsymbol{\lambda}^\top \boldsymbol{\beta}(s). \tag{4.34}$$

The approximation above has the same form as Equation (4.28), but the interpretation is different. The parameters $\lambda_{1:m}$ do not correspond to values at discrete states. The basis functions $\beta_{1:m}$ are not necessarily related to a distance measure, and they need not sum to 1.

Algorithm 4.5 shows how to incorporate linear regression into value iteration. The algorithm is nearly identical to Algorithm 4.4, except for Line 6. Instead of simply assigning $\boldsymbol{\lambda} \leftarrow \mathbf{u}$ as done in local approximation, we call $\lambda_{1:m} \leftarrow \textsc{Regress}(\beta, s_{1:n}, u_{1:n})$. The Regress function finds the $\boldsymbol{\lambda}$ that leads to the best approximation of the target values $u_{1:n}$ at points $s_{1:n}$ using the basis function β. A common regression objective is to minimize the *sum-squared error*:

$$\sum_{i=1}^{n} \left(\boldsymbol{\lambda}^\top \boldsymbol{\beta}(s_i) - u_i \right)^2. \tag{4.35}$$

Linear least-squares regression can compute the $\boldsymbol{\lambda}$ that minimizes the sum-squared error through simple matrix operations. There are a wide variety of other well-studied regression approaches, both linear and non-linear.

Algorithm 4.5 Linear regression value iteration

```
function LinearRegressionValueIteration
    λ ← 0
    loop
        for i ← 1 to n
            u_i ← max_a [R(s_i, a) + γ Σ_{s'} T(s' | s_i, a) λ^⊤ β(s')]
        λ_{1:m} ← Regress(β, s_{1:n}, u_{1:n})
    return λ
```

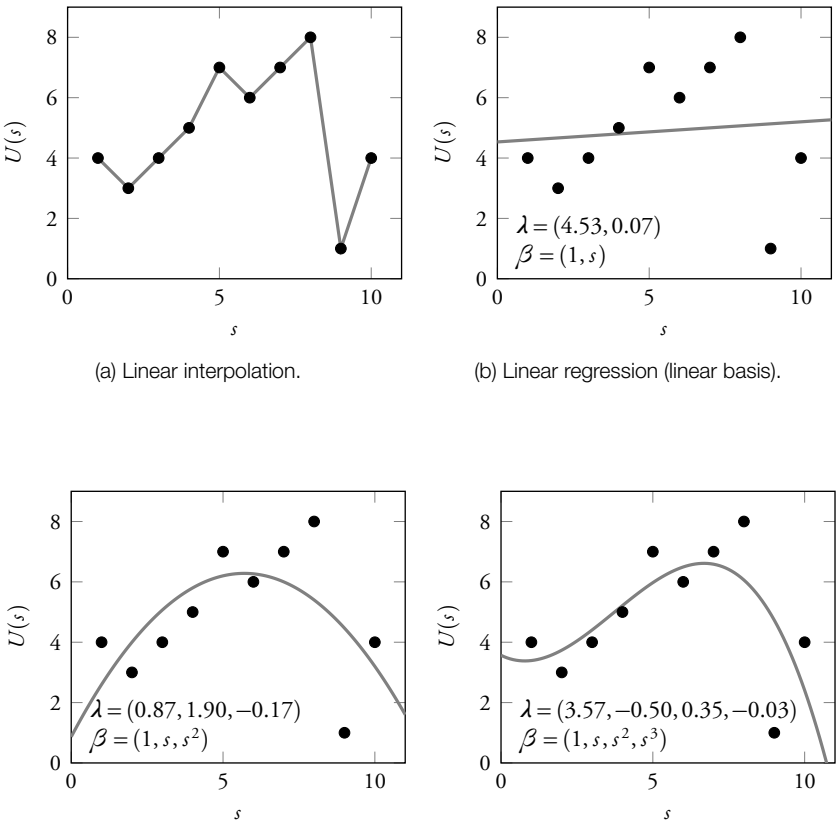
Figure 4.11 compares linear interpolation with linear regression by using different basis functions. For simplicity, the figure assumes a one-dimensional state space, and the states $s_{1:10}$ are evenly spaced. The target values $u_{1:10}$ obtained through dynamic programming are plotted as dots.

Figure 4.11a shows linear interpolation, which produces an approximate value function that matches $u_{1:10}$ exactly at the states $s_{1:10}$. Linear interpolation, of course, requires 10 parameters.

Figure 4.11b shows the result of linear least-squares regression with basis functions $\beta_1(s) = 1$ and $\beta_2(s) = s$. In this case, $\lambda_1 = 4.53$ and $\lambda_2 = 0.07$, meaning that $U(s)$ is approximated as $4.53 + 0.07s$. Although this $\boldsymbol{\lambda}$ minimizes the sum-squared error given those two basis functions, the plot shows that the resulting approximate value function is not especially accurate.

Figure 4.11c shows the result of adding an additional basis function $\beta_3(s) = s^2$. The approximate value function is now quadratic over the state space. Adding this additional basis function results in new values for λ_1 and λ_2. The sum-squared error of the quadratic value function at the states $s_{1:n}$ is much smaller than with the linear function.

Figure 4.11d adds an additional cubic basis function $\beta_4(s) = s^3$, further improving the approximation. All of the basis functions in this example are polynomials, but we could have easily added other basis functions such as $\sin(s)$ and e^s. Adding additional basis functions can generally improve the ability to match the target values at the known states, but too many basis functions can lead to poor approximations at other states. Principled methods exist for choosing an appropriate set of basis functions for regression.



(a) Linear interpolation.

(b) Linear regression (linear basis).

(c) Linear regression (quadratic basis).

(d) Linear regression (cubic basis).

Figure 4.11 Approximations of the value function.

4.6 Online Methods

All the methods presented in this chapter so far involve computing the policy for the entire state space *offline*—that is, prior to execution in the environment. Although factored representations and value function approximation can help scale dynamic programming to higher dimensional state spaces, computing and representing a policy over the full state space can still be intractable. This section discusses *online* methods that restrict computation to states that are reachable from the current state. Because the reachable state space can be orders of magnitude smaller than the full state space, online methods can significantly reduce the amount of storage and computation required to choose optimal (or approximately optimal) actions.

4.6.1 Forward Search

Forward search (Algorithm 4.6) is a simple online action-selection method that looks ahead from some initial state s_0 to some horizon (or depth) d. The forward search function SelectAction(s, d) returns the optimal action a^* and its value v^*. The pseudocode uses $A(s)$ to represent the set of actions available from state s, which may be a subset of the full action space A. The set of possible states that can follow immediately from s after executing action a is denoted $S(s, a)$, which may be a small subset of the full state spaces S.

Algorithm 4.6 Forward search

```
function SelectAction(s, d)
    if d = 0
        return (nil, 0)
    (a*, v*) ← (nil, −∞)
    for a ∈ A(s)
        v ← R(s, a)
        for s′ ∈ S(s, a)
            (a′, v′) ← SelectAction(s′, d − 1)
            v ← v + γ T(s′ | s, a) v′
        if v > v*
            (a*, v*) ← (a, v)
    return (a*, v*)
```

Algorithm 4.6 iterates over all possible action and next state pairings and calls itself recursively until the desired depth is reached. The call tree has depth d with a worst-case

branching factor of $|S| \times |A|$ and proceeds depth-first. The computational complexity is $O((|S| \times |A|)^d)$.

4.6.2 Branch and Bound Search

Branch and bound search (Algorithm 4.7) is an extension to forward search that uses knowledge of the upper and lower bounds of the value function to prune portions of the search tree. This algorithm assumes that prior knowledge is available that allows us to easily compute a lower bound on the value function $\underline{U}(s)$ and an upper bound on the state-action value function $\overline{U}(s,a)$. The pseudocode is identical to Algorithm 4.6, except for the use of the lower bound in Line 3 and the pruning check in Line 6. The call to SelectAction(s,d) returns the action to execute and a lower bound on the value function.

The order in which we iterate over the actions in Line 5 is important. In order to prune, the actions must be in descending order of upper bound. In other words, if action a_i is evaluated before a_j, then $\overline{U}(s,a_i) \geq \overline{U}(s,a_j)$. The tighter we are able to make the upper and lower bounds, the more we can prune the search space and decrease computation time. The worst-case computational complexity, however, remains the same as for forward search.

Algorithm 4.7 Branch-and-bound search

```
1:  function SelectAction(s, d)
2:      if d = 0
3:          return (nil, U̲(s))
4:      (a*, v*) ← (nil, −∞)
5:      for a ∈ A(s)
6:          if Ū(s, a) < v*
7:              return (a*, v*)
8:          v ← R(s, a)
9:          for s′ ∈ S(s, a)
10:             (a′, v′) ← SelectAction(s′, d − 1)
11:             v ← v + γ T(s′ | s, a) v′
12:         if v > v*
13:             (a*, v*) ← (a, v)
14:     return (a*, v*)
```

4.6.3 Sparse Sampling

Sampling methods can be used to avoid the worst-case exponential complexity of forward and branch-and-bound search. Although these methods are not guaranteed to produce the optimal action, they can be shown to produce approximately optimal actions most of the time and can work well in practice. One of the simplest approaches is referred to as *sparse sampling* (Algorithm 4.8).

Sparse sampling uses a generative model G to produce samples of the next state s' and reward r. An advantage of using a generative model is that it is often easier to implement code for drawing random samples from a complex, multidimensional distribution rather than explicitly representing probabilities. Line 8 of the algorithm draws $(s', r) \sim G(s, a)$. All of the information about the state transitions and rewards is represented by G; the state transition probabilities $T(s' \mid s, a)$ and expected reward function $R(s, a)$ are not used directly.

Algorithm 4.8 Sparse sampling

```
1:  function SelectAction(s, d)
2:      if d = 0
3:          return (nil, 0)
4:      (a*, v*) ← (nil, −∞)
5:      for a ∈ A(s)
6:          v ← 0
7:          for i ← 1 to n
8:              (s', r) ~ G(s, a)
9:              (a', v') ← SelectAction(s', d − 1)
10:             v ← v + (r + γv')/n
11:         if v > v*
12:             (a*, v*) ← (a, v)
13:     return (a*, v*)
```

Sparse sampling is similar to forward search, except that it iterates over n samples instead of all the states in $S(s, a)$. Each iteration results in a sample of $r + \gamma v'$, where r comes from the generative model and v' comes from a recursive call to SelectAction$(s', d - 1)$. These samples of $r + \gamma v'$ are averaged together to estimate $Q(s, a)$. The run time complexity $O((n \times |A|)^d)$ is still exponential in the horizon but does not depend on the size of the state space.

4.6.4 Monte Carlo Tree Search

One of the most successful sampling-based online approaches in recent years is *Monte Carlo tree search*. Algorithm 4.9 is the Upper Confidence Bound for Trees (UCT) implementation of Monte Carlo tree search. In contrast with sparse sampling, the complexity of Monte Carlo tree search does not grow exponentially with the horizon. As in sparse sampling, we use a generative model.

Algorithm 4.9 Monte Carlo tree search

```
function SelectAction(s, d)
    loop
        Simulate(s, d, π_0)
    return arg max_a Q(s, a)
function Simulate(s, d, π_0)
    if d = 0
        return 0
    if s ∉ T
        for a ∈ A(s)
            (N(s, a), Q(s, a)) ← (N_0(s, a), Q_0(s, a))
        T = T ∪ {s}
        return Rollout(s, d, π_0)
    a ← arg max_a Q(s, a) + c sqrt(log N(s) / N(s, a))
    (s', r) ~ G(s, a)
    q ← r + γ Simulate(s', d − 1, π_0)
    N(s, a) ← N(s, a) + 1
    Q(s, a) ← Q(s, a) + (q − Q(s, a)) / N(s, a)
    return q
```

Algorithm 4.10 Rollout evaluation

```
function Rollout(s, d, π_0)
    if d = 0
        return 0
    a ~ π_0(s)
    (s', r) ~ G(s, a)
    return r + γ Rollout(s', d − 1, π_0)
```

The algorithm involves running many simulations from the current state while updating an estimate of the state-action value function $Q(s,a)$. There are three stages in each simulation:

- *Search*. If the current state in the simulation is in the set T (initially empty), then we enter the search stage. Otherwise we proceed to the expansion stage. During the search stage, we update $Q(s,a)$ for the states and actions visited and tried in our search. We also keep track of the number of times we have taken an action from a state $N(s,a)$. During the search, we execute the action that maximizes

$$Q(s,a) + c\sqrt{\frac{\log N(s)}{N(s,a)}}, \tag{4.36}$$

  where $N(s) = \sum_a N(s,a)$ and c is a parameter that controls the amount of exploration in the search (exploration will be covered in depth in the next chapter). The second term is an *exploration bonus* that encourages selecting actions that have not been tried as frequently.
- *Expansion*. Once we have reached a state that is not in the set T, we iterate over all of the actions available from that state and initialize $N(s,a)$ and $Q(s,a)$ with $N_0(s,a)$ and $Q_0(s,a)$, respectively. The functions N_0 and Q_0 can be based on prior expert knowledge of the problem; if none is available, then they can both be initialized to 0. We then add the current state to the set T.
- *Rollout*. After the expansion stage, we simply select actions according to some *rollout* (or default) policy π_0 until the desired depth is reached (Algorithm 4.10). Typically, rollout policies are stochastic, and so the action to execute is sampled $a \sim \pi_0(s)$. The rollout policy does not have to be close to optimal, but it is a way for an expert to bias the search into areas that are promising. The expected value is returned and used in the search to update the value for $Q(s,a)$.

Simulations are run until some stopping criterion is met, often simply a fixed number of iterations. We then execute the action that maximizes $Q(s,a)$. Once that action has been executed, we can rerun the Monte Carlo tree search to select the next action. It is common to carry over the values of $N(s,a)$ and $Q(s,a)$ computed in the previous step.

4.7 Direct Policy Search

The previous sections have presented methods that involve computing or approximating the value function. An alternative is to search the space of policies directly. Although the state space may be high dimensional, making approximation of the value function difficult, the space of possible policies may be relatively low dimensional and can be easier to search directly.

4.7.1 Objective Function

Suppose we have a policy that is parametrized by λ. The probability that the policy selects action a given state s is written $\pi_\lambda(a \mid s)$. Given an initial state s, we can estimate

$$U^{\pi_\lambda}(s) \approx \frac{1}{n} \sum_{i=1}^{n} u_i, \tag{4.37}$$

where u_i is the ith rollout of the policy π_λ to some depth.

The objective in direct policy search is to find the parameter λ that maximizes the function

$$V(\lambda) = \sum_s b(s) U^{\pi_\lambda}(s), \tag{4.38}$$

where $b(s)$ is a distribution over the initial state. We can estimate $V(\lambda)$ using Monte Carlo simulation and a generative model G up to depth d as outlined in Algorithm 4.11.

Algorithm 4.11 Monte Carlo policy evaluation

```
function MonteCarloPolicyEvaluation(λ, d)
    for i ← 1 to n
        s ~ b
        u_i ← Rollout(s, d, π_λ)
    return (1/n) Σ_{i=1}^{n} u_i
```

The function $V(\lambda)$, as estimated by Algorithm 4.11, is a *stochastic function*; given the same input λ, it may give different outputs. As the number of samples (determined by n and m) increases, the variability of the outputs of the function decreases. Many different methods exist for searching the space of policy parameters that maximizes $V(\lambda)$, and we will discuss a few of them.

4.7.2 Local Search Methods

A common stochastic optimization approach is *local search*, also known as *hill climbing* or *gradient ascent*. Local search begins at a single point in the search space and then incrementally moves from neighbor to neighbor in the search space until convergence. The search operates with the assumption that the value of the stochastic function at a point in the search space is an indication of how close that point is to the global optimum. Therefore, local search generally selects the neighbor with the largest value.

Some local search techniques directly estimate the gradient $\nabla_\lambda V$ for a particular policy and then step some amount in the direction of steepest ascent. For some policy representations, it is possible to analytically derive the gradient. Other local search techniques evaluate a finite sampling of the neighborhood of the current search point and then move to the neighbor with the greatest value. Local search is susceptible to local optima and plateaus in $V(\lambda)$. Simulated annealing or some of the other methods suggested at the end of Section 2.4.2 can be applied to help find a global optimum.

4.7.3 Cross Entropy Methods

There is another class of policy search methods that maintain a distribution over policies and updates the distribution based on policies that perform well. One approach to updating this distribution is to use the *cross entropy* method. Cross entropy is a concept from information theory and is a measure of the difference between two distributions. If distributions p and q are discrete, then the cross entropy is given by

$$H(p, q) = -\sum_x p(x) \log q(x). \tag{4.39}$$

For continuous distributions, the summation is replaced with an integral. In the context of direct policy search, we are interested in distributions over λ. These distributions are parameterized by θ, which may be multivariate.

The cross entropy method takes as input an initial θ and parameters n and m that determine the number of samples to use. The process consists of two stages that are repeated until convergence or some other stopping criterion is met:

- *Sample.* We draw n samples from $P(\lambda; \theta)$ and evaluate their performance using Algorithm 4.11. Sort the samples in decreasing order of performance so that $i < j$ implies that $V(\lambda_i) \geq V(\lambda_j)$.
- *Update.* Use the top m performing samples (often called the *elite* samples) to update θ using cross entropy minimization, which boils down to:

$$\theta \leftarrow \arg\max_\theta \sum_{j=1}^{m} \log P(\lambda_j \mid \theta). \tag{4.40}$$

The new θ happens to correspond to the maximum likelihood estimate based on the m top-performing samples. Further explanation can be found in the references at the end of the chapter.

Algorithm 4.12 Cross entropy policy search

```
function CrossEntropyPolicySearch(θ)
    repeat
        for i ← 1 to n
            λ_i ~ P(·; θ)
            v_i ← MonteCarloPolicyEvaluation(λ_i)
        Sort (λ_1, ... λ_n) in decreasing order of v_i
        θ ← arg max_θ Σ_{j=1}^{m} log P(λ_j | θ)
    until convergence
    return λ ← arg max P(λ | θ)
```

The full process is outlined in Algorithm 4.12. The initial distribution parameter θ, the number of samples n, and the number of elite samples m are the input parameters to this process. To prevent the search from focusing too much on local maxima, the initial θ should provide a diffuse distribution over λ. The choice of the number of samples and elite samples depends on the problem.

To illustrate the cross entropy method, we will assume that the space of policies is one dimensional and that $V(\lambda)$ is as shown in Figure 4.12. We will assume for this example that the parameter $\theta = (\mu, \sigma)$ and $P(\lambda \mid \theta) = \mathcal{N}(\lambda \mid \mu, \sigma^2)$. Initially, we set $\theta = (0, 10)$. To not overwhelm the plots, we use only $n = 20$ samples and $m = 5$ elite samples. Typically, especially for higher dimensional problems, we use one to two orders of magnitude more samples.

Figure 4.12a shows the initial distribution over λ. We draw 20 samples from this distribution. The 5 elite samples are shown with circles, and the 15 other samples are shown with crosses. Those 5 elite samples are used to update θ. Because we are using a Gaussian distribution, the update simply sets the mean to the mean of the elite samples and the standard deviation to the sample standard deviation of the elite samples. The updated distribution is shown in Figure 4.12b. The process is repeated. In the third iteration (Figure 4.12c), the distribution is moved toward the more promising area of the search space. By the fourth iteration (Figure 4.12d), we have found the global optimum.

4.7.4 Evolutionary Methods

Evolutionary search methods derive inspiration from biological evolution. A common approach is to use a *genetic algorithm* that evolves populations of (typically binary) strings representing policies, starting with an initial random population. The strings recombine through genetic crossover and mutation at a rate proportional to their measured fitness

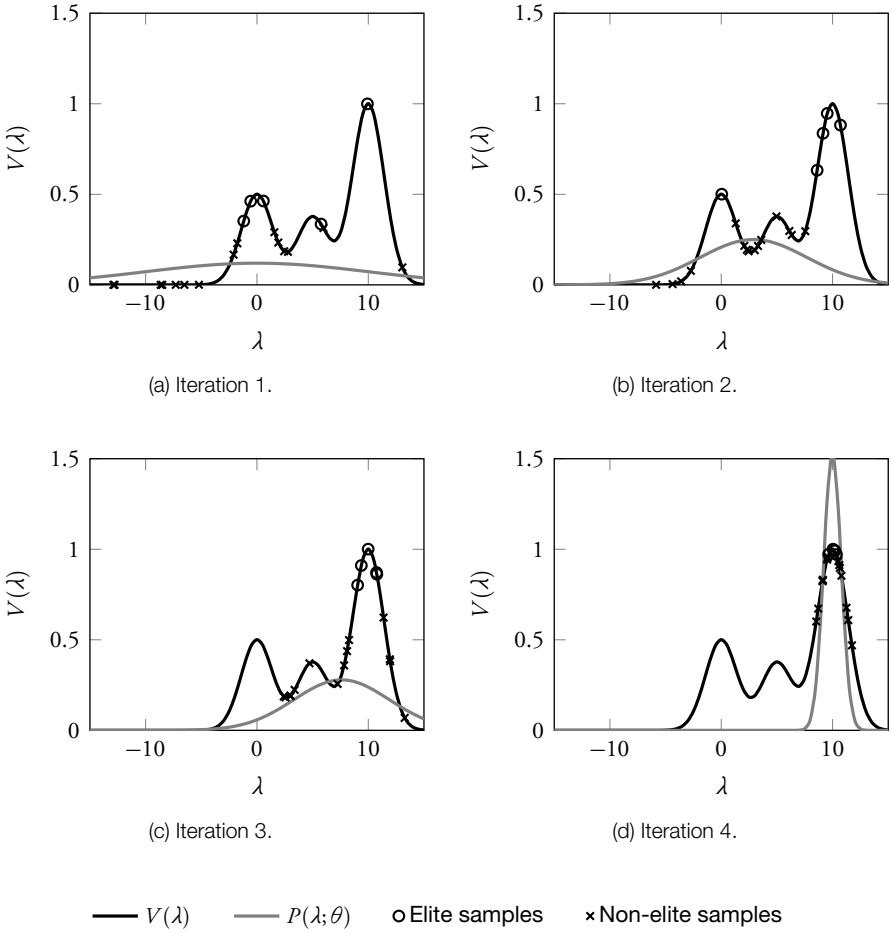


Figure 4.12 Cross entropy method iterations.

to produce a new generation. The process continues until arriving at a satisfactory solution.

A related approach is *genetic programming*, which involves evolving tree structures representing policies. Trees consist of symbols selected from predefined sets of terminals and non-terminals, allowing more flexible policy representations than fixed-length bit strings. Crossover works by swapping subtrees, and mutation works by randomly modifying subtrees.

Genetic algorithms and genetic programming may be combined with other methods, including local search. For example, a genetic algorithm might evolve a satisfactory policy and then use local search to further improve the policy. Such an approach is called *genetic local search* or *memetic algorithms*.

4.8 Summary

- Markov decision processes represent sequential decision-making problems using a transition and reward function.
- Optimal policies can be found using dynamic programming algorithms.
- Continuous problems with linear Gaussian dynamics and quadratic costs can be solved analytically.
- Structured dynamic programming can efficiently solve factored Markov decision processes.
- Problems with large or continuous state spaces can be solved approximately using function approximation.
- Instead of solving for the optimal strategy for the full state space offline, online methods search for the optimal action from the current state.
- In some problems, it can be easier to search the space of policies directly using stochastic optimization methods.

4.9 Further Reading

Much of the pioneering work on sequential decision problems was begun in 1949 by Richard Bellman [1]. Markov decision processes have since become the standard framework for modeling such problems, and there are several books on the subject [2]–[5]. The example grid world problem in Section 4.2.5 comes from *Artificial Intelligence: Foundations of Computational Agents* by Poole and Mackworth [6]. The website associated with the book contains an open-source software demonstration of the grid world example.

Boutilier, Dearden, and Goldszmidt present structured value iteration and policy iteration algorithms for factored MDPs using decision trees [7]. As mentioned in Section 4.3.2, it can be more efficient to use decision diagrams instead of trees [7]–[9].

Approximate linear programming approaches to factored MDPs have been explored by Guestrin et al. [10].

Optimal control in linear systems with quadratic costs has been well studied in the control theory community, and there are many books on the subject [11]–[13]. Section 4.4 presented a special case of the linear-quadratic-Gaussian (LQG) control problem in which the state of the system is known perfectly. Chapter 6 will present the more traditional version of LQG with imperfect state information.

An overview of the field of approximate dynamic programming is provided in *Approximate Dynamic Programming: Solving the Curses of Dimensionality* by Powell [14]. The book *Reinforcement Learning and Dynamic Programming Using Function Approximators* by Busoniu et al. outlines approximation methods and provides source code for cases where the model is known and unknown [15]. Solving problems in which the model is unknown is called reinforcement learning and is discussed in the next chapter. Reinforcement learning is often used in problems with a known model that is too complex or high dimensional to apply exact dynamic programming.

As discussed in Section 4.6, online methods are often appropriate when the state space is high dimensional and adequate computational resources are available to perform planning during execution. Land and Doig originally proposed the branch and bound method for discrete programming problems [16]. This method has been applied to a wide variety of optimization problems. Sparse sampling was developed by Kearns, Mansour, and Ng [17]. Other methods that can be used online include real-time dynamic programming [18] and LAO* [19].

Kocsis and Szepesvári originally introduced the idea of Monte Carlo tree search with the use of the exploration bonus in Algorithm 4.9 [20]. Since that paper's publication and with the successful application to the game Go, a tremendous amount of work has focused on Monte Carlo tree search methods [21]. One important extension to the algorithm presented in this chapter is the idea of progressive widening of the actions and states considered at each step in the search [22]. Progressive widening allows the algorithm to better handle large or continuous state or action spaces.

Many methods have been proposed for searching the space of policies directly. Examples of local search algorithms using gradient methods include those of Williams [23] and Baxter and Bartlett [24]. The cross entropy method [25], [26] has been applied to a variety of formulations of MDP policy search [27]–[29]. Any stochastic optimization technique can be applied to policy search. The evolutionary methods discussed in Section 4.7.4 date back to the 1950s [30]. Genetic algorithms, in particular, were made popular by the work of Holland [31], with more recent theoretical work done by Schmitt [32], [33]. Genetic programming was introduced by Koza [34].

References

1. S. Dreyfus, "Richard Bellman on the Birth of Dynamic Programming," *Operations Research*, vol. 50, no. 1, pp. 48–51, 2002.
2. R.E. Bellman, *Dynamic Programming*. Princeton, NJ: Princeton University Press, 1957.
3. D. Bertsekas, *Dynamic Programming and Optimal Control*. Belmont, MA: Athena Scientific, 2007.
4. M.L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Hoboken, NJ: Wiley, 2005.
5. O. Sigaud and O. Buffet, eds., *Markov Decision Processes in Artificial Intelligence*. New York: Wiley, 2010.
6. D.L. Poole and A.K. Mackworth, *Artificial Intelligence: Foundations of Computational Agents*. New York: Cambridge University Press, 2010.
7. C. Boutilier, R. Dearden, and M. Goldszmidt, "Stochastic Dynamic Programming with Factored Representations," *Artificial Intelligence*, vol. 121, no. 1-2, pp. 49–107, 2000. DOI: 10.1016/S0004-3702(00)00033-3.
8. J. Hoey, R. St-Aubin, A.J. Hu, and C. Boutilier, "SPUDD: Stochastic Planning Using Decision Diagrams," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 1999.
9. R. St-Aubin, J. Hoey, and C. Boutilier, "APRICODD: Approximate Policy Construction Using Decision Diagrams," in *Advances in Neural Information Processing Systems (NIPS)*, 2000.
10. C. Guestrin, D. Koller, R. Parr, and S. Venkataraman, "Efficient Solution Algorithms for Factored MDPs," *Journal of Artificial Intelligence Research*, vol. 19, pp. 399–468, 2003. DOI: 10.1613/jair.1000.
11. F.L. Lewis, D.L. Vrabie, and V.L. Syrmos, *Optimal Control*, 3rd ed. Hoboken, NJ: Wiley, 2012.
12. R.F. Stengel, *Optimal Control and Estimation*. New York: Dover Publications, 1994.
13. D.E. Kirk, *Optimal Control Theory: An Introduction*. Englewood Cliffs, NJ: Prentice-Hall, 1970.
14. W.B. Powell, *Approximate Dynamic Programming: Solving the Curses of Dimensionality*, 2nd ed. Hoboken, NJ: Wiley, 2011.
15. L. Busoniu, R. Babuska, B. De Schutter, and D. Ernst, *Reinforcement Learning and Dynamic Programming Using Function Approximators*. Boca Raton, FL: CRC Press, 2010.

16. A.H. Land and A.G. Doig, "An Automatic Method of Solving Discrete Programming Problems," *Econometrica*, vol. 28, no. 3, pp. 497–520, 1960. DOI: 10.2307/1910129.

17. M.J. Kearns, Y. Mansour, and A.Y. Ng, "A Sparse Sampling Algorithm for Near-Optimal Planning in Large Markov Decision Processes," *Machine Learning*, vol. 49, no. 2-3, pp. 193–208, 2002. DOI: 10.1023/A:1017932429737.

18. A.G. Barto, S.J. Bradtke, and S.P. Singh, "Learning to Act Using Real-Time Dynamic Programming," *Artificial Intelligence*, vol. 72, no. 1-2, pp. 81–138, 1995. DOI: 10.1016/0004-3702(94)00011-O.

19. E.A. Hansen and S. Zilberstein, "LAO*: A Heuristic Search Algorithm That Finds Solutions with Loops," *Artificial Intelligence*, vol. 129, no. 1-2, pp. 35–62, 2001. DOI: 10.1016/S0004-3702(01)00106-0.

20. L. Kocsis and C. Szepesvári, "Bandit Based Monte-Carlo Planning," in *European Conference on Machine Learning (ECML)*, 2006.

21. C.B. Browne, E. Powley, D. Whitehouse, S.M. Lucas, P.I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A Survey of Monte Carlo Tree Search Methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012. DOI: 10.1109/TCIAIG.2012.2186810.

22. A. Couëtoux, J.-B. Hoock, N. Sokolovska, O. Teytaud, and N. Bonnard, "Continuous Upper Confidence Trees," in *Learning and Intelligent Optimization (LION)*, 2011.

23. R.J. Williams, "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning," *Machine Learning*, vol. 8, pp. 229–256, 1992. DOI: 10.1007/BF00992696.

24. J. Baxter and P.L. Bartlett, "Infinite-Horizon Policy-Gradient Estimation," *Journal of Artificial Intelligence Research*, vol. 15, pp. 319–350, 2001. DOI: 10.1613/jair.806.

25. P.-T. de Boer, D.P. Kroese, S. Mannor, and R.Y. Rubinstein, "A Tutorial on the Cross-Entropy Method," *Annals of Operations Research*, vol. 134, no. 1, pp. 19–67, 2005. DOI: 10.1007/s10479-005-5724-z.

26. R.Y. Rubinstein and D.P. Kroese, *The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation, and Machine Learning*. New York: Springer, 2004.

27. S. Mannor, R.Y. Rubinstein, and Y. Gat, "The Cross Entropy Method for Fast Policy Search," in *International Conference on Machine Learning (ICML)*, 2003.

28. I. Szita and A. Lörincz, "Learning Tetris Using the Noisy Cross-Entropy Method," *Neural Computation*, vol. 18, no. 12, pp. 2936–2941, 2006. DOI: 10.1162/neco.2006.18.12.2936.

29. ——, "Learning to Play Using Low-Complexity Rule-Based Policies: Illustrations Through Ms. Pac-Man," *Journal of Artificial Intelligence Research*, vol. 30, pp. 659–684, 2007. DOI: 10.1613/jair.2368.

30. D.B. Fogel, ed., *Evolutionary Computation: The Fossil Record*. New York: Wiley-IEEE Press, 1998.

31. J.H. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press, 1975.

32. L.M. Schmitt, "Theory of Genetic Algorithms," *Theoretical Computer Science*, vol. 259, no. 1-2, pp. 1–61, 2001. DOI: 10.1016/S0304-3975(00)00406-0.

33. ——, "Theory of Genetic Algorithms II: Models for Genetic Operators over the String-Tensor Representation of Populations and Convergence to Global Optima for Arbitrary Fitness Function Under Scaling," *Theoretical Computer Science*, vol. 310, no. 1-3, pp. 181–231, 2004. DOI: 10.1016/S0304-3975(03)00393-1.

34. J.R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press, 1992.