7 Exact Solution Methods

This chapter introduces a model known as a *Markov decision process* (*MDP*) to represent sequential decision problems where the effects of our actions are uncertain.[1] We begin with a description of the model, which specifies both the stochastic dynamics of the system as well as the utility associated with its evolution. Different algorithms can be used to compute the utility associated with a decision strategy and to search for an optimal strategy. Under certain assumptions, we can find exact solutions to MDPs. Later chapters will discuss approximation methods that tend to scale better to larger problems.

[1] Such models were originally studied in the 1950s. R. E. Bellman, *Dynamic Programming*. Princeton University Press, 1957. A modern treatment can be found in M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 2005.

7.1 Markov Decision Processes

In an MDP (algorithm 7.1), we choose action a_t at time t based on observing state s_t. We then receive a reward r_t. The *action space* $\mathcal{A}$ is the set of possible actions, and the *state space* $\mathcal{S}$ is the set of possible states. Some of the algorithms assume that these sets are finite, but this is not required in general. The state evolves probabilistically based on the current state and action we take. 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*.

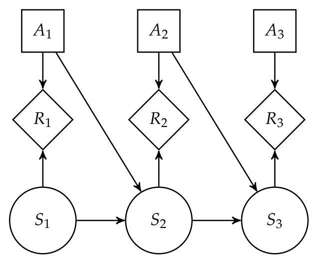


Figure 7.1. MDP decision network diagram.

An MDP can be represented using a decision network as shown in figure 7.1. There are informational edges (not shown here) from $A_{1:t-1}$ and $S_{1:t}$ to A_t. The utility function is decomposed into rewards $R_{1:t}$. We focus on *stationary* MDPs in which $P(S_{t+1} \mid S_t, A_t)$ and $P(R_t \mid S_t, A_t)$ do not vary with time. Stationary MDPs can be compactly represented by a dynamic decision diagram as shown in figure 7.2. The *state transition model* $T(s' \mid s, a)$ 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 action a from state s.

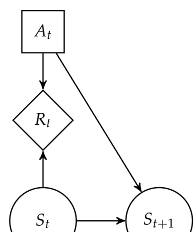


Figure 7.2. Stationary MDP decision network diagram. All MDPs have this general structure.

The reward function is a deterministic function of s and a because it represents an expectation, but rewards may be generated stochastically in the environment or even depend on the resulting next state.[2] Example 7.1 shows how to frame a collision avoidance problem as an MDP.

[2] For example, if the reward depends on the next state as given by $R(s, a, s')$, then the expected reward function would be

$$R(s,a) = \sum_{s'} T(s' \mid s,a) R(s,a,s')$$

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.

Given knowledge of the current state, we must decide whether an avoidance maneuver is required. The problem is challenging because the positions of the aircraft evolve probabilistically, and we want to make sure that we start our maneuver early enough to avoid collision, but late enough so that we avoid unnecessary maneuvering.

Example 7.1. Aircraft collision avoidance framed as an MDP. Many other real-world applications are discussed in D. J. White, "A Survey of Applications of Markov Decision Processes," *Journal of the Operational Research Society*, vol. 44, no. 11, pp. 1073–1096, 1993.

```
struct MDP
    γ  # discount factor
    𝒮  # state space
    𝒜  # action space
    T  # transition function
    R  # reward function
    TR # sample transition and reward
end
```

Algorithm 7.1. Data structure for an MDP. We will use the `TR` field later to sample the next state and reward given the current state and action: `s′, r = TR(s, a)`. In mathematical writing, MDPs are sometimes defined in terms of a tuple consisting of the various components of the MDP, written $(\mathcal{S}, \mathcal{A}, T, R, \gamma)$.

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

$$\sum_{t=1}^{n} r_t \tag{7.1}$$

The sum of rewards is sometimes called the *return*.

In an *infinite horizon* problem in which the number of decisions is unbounded, the sum of rewards can become infinite.[3] There are several ways to define utility in terms of individual rewards in infinite horizon problems. One way is to impose

[3] Suppose that 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.

a *discount factor* γ between 0 and 1. The utility is then given by

$$\sum_{t=1}^{\infty} \gamma^{t-1} r_t \tag{7.2}$$

This value is sometimes called the *discounted return*. 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 are 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*, also called the *average return*, given by

$$\lim_{n \to \infty} \frac{1}{n} \sum_{t=1}^{n} r_t \tag{7.3}$$

This formulation can be attractive because we do not have to choose a discount factor, but there is often no practical difference between this formulation and a discounted return with a discount factor close to 1. Because the discounted return is often computationally simpler to work with, we will focus on the discounted formulation.

A *policy* tells us what action to select given the past history of states and actions. The action to select at time t, given the *history* $h_t = (s_{1:t}, a_{1:t-1})$, is written $\pi_t(h_t)$. Because the future states and rewards depend only on the current state and action (as made apparent in the conditional independence assumptions in figure 7.1), we can restrict our attention to policies that depend only on the current state. In addition, we will primarily focus on *deterministic policies* because there is guaranteed to exist in MDPs an optimal policy that is deterministic. Later chapters discuss *stochastic policies*, where $\pi_t(a_t \mid s_t)$ denotes the probability that the policy assigns to taking action a_t in state s_t at time t.

In infinite horizon problems with stationary transitions and rewards, we can further restrict our attention to *stationary policies*, which do not depend on time. 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 only a couple of seconds remaining in the game. We can make stationary policies account for time by incorporating time as a state variable.

The expected utility of executing π from state s is denoted as $U^\pi(s)$. 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:[4]

$$\pi^*(s) = \arg\max_{\pi} U^\pi(s) \tag{7.4}$$

for all states s. Depending on the model, there may be multiple policies that are optimal. The value function associated with an optimal policy π^* is called the *optimal value function* and is denoted as U^*.

An optimal policy can be found by using a computational technique called *dynamic programming*,[5] which involves simplifying a complicated problem by breaking it down into simpler subproblems in a recursive manner. 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 and finding the longest common subsequence between two strings.[6] In general, algorithms that use dynamic programming for solving MDPs are much more efficient than brute force methods.

[4] Doing so is consistent with the maximum expected utility principle introduced in section 6.4.

[5] The term "dynamic programming" was coined by the American mathematician Richard Ernest Bellman (1920–1984). Dynamic refers to the fact that the problem is time-varying and programming refers to a methodology to find an optimal program or decision strategy. R. Bellman, *Eye of the Hurricane: An Autobiography*. World Scientific, 1984.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.

7.2 Policy Evaluation

Before we discuss how to go about computing an optimal policy, we will discuss *policy evaluation*, where we compute the value function U^π. Policy evaluation can be done iteratively. If the policy is executed for a single step, the utility is $U_1^\pi(s) = R(s, \pi(s))$. Further steps can be obtained from the *lookahead* equation:

$$U_{k+1}^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s' \mid s, \pi(s)) U_k^\pi(s') \tag{7.5}$$

This equation is implemented in algorithm 7.2. Iterative policy evaluation is implemented in algorithm 7.3. Several iterations are shown in figure 7.3.

The value function U^π can be computed to an arbitrary precision given sufficient iterations of the lookahead equation. Convergence is guaranteed because the update in equation (7.5) is a *contraction mapping* (reviewed in appendix A.15).[7] At convergence, the following equality holds:

$$U^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s' \mid s, \pi(s)) U^\pi(s') \tag{7.6}$$

[7] See exercise 7.12.

```
function lookahead(𝒫::MDP, U, s, a)
    𝒮, T, R, γ = 𝒫.𝒮, 𝒫.T, 𝒫.R, 𝒫.γ
    return R(s,a) + γ*sum(T(s,a,s′)*U(s′) for s′ in 𝒮)
end
function lookahead(𝒫::MDP, U::Vector, s, a)
    𝒮, T, R, γ = 𝒫.𝒮, 𝒫.T, 𝒫.R, 𝒫.γ
    return R(s,a) + γ*sum(T(s,a,s′)*U[i] for (i,s′) in enumerate(𝒮))
end
```

Algorithm 7.2. Functions for computing the lookahead state-action value from a state `s` given an action `a` using an estimate of the value function `U` for the MDP $\mathcal{P}$. The second version handles the case when `U` is a vector.

```
function iterative_policy_evaluation(𝒫::MDP, π, k_max)
    𝒮, T, R, γ = 𝒫.𝒮, 𝒫.T, 𝒫.R, 𝒫.γ
    U = [0.0 for s in 𝒮]
    for k in 1:k_max
        U = [lookahead(𝒫, U, s, π(s)) for s in 𝒮]
    end
    return U
end
```

Algorithm 7.3. Iterative policy evaluation, which iteratively computes the value function for a policy `π` for MDP $\mathcal{P}$ with discrete state and action spaces using `k_max` iterations.

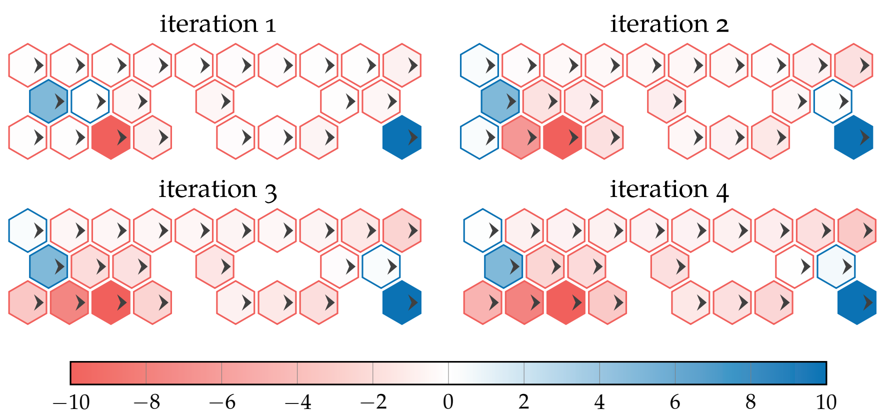


Figure 7.3. Iterative policy evaluation used to evaluate an east-moving policy on the hex world problem (see appendix F.1). The arrows indicate the direction recommended by the policy (i.e., always move east), and the colors indicate the values associated with the states. The values change with each iteration.

This equality is called the *Bellman expectation equation*.[8]

Policy evaluation can be done without iteration by solving the system of equations in the Bellman expectation equation directly. Equation (7.6) defines a set of $|S|$ linear equations with $|S|$ unknowns corresponding to the values at each state. One way to solve this system of equations is to first convert it into matrix form:

$$\mathbf{U}^\pi = \mathbf{R}^\pi + \gamma \mathbf{T}^\pi \mathbf{U}^\pi \tag{7.7}$$

where $\mathbf{U}^\pi$ and $\mathbf{R}^\pi$ are the utility and reward functions represented in vector form with $|\mathcal{S}|$ components. The $|\mathcal{S}| \times |\mathcal{S}|$ matrix $\mathbf{T}^\pi$ contains state transition probabilities where T^π_{ij} is the probability of transitioning from the ith state to the jth state.

The value function is obtained as follows:

$$\mathbf{U}^\pi - \gamma \mathbf{T}^\pi \mathbf{U}^\pi = \mathbf{R}^\pi \tag{7.8}$$

$$(\mathbf{I} - \gamma \mathbf{T}^\pi)\mathbf{U}^\pi = \mathbf{R}^\pi \tag{7.9}$$

$$\mathbf{U}^\pi = (\mathbf{I} - \gamma \mathbf{T}^\pi)^{-1}\mathbf{R}^\pi \tag{7.10}$$

This method is implemented in algorithm 7.4. Solving for $\mathbf{U}^\pi$ in this way requires $O(|\mathcal{S}|^3)$ time. The method is used to evaluate a policy in figure 7.4.

[8] This equation is named for Richard E. Bellman, one of the pioneers of dynamic programming. R. E. Bellman, *Dynamic Programming*. Princeton University Press, 1957.

```
function policy_evaluation(𝒫::MDP, π)
    𝒮, R, T, γ = 𝒫.𝒮, 𝒫.R, 𝒫.T, 𝒫.γ
    R′ = [R(s, π(s)) for s in 𝒮]
    T′ = [T(s, π(s), s′) for s in 𝒮, s′ in 𝒮]
    return (I - γ*T′)\R′
end
```

Algorithm 7.4. Exact policy evaluation, which computes the value function for a policy π for an MDP 𝒫 with discrete state and action spaces.

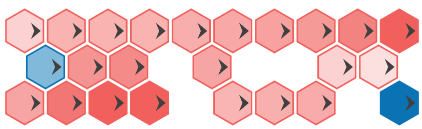


Figure 7.4. Exact policy evaluation used to evaluate an east-moving policy for the hex world problem. The exact solution contains lower values than what was contained in the first few steps of iterative policy evaluation in figure 7.3. If we ran iterative policy evaluation for more iterations, it would converge to the same value function.

7.3 *Value Function Policies*

The previous section showed how to compute a value function associated with a policy. This section shows how to extract a policy from a value function, which we later use when generating optimal policies. Given a value function U, which may or may not correspond to the optimal value function, we can construct a policy π that maximizes the lookahead equation introduced in equation (7.5):

$$\pi(s) = \arg\max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U(s') \right) \tag{7.11}$$

We refer to this policy as a *greedy policy* with respect to U. If U is the optimal value function, then the extracted policy is optimal. Algorithm 7.5 implements this idea.

An alternative way to represent a policy is to use the *action value function*, sometimes called the *Q-function*. The action value function represents the expected return when starting in state s, taking action a, and then continuing with the greedy policy with respect to Q:

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U(s') \tag{7.12}$$

From this action value function, we can obtain the value function,

$$U(s) = \max_a Q(s,a) \tag{7.13}$$

as well as the policy,

$$\pi(s) = \arg\max_a Q(s,a) \tag{7.14}$$

Storing Q explicitly for discrete problems requires $O(|\mathcal{S}| \times |\mathcal{A}|)$ storage instead of $O(|\mathcal{S}|)$ storage for U, but we do not have to use R and T to extract the policy.

Policies can also be represented using the *advantage function*, which quantifies the advantage of taking an action in comparison to the greedy action. It is defined in terms of the difference between Q and U:

$$A(s,a) = Q(s,a) - U(s) \tag{7.15}$$

Greedy actions have zero advantage, and nongreedy actions have negative advantage. Some algorithms that we will discuss later in the book use U representations, but others will use Q or A.

```
struct ValueFunctionPolicy
    𝒫 # problem
    U # utility function
end

function greedy(𝒫::MDP, U, s)
    u, a = findmax(a->lookahead(𝒫, U, s, a), 𝒫.𝒜)
    return (a=a, u=u)
end

(π::ValueFunctionPolicy)(s) = greedy(π.𝒫, π.U, s).a
```

Algorithm 7.5. A value function policy extracted from a value function `U` for an MDP 𝒫. The `greedy` function will be used in other algorithms.

7.4 Policy Iteration

Policy iteration (algorithm 7.6) is one way to compute an optimal policy. It involves iterating between policy evaluation (section 7.2) and policy improvement through a greedy policy (algorithm 7.5). Policy iteration is guaranteed to converge given any initial policy. It converges in a finite number of iterations because there are finitely many policies and every iteration improves the policy if it can be improved. Although the number of possible policies is exponential in the number of states, policy iteration often converges quickly. Figure 7.5 demonstrates policy iteration on the hex world problem.

```
struct PolicyIteration
    π # initial policy
    k_max # maximum number of iterations
end

function solve(M::PolicyIteration, 𝒫::MDP)
    π, 𝒮 = M.π, 𝒫.𝒮
    for k = 1:M.k_max
        U = policy_evaluation(𝒫, π)
        π′ = ValueFunctionPolicy(𝒫, U)
        if all(π(s) == π′(s) for s in 𝒮)
            break
        end
        π = π′
    end
    return π
end
```

Algorithm 7.6. Policy iteration, which iteratively improves an initial policy `π` to obtain an optimal policy for an MDP 𝒫 with discrete state and action spaces.

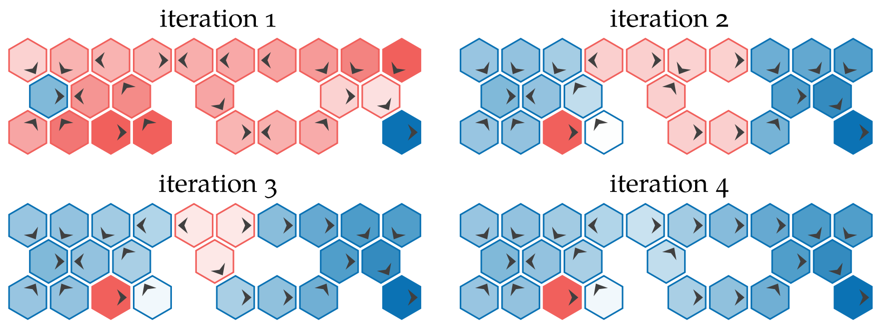


Figure 7.5. Policy iteration used to iteratively improve an initially east-moving policy in the hex world problem to obtain an optimal policy. In the first iteration, we see the value function associated with the east-moving policy and arrows indicating the policy that is greedy with respect to that value function. Policy iteration converges in four iterations; if we ran for a fifth or more iterations, we would get the same policy.

Policy iteration tends to be expensive because we must evaluate the policy in each iteration. A variation of policy iteration called *modified policy iteration*[9] approximates the value function using iterative policy evaluation instead of exact policy evaluation. We can choose the number of policy evaluation iterations between steps of policy improvement. If we use only one iteration between steps, then this approach is identical to value iteration.

[9] M. L. Puterman and M. C. Shin, "Modified Policy Iteration Algorithms for Discounted Markov Decision Problems," *Management Science*, vol. 24, no. 11, pp. 1127–1137, 1978.

7.5 Value Iteration

Value iteration is an alternative to policy iteration that is often used because of its simplicity. Unlike policy improvement, value iteration updates the value function directly. It begins with any bounded value function U, meaning that $|U(s)| < \infty$ for all s. One common initialization is $U(s) = 0$ for all s.

The value function can be improved by applying the *Bellman backup*, also called the *Bellman update*:[10]

$$U_{k+1}(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U_k(s') \right) \tag{7.16}$$

[10] It is referred to as a backup operation because it transfers information back to a state from its future states.

This backup procedure is implemented in algorithm 7.7.

```
function backup(𝒫::MDP, U, s)
    return maximum(lookahead(𝒫, U, s, a) for a in 𝒫.𝒜)
end
```

Algorithm 7.7. The backup procedure applied to an MDP 𝒫, which improves a value function `U` at state `s`.

Repeated application of this update is guaranteed to converge to the optimal value function. Like iterative policy evaluation, we can use the fact that the update is a contraction mapping to prove convergence.[11] This optimal policy is guaranteed to satisfy the *Bellman optimality equation*:

$$U^*(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U^*(s') \right) \tag{7.17}$$

[11] See exercise 7.13.

Further applications of the Bellman backup once this equality holds do not change the value function. An optimal policy can be extracted from U^* using equation (7.11). Value iteration is implemented in algorithm 7.8 and is applied to the hex world problem in figure 7.6.

The implementation in algorithm 7.8 stops after a fixed number of iterations, but it is also common to terminate the iterations early based on the maximum change in value $\|U_{k+1} - U_k\|_\infty$, called the *Bellman residual*. If the Bellman residual drops below a threshold δ, then the iterations terminate. A Bellman residual of δ guarantees that the optimal value function obtained by value iteration is within $\epsilon = \delta\gamma/(1-\gamma)$ of U^*.[12] Discount factors closer to 1 significantly inflate this error, leading to slower convergence. If we heavily discount future reward (γ closer to 0), then we do not need to iterate as much into the future. This effect is demonstrated in example 7.2.

[12] See exercise 7.8.

Knowing the maximum deviation of the estimated value function from the optimal value function, $\|U_k - U^*\|_\infty < \epsilon$, allows us to bound the maximum deviation of reward obtained under the extracted policy π from an optimal policy π^*. This *policy loss* $\|U^\pi - U^*\|_\infty$ is bounded by $2\epsilon\gamma/(1-\gamma)$.[13]

[13] S. P. Singh and R. C. Yee, "An Upper Bound on the Loss from Approximate Optimal-Value Functions," *Machine Learning*, vol. 16, no. 3, pp. 227–233, 1994.

```
struct ValueIteration
    k_max # maximum number of iterations
end

function solve(M::ValueIteration, 𝒫::MDP)
    U = [0.0 for s in 𝒫.𝒮]
    for k = 1:M.k_max
        U = [backup(𝒫, U, s) for s in 𝒫.𝒮]
    end
    return ValueFunctionPolicy(𝒫, U)
end
```

Algorithm 7.8. Value iteration, which iteratively improves a value function `U` to obtain an optimal policy for an MDP `𝒫` with discrete state and action spaces. The method terminates after `k_max` iterations.

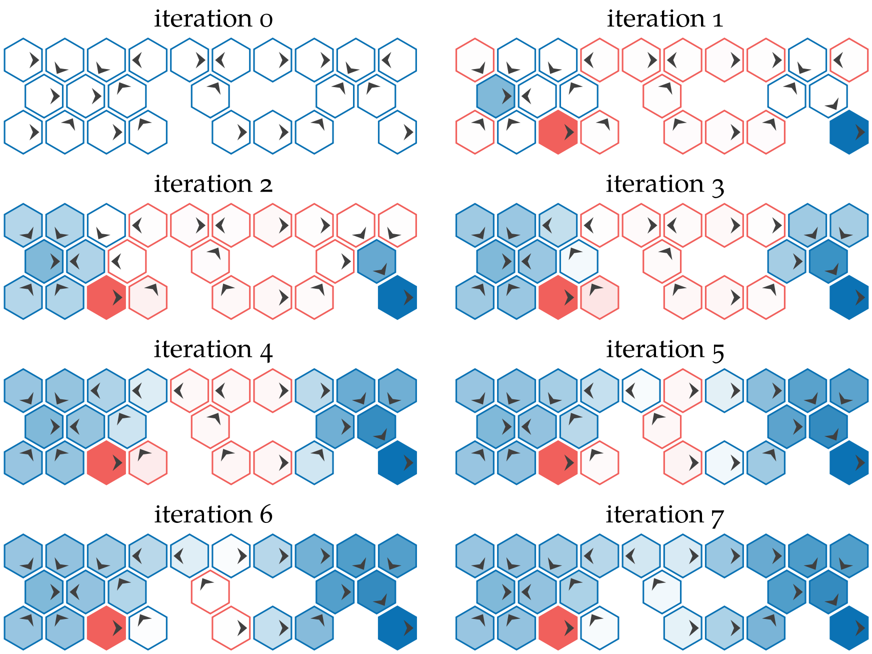
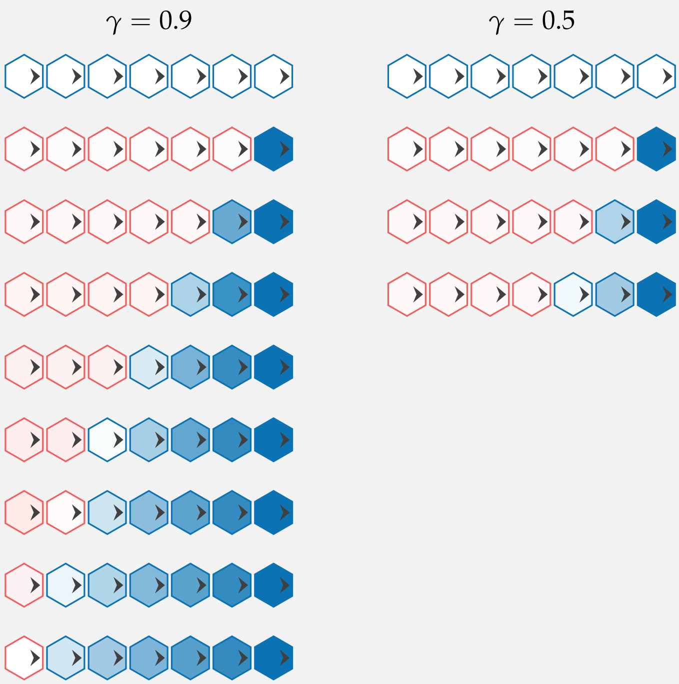


Figure 7.6. Value iteration in the hex world problem to obtain an optimal policy. Each hex is colored according to the value function, and arrows indicate the policy that is greedy with respect to that value function.

Consider a simple variation of the hex world problem, consisting of a straight line of tiles with a single consuming tile at the end producing a reward of 10. The discount factor directly affects the rate at which reward from the consuming tile propagates down the line to the other tiles, and thus how quickly value iteration converges.



Example 7.2. The effect of the discount factor on convergence of value iteration. In each case, value iteration was run until the Bellman residual was less than 1.

7.6 *Asynchronous Value Iteration*

Value iteration tends to be computationally intensive, as every entry in the value function U_k is updated in each iteration to obtain U_{k+1}. In *asynchronous value iteration*, only a subset of the states are updated with each iteration. Asynchronous value iteration is still guaranteed to converge on the optimal value function, provided that each state is updated an infinite number of times.

One common asynchronous value iteration method, *Gauss-Seidel value iteration* (algorithm 7.9), sweeps through an ordering of the states and applies the Bellman update in place:

$$U(s) \leftarrow \max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U(s') \right) \tag{7.18}$$

The computational savings lies in not having to construct a second value function in memory with each iteration. Gauss-Seidel value iteration can converge more quickly than standard value iteration, depending on the ordering chosen.[14] In some problems, the state contains a time index that increments deterministically forward in time. If we apply Gauss-Seidel value iteration starting at the last time index and work our way backward, this process is sometimes called *backward induction value iteration*. An example of the impact of the state ordering is given in example 7.3.

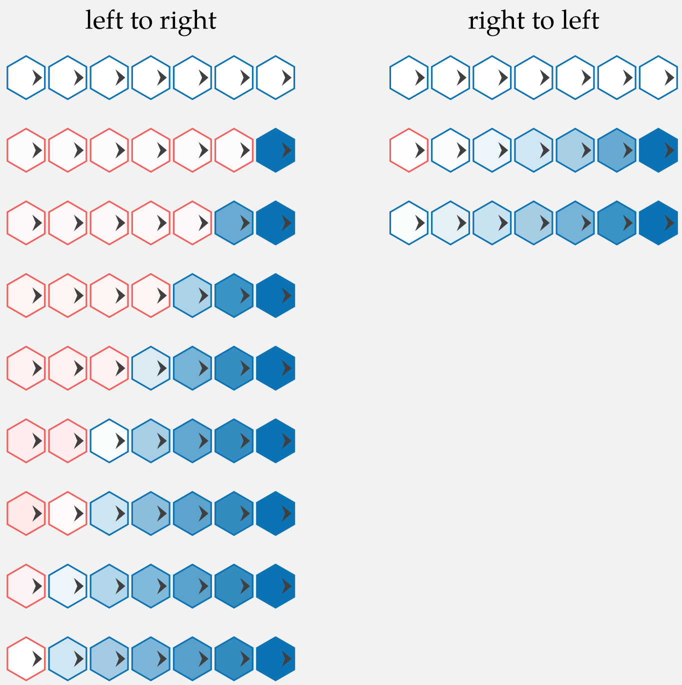
[14] A poor ordering in Gauss-Seidel value iteration cannot cause the algorithm to be slower than standard value iteration.

```
struct GaussSeidelValueIteration
    k_max # maximum number of iterations
end

function solve(M::GaussSeidelValueIteration, 𝒫::MDP)
    U = [0.0 for s in 𝒫.𝒮]
    for k = 1:M.k_max
        for (i, s) in enumerate(𝒫.𝒮)
            U[i] = backup(𝒫, U, s)
        end
    end
    return ValueFunctionPolicy(𝒫, U)
end
```

Algorithm 7.9. Asynchronous value iteration, which updates states in a different manner than value iteration, often saving computation time. The method terminates after `k_max` iterations.

Consider the linear variation of the hex world problem from example 7.2. We can solve the same problem using asynchronous value iteration. The ordering of the states directly affects the rate at which reward from the consuming tile propagates down the line to the other tiles, and thus how quickly the method converges.



Example 7.3. The effect of the state ordering on convergence of asynchronous value iteration. In this case, evaluating right to left allows convergence to occur in far fewer iterations.

7.7 *Linear Program Formulation*

The problem of finding an optimal policy can be formulated as a *linear program*, which is an optimization problem with a linear objective function and a set of linear equality or inequality constraints. Once a problem is represented as a linear program, we can use one of many linear programming solvers.[15]

To show how we can convert the Bellman optimality equation into a linear program, we begin by replacing the equality in the Bellman optimality equation with a set of inequality constraints while minimizing $U(s)$ at each state s:[16]

$$\begin{aligned} &\text{minimize } \sum_s U(s) \\ &\text{subject to } U(s) \geq \max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U(s') \right) \text{ for all } s \end{aligned} \tag{7.19}$$

The variables in the optimization are the utilities at each state. Once we know those utilities, we can extract an optimal policy using equation (7.11).

The maximization in the inequality constraints can be replaced by a set of linear constraints, making it a linear program:

$$\begin{aligned} &\text{minimize } \sum_s U(s) \\ &\text{subject to } U(s) \geq R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U(s') \text{ for all } s \text{ and } a \end{aligned} \tag{7.20}$$

In the linear program shown in equation (7.20), the number of variables is equal to the number of states and the number of constraints is equal to the number of states times the number of actions. Because linear programs can be solved in polynomial time,[17] MDPs can be solved in polynomial time as well. Although a linear programming approach provides this asymptotic complexity guarantee, it is often more efficient in practice to simply use value iteration. Algorithm 7.10 provides an implementation of this.

[15] For an overview of linear programming, see R. Vanderbei, *Linear Programming, Foundations and Extensions*, 4th ed. Springer, 2014.

[16] Intuitively, we want to push the value $U(s)$ at all states s down in order to convert the inequality constraints into equality constraints. Hence, we minimize the sum of all utilities.

[17] This was proved by L. G. Khachiyan, "Polynomial Algorithms in Linear Programming," *USSR Computational Mathematics and Mathematical Physics*, vol. 20, no. 1, pp. 53–72, 1980. Modern algorithms tend to be more efficient in practice.

7.8 *Linear Systems with Quadratic Reward*

So far, we have assumed discrete state and action spaces. This section relaxes this assumption, allowing for continuous, vector-valued states and actions. The Bellman optimality equation for discrete problems can be modified as follows:[18]

[18] This section assumes that the problem is undiscounted and finite horizon, but these equations can be easily generalized.

```
struct LinearProgramFormulation end

function tensorform(𝒫::MDP)
    𝒮, 𝒜, R, T = 𝒫.𝒮, 𝒫.𝒜, 𝒫.R, 𝒫.T
    𝒮′ = eachindex(𝒮)
    𝒜′ = eachindex(𝒜)
    R′ = [R(s,a) for s in 𝒮, a in 𝒜]
    T′ = [T(s,a,s′) for s in 𝒮, a in 𝒜, s′ in 𝒮]
    return 𝒮′, 𝒜′, R′, T′
end

solve(𝒫::MDP) = solve(LinearProgramFormulation(), 𝒫)

function solve(M::LinearProgramFormulation, 𝒫::MDP)
    𝒮, 𝒜, R, T = tensorform(𝒫)
    model = Model(GLPK.Optimizer)
    @variable(model, U[𝒮])
    @objective(model, Min, sum(U))
    @constraint(model, [s=𝒮,a=𝒜], U[s] ≥ R[s,a] + 𝒫.γ*T[s,a,:]⋅U)
    optimize!(model)
    return ValueFunctionPolicy(𝒫, value.(U))
end
```

Algorithm 7.10. A method for solving a discrete MDP using a linear program formulation. For convenience in specifying the linear program, we define a function for converting an MDP into its tensor form, where the states and actions consist of integer indices, the reward function is a matrix, and the transition function is a three-dimensional tensor. It uses the `JuMP.jl` package for mathematical programming. The optimizer is set to use `GLPK.jl`, but others can be used instead. We also define the default solve behavior for MDPs to use this formulation.

$$U_{h+1}(\mathbf{s}) = \max_{\mathbf{a}}\left(R(\mathbf{s},\mathbf{a}) + \int T(\mathbf{s}' \mid \mathbf{s},\mathbf{a})U_h(\mathbf{s}')\,\mathrm{d}\mathbf{s}'\right) \tag{7.21}$$

where s and a in equation (7.16) are replaced with their vector equivalents, the summation is replaced with an integral, and T provides a probability density rather than a probability mass. Computing equation (7.21) is not straightforward for an arbitrary continuous transition distribution and reward function.

In some cases, exact solution methods do exist for MDPs with continuous state and action spaces.[19] In particular, if a problem has *linear dynamics* and has *quadratic reward*, then the optimal policy can be efficiently found in closed form. Such a system is known in control theory as a *linear quadratic regulator* (*LQR*) and has been well studied.[20]

A problem has linear dynamics if the next state $\mathbf{s}'$ after taking action $\mathbf{a}$ from state $\mathbf{s}$ is determined by an equation of the form:

$$\mathbf{s}' = \mathbf{T}_s\mathbf{s} + \mathbf{T}_a\mathbf{a} + \mathbf{w} \tag{7.22}$$

where $\mathbf{T}_s$ and $\mathbf{T}_a$ are matrices and $\mathbf{w}$ is a random disturbance drawn from a zero mean, finite variance distribution that does not depend on $\mathbf{s}$ and $\mathbf{a}$. One common choice is the multivariate Gaussian.

[19] For a detailed overview, see chapter 4 of volume I of D. P. Bertsekas, *Dynamic Programming and Optimal Control*. Athena Scientific, 2007.

[20] For a compact summary of LQR and other related control problems, see A. Shaiju and I. R. Petersen, "Formulas for Discrete Time LQR, LQG, LEQG and Minimax LQG Optimal Control Problems," *IFAC Proceedings Volumes*, vol. 41, no. 2, pp. 8773–8778, 2008.

A reward function is quadratic if it can be written in the form:[21]

$$R(\mathbf{s}, \mathbf{a}) = \mathbf{s}^\top \mathbf{R}_s \mathbf{s} + \mathbf{a}^\top \mathbf{R}_a \mathbf{a} \tag{7.23}$$

[21] A third term, $2\mathbf{s}^\top \mathbf{R}_{sa}\mathbf{a}$, can also be included. For an example, see Shaiju and Petersen (2008).

where $\mathbf{R}_s$ and $\mathbf{R}_a$ are matrices that determine how state and action component combinations contribute reward. We additionally require that $\mathbf{R}_s$ be negative semidefinite and $\mathbf{R}_a$ be negative definite. Such a reward function penalizes states and actions that deviate from $\mathbf{0}$.

Problems with linear dynamics and quadratic reward are common in control theory where one often seeks to regulate a process such that it does not deviate far from a desired value. The quadratic cost assigns a much higher cost to states far from the origin than to those near it. The optimal policy for a problem with linear dynamics and quadratic reward has an analytic, closed-form solution. Many MDPs can be approximated with linear quadratic MDPs and solved, often yielding reasonable policies for the original problem.

Substituting the transition and reward functions into equation (7.21) produces

$$U_{h+1}(\mathbf{s}) = \max_{\mathbf{a}} \left(\mathbf{s}^\top \mathbf{R}_s \mathbf{s} + \mathbf{a}^\top \mathbf{R}_a \mathbf{a} + \int p(\mathbf{w}) U_h(\mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a} + \mathbf{w}) \, d\mathbf{w} \right) \tag{7.24}$$

where $p(\mathbf{w})$ is the probability density of the random, zero-mean disturbance $\mathbf{w}$.

The optimal one-step value function is

$$U_1(\mathbf{s}) = \max_{\mathbf{a}} \left(\mathbf{s}^\top \mathbf{R}_s \mathbf{s} + \mathbf{a}^\top \mathbf{R}_a \mathbf{a} \right) = \mathbf{s}^\top \mathbf{R}_s \mathbf{s} \tag{7.25}$$

for which the optimal action is $\mathbf{a} = \mathbf{0}$.

We will show through induction that $U_h(\mathbf{s})$ has a quadratic form, $\mathbf{s}^\top \mathbf{V}_h \mathbf{s} + q_h$, with symmetric matrices $\mathbf{V}_h$. For the one-step value function, $\mathbf{V}_1 = \mathbf{R}_s$ and $q_1 = 0$. Substituting this quadratic form into equation (7.24) yields

$$U_{h+1}(\mathbf{s}) = \mathbf{s}^\top \mathbf{R}_s \mathbf{s} + \max_{\mathbf{a}} \left(\mathbf{a}^\top \mathbf{R}_a \mathbf{a} + \int p(\mathbf{w}) \left((\mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a} + \mathbf{w})^\top \mathbf{V}_h (\mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a} + \mathbf{w}) + q_h \right) d\mathbf{w} \right) \tag{7.26}$$

This can be simplified by expanding and using the fact that $\int p(\mathbf{w}) \, d\mathbf{w} = 1$ and $\int \mathbf{w} p(\mathbf{w}) \, d\mathbf{w} = 0$:

$$\begin{aligned} U_{h+1}(\mathbf{s}) = {} & \mathbf{s}^\top \mathbf{R}_s \mathbf{s} + \mathbf{s}^\top \mathbf{T}_s^\top \mathbf{V}_h \mathbf{T}_s \mathbf{s} \\ & + \max_{\mathbf{a}} \left(\mathbf{a}^\top \mathbf{R}_a \mathbf{a} + 2\mathbf{s}^\top \mathbf{T}_s^\top \mathbf{V}_h \mathbf{T}_a \mathbf{a} + \mathbf{a}^\top \mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_a \mathbf{a} \right) \\ & + \int p(\mathbf{w}) \left(\mathbf{w}^\top \mathbf{V}_h \mathbf{w} \right) d\mathbf{w} + q_h \end{aligned} \tag{7.27}$$

We can obtain the optimal action by differentiating with respect to $\mathbf{a}$ and setting it to $\mathbf{0}$:[22]

$$\begin{aligned}\mathbf{0} &= \left(\mathbf{R}_a + \mathbf{R}_a^\top\right)\mathbf{a} + 2\mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_s \mathbf{s} + \left(\mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_a + \left(\mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_a\right)^\top\right)\mathbf{a} \\ &= 2\mathbf{R}_a \mathbf{a} + 2\mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_s \mathbf{s} + 2\mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_a \mathbf{a}\end{aligned} \tag{7.28}$$

[22] Recall that

$$\begin{aligned}\nabla_{\mathbf{x}} \mathbf{A}\mathbf{x} &= \mathbf{A}^\top \\ \nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A}\mathbf{x} &= (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}\end{aligned}$$

Solving for the optimal action yields[23]

$$\mathbf{a} = -\left(\mathbf{R}_a + \mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_a\right)^{-1} \mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_s \mathbf{s} \tag{7.29}$$

[23] The matrix $\mathbf{R}_a + \mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_a$ is negative definite, and thus invertible.

Substituting the optimal action into $U_{h+1}(\mathbf{s})$ yields the quadratic form that we were seeking, $U_{h+1}(\mathbf{s}) = \mathbf{s}^\top \mathbf{V}_{h+1} \mathbf{s} + q_{h+1}$, with[24]

$$\mathbf{V}_{h+1} = \mathbf{R}_s + \mathbf{T}_s^\top \mathbf{V}_h^\top \mathbf{T}_s - \left(\mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_s\right)^\top \left(\mathbf{R}_a + \mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_a\right)^{-1} \left(\mathbf{T}_a^\top \mathbf{V}_h \mathbf{T}_s\right) \tag{7.30}$$

[24] This equation is sometimes referred to as the *discrete-time Riccati equation*, named after the Venetian mathematician Jacopo Riccati (1676–1754).

and

$$q_{h+1} = \sum_{i=1}^{h} \mathbb{E}_{\mathbf{w}}\left[\mathbf{w}^\top \mathbf{V}_i \mathbf{w}\right] \tag{7.31}$$

If $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \mathbf{\Sigma})$, then

$$q_{h+1} = \sum_{i=1}^{h} \mathrm{Tr}(\mathbf{\Sigma} \mathbf{V}_i) \tag{7.32}$$

We can compute $\mathbf{V}_h$ and q_h up to any horizon h starting from $\mathbf{V}_1 = \mathbf{R}_s$ and $q_1 = 0$ and iterating using equations (7.30) and (7.31). The optimal action for an h-step policy comes directly from equation (7.29):

$$\pi_h(\mathbf{s}) = -\left(\mathbf{T}_a^\top \mathbf{V}_{h-1} \mathbf{T}_a + \mathbf{R}_a\right)^{-1} \mathbf{T}_a^\top \mathbf{V}_{h-1} \mathbf{T}_s \mathbf{s} \tag{7.33}$$

Note that the optimal action is independent of the zero-mean disturbance distribution.[25] The variance of the disturbance, however, does affect the expected utility. Algorithm 7.11 provides an implementation. Example 7.4 demonstrates this process on a simple problem with linear Gaussian dynamics.

[25] In this case, we can replace the random disturbances with its expected value without changing the optimal policy. This property is known as *certainty equivalence*.

7.9 Summary

- Discrete MDPs with bounded rewards can be solved exactly through dynamic programming.

```julia
struct LinearQuadraticProblem
    Ts # transition matrix with respect to state
    Ta # transition matrix with respect to action
    Rs # reward matrix with respect to state (negative semidefinite)
    Ra # reward matrix with respect to action (negative definite)
    h_max # horizon
end

function solve(𝒫::LinearQuadraticProblem)
    Ts, Ta, Rs, Ra, h_max = 𝒫.Ts, 𝒫.Ta, 𝒫.Rs, 𝒫.Ra, 𝒫.h_max
    V = zeros(size(Rs))
    πs = Any[s -> zeros(size(Ta, 2))]
    for h in 2:h_max
        V = Ts'*(V - V*Ta*((Ta'*V*Ta + Ra) \ Ta'*V))*Ts + Rs
        L = -(Ta'*V*Ta + Ra) \ Ta' * V * Ts
        push!(πs, s -> L*s)
    end
    return πs
end
```

Algorithm 7.11. A method that computes an optimal policy for an `h_max`-step horizon MDP with stochastic linear dynamics parameterized by matrices `Ts` and `Ta` and quadratic reward parameterized by matrices `Rs` and `Ra`. The method returns a vector of policies where entry h produces the optimal first action in an h-step policy.

- Policy evaluation for such problems can be done exactly through matrix inversion or can be approximated by an iterative algorithm.
- Policy iteration can be used to solve for optimal policies by iterating between policy evaluation and policy improvement.
- Value iteration and asynchronous value iteration save computation by directly iterating the value function.
- The problem of finding an optimal policy can be framed as a linear program and solved in polynomial time.
- Continuous problems with linear transition functions and quadratic rewards can be solved exactly.

7.10 Exercises

Exercise 7.1. Show that for an infinite sequence of constant rewards ($r_t = r$ for all t), the infinite horizon discounted return converges to $r/(1-\gamma)$.

Example 7.4. Solving a finite horizon MDP with a linear transition function and quadratic reward. The illustration shows the progression of the system from $[-10, 0]$. The blue contour lines show the Gaussian distributions over the state at each iteration. The initial belief is circular, but it gets distorted to a noncircular shape as we propagate the belief forward using the Kalman filter.

Consider a continuous MDP where the state is composed of a scalar position and velocity $s = [x, v]$. Actions are scalar accelerations a that are each executed over a time step $\Delta t = 1$. Find an optimal five-step policy from $s_0 = [-10, 0]$, given a quadratic reward:

$$R(\mathbf{s}, a) = -x^2 - v^2 - 0.5a^2$$

such that the system tends toward rest at $s = \mathbf{0}$.

The transition dynamics are

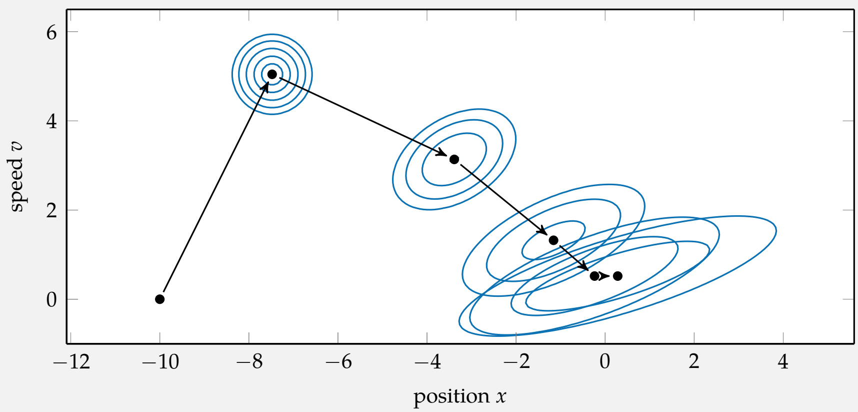
$$\begin{bmatrix} x' \\ v' \end{bmatrix} = \begin{bmatrix} x + v\Delta t + \frac{1}{2}a\Delta t^2 + w_1 \\ v \quad + \quad a\Delta t + w_2 \end{bmatrix} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ v \end{bmatrix} + \begin{bmatrix} 0.5\Delta t^2 \\ \Delta t \end{bmatrix} [a] + \mathbf{w}$$

where $\mathbf{w}$ is drawn from a zero-mean multivariate Gaussian distribution with covariance $0.1\mathbf{I}$.

The reward matrices are $\mathbf{R}_s = -\mathbf{I}$ and $\mathbf{R}_a = -[0.5]$.

The resulting optimal policies are:

$$\pi_1(\mathbf{s}) = \begin{bmatrix} 0 & 0 \end{bmatrix} \mathbf{s}$$
$$\pi_2(\mathbf{s}) = \begin{bmatrix} -0.286 & -0.857 \end{bmatrix} \mathbf{s}$$
$$\pi_3(\mathbf{s}) = \begin{bmatrix} -0.462 & -1.077 \end{bmatrix} \mathbf{s}$$
$$\pi_4(\mathbf{s}) = \begin{bmatrix} -0.499 & -1.118 \end{bmatrix} \mathbf{s}$$
$$\pi_5(\mathbf{s}) = \begin{bmatrix} -0.504 & -1.124 \end{bmatrix} \mathbf{s}$$



Solution: We can prove that the infinite sequence of discounted constant rewards converges to $r/(1-\gamma)$ in the following steps:

$$\begin{aligned}\sum_{t=1}^{\infty} \gamma^{t-1} r_t &= r + \gamma^1 r + \gamma^2 r + \cdots \\ &= r + \gamma \sum_{t=1}^{\infty} \gamma^{t-1} r_t\end{aligned}$$

We can move the summation to the left side and factor out $(1-\gamma)$:

$$\begin{aligned}(1-\gamma) \sum_{t=1}^{\infty} \gamma^{t-1} r &= r \\ \sum_{t=1}^{\infty} \gamma^{t-1} r &= \frac{r}{1-\gamma}\end{aligned}$$

Exercise 7.2. Suppose we have an MDP consisting of five states, $s_{1:5}$, and two actions, to stay (a_S) and continue (a_C). We have the following:

$$\begin{aligned}T(s_i \mid s_i, a_S) &= 1 \text{ for } i \in \{1,2,3,4\} \\ T(s_{i+1} \mid s_i, a_C) &= 1 \text{ for } i \in \{1,2,3,4\} \\ T(s_5 \mid s_5, a) &= 1 \text{ for all actions } a \\ R(s_i, a) &= 0 \text{ for } i \in \{1,2,3,5\} \text{ and for all actions } a \\ R(s_4, a_S) &= 0 \\ R(s_4, a_C) &= 10\end{aligned}$$

What is the discount factor γ if the optimal value $U^*(s_1) = 1$?

Solution: The optimal value of $U^*(s_1)$ is associated with following the optimal policy π^* starting from s_1. Given the transition model, the optimal policy from s_1 is to continue until reaching s_5, which is a terminal state where we can no longer transition to another state or accumulate additional reward. Thus, the optimal value of s_1 can be computed as

$$\begin{aligned}U^*(s_1) &= \sum_{t=1}^{\infty} \gamma^{t-1} r_t \\ U^*(s_1) &= R(s_1, a_C) + \gamma^1 R(s_2, a_C) + \gamma^2 R(s_3, a_C) + \gamma^3 R(s_4, a_C) + \gamma^4 R(s_5, a_C) + \cdots \\ U^*(s_1) &= 0 + \gamma^1 \times 0 + \gamma^2 \times 0 + \gamma^3 \times 10 + \gamma^4 \times 0 + 0 \\ 1 &= 10\gamma^3\end{aligned}$$

Thus, the discount factor is $\gamma = 0.1^{1/3} \approx 0.464$.

Exercise 7.3. What is the time complexity of performing k steps of iterative policy evaluation?

Solution: Iterative policy evaluation requires computing the lookahead equation:

$$U_{k+1}^{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s' \mid s, \pi(s)) U_k^{\pi}(s')$$

Updating the value at a single state requires summing over all $|\mathcal{S}|$ states. For a single iteration over all states, we must do this operation $|\mathcal{S}|$ times. Thus, the time complexity of k steps of iterative policy evaluation is $O(k|\mathcal{S}|^2)$.

Exercise 7.4. Suppose that we have an MDP with six states, $s_{1:6}$, and four actions, $a_{1:4}$. Using the following tabular form of the action value function $Q(s, a)$, compute $U(s)$, $\pi(s)$, and $A(s, a)$.

$Q(s, a)$	a_1	a_2	a_3	a_4
s_1	0.41	0.46	0.37	0.37
s_2	0.50	0.55	0.46	0.37
s_3	0.60	0.50	0.38	0.44
s_4	0.41	0.50	0.33	0.41
s_5	0.50	0.60	0.41	0.39
s_6	0.71	0.70	0.61	0.59

Solution: We can compute $U(s)$, $\pi(s)$, and $A(s, a)$ using the following equations:

$$U(s) = \max_a Q(s, a) \qquad \pi(s) = \arg\max_a Q(s, a) \qquad A(s, a) = Q(s, a) - U(s)$$

s	$U(s)$	$\pi(s)$	$A(s, a_1)$	$A(s, a_2)$	$A(s, a_3)$	$A(s, a_4)$
s_1	0.46	a_2	−0.05	0.00	−0.09	−0.09
s_2	0.55	a_2	−0.05	0.00	−0.09	−0.18
s_3	0.60	a_1	0.00	−0.10	−0.22	−0.16
s_4	0.50	a_2	−0.09	0.00	−0.17	−0.09
s_5	0.60	a_2	−0.10	0.00	−0.19	−0.21
s_6	0.71	a_1	0.00	−0.01	−0.10	−0.12

Exercise 7.5. Suppose that we have a three-tile, straight-line hex world (appendix F.1) where the rightmost tile is an absorbing state. When we take any action in the rightmost state, we get a reward of 10 and we are transported to a fourth terminal state where we no longer receive any reward. Use a discount factor of $\gamma = 0.9$, and perform a single step of policy iteration where the initial policy π has us move east in the first tile, northeast in the second tile, and southwest in the third tile. For the policy evaluation step, write out the transition matrix $\mathbf{T}^{\pi}$ and the reward vector $\mathbf{R}^{\pi}$, and then solve the infinite horizon value function $\mathbf{U}^{\pi}$ directly using matrix inversion. For the policy improvement step, compute the updated policy π' by maximizing the lookahead equation.

Solution: For the policy evaluation step, we use equation (7.10), repeated here:

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

Forming the transition matrix $\mathbf{T}^\pi$ and reward vector $\mathbf{R}^\pi$ with an additional state for the terminal state, we can solve for the infinite horizon value function $\mathbf{U}^\pi$:[26]

$$\mathbf{U}^\pi = \left(\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} - (0.9) \begin{bmatrix} 0.3 & 0.7 & 0 & 0 \\ 0 & 0.85 & 0.15 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \right)^{-1} \begin{bmatrix} -0.3 \\ -0.85 \\ 10 \\ 0 \end{bmatrix} \approx \begin{bmatrix} 1.425 \\ 2.128 \\ 10 \\ 0 \end{bmatrix}$$

[26] The hex world problem defines $R(s, a, s')$, so in order to produce entries for $\mathbf{R}^\pi$, we must compute

$$R(s,a) = \sum_{s'} T(s' \mid s, a) R(s, a, s')$$

For example, -0.3 comes from the 30 % chance that moving east causes a collision with the border, with cost -1.

For the policy improvement step, we apply equation (7.11) using the updated value function. The actions in the arg max term correspond to a_E, a_{NE}, a_{NW}, a_W, a_{SW}, and a_{SE}:

$$\begin{aligned} \pi(s_1) &= \arg\max(1.425, 0.527, 0.283, 0.283, 0.283, 0.527) = a_E \\ \pi(s_2) &= \arg\max(6.575, 2.128, 0.970, 1.172, 0.970, 2.128) = a_E \\ \pi(s_3) &= \arg\max(10, 10, 10, 10, 10, 10) \text{ (all actions are equally desirable)} \end{aligned}$$

Exercise 7.6. Perform two steps of value iteration to the problem in exercise 7.5, starting with an initial value function $U_0(s) = 0$ for all s.

Solution: We need to use the Bellman backup (equation (7.16)) to iteratively update the value function. The actions in the max term correspond to a_E, a_{NE}, a_{NW}, a_W, a_{SW}, and a_{SE}. For our first iteration, the value function is zero for all states, so we only need to consider the reward component:

$$\begin{aligned} U_1(s_1) &= \max(-0.3, -0.85, -1, -1, -1, -0.85) = -0.3 \\ U_1(s_2) &= \max(-0.3, -0.85, -0.85, -0.3, -0.85, -0.85) = -0.3 \\ U_1(s_3) &= \max(10, 10, 10, 10, 10, 10) = 10 \end{aligned}$$

For the second iteration,

$$\begin{aligned} U_2(s_1) &= \max(-0.57, -1.12, -1.27, -1.27, -1.27, -1.12) = -0.57 \\ U_2(s_2) &= \max(5.919, 0.271, -1.12, -0.57, -1.12, 0.271) = 5.919 \\ U_2(s_3) &= \max(10, 10, 10, 10, 10, 10) = 10 \end{aligned}$$

Exercise 7.7. Apply one sweep of asynchronous value iteration to the problem in exercise 7.5, starting with an initial value function $U_0(s) = 0$ for all s. Update the states from right to left.

Solution: We use the Bellman backup (equation (7.16)) to iteratively update the value function over each state following our ordering. The actions in the max term correspond to a_E, a_{NE}, a_{NW}, a_W, a_{SW}, and a_{SE}:

$$
\begin{aligned}
U(s_3) &= \max(10, 10, 10, 10, 10, 10) = 10 \\
U(s_2) &= \max(6, 0.5, -0.85, -0.3, -0.85, 0.5) = 6 \\
U(s_1) &= \max(3.48, -0.04, -1, -1, -1, -0.04) = 3.48
\end{aligned}
$$

Exercise 7.8. Prove that a Bellman residual of δ guarantees that the value function obtained by value iteration is within $\delta\gamma/(1-\gamma)$ of $U^*(s)$ at every state s.

Solution: For a given U_k, suppose we know that $\|U_k - U_{k-1}\|_\infty < \delta$. Then we bound the improvement in the next iteration:

$$
\begin{aligned}
U_{k+1}(s) - U_k(s) &= \max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U_k(s') \right) \\
&\qquad - \max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U_{k-1}(s') \right) \\
&< \max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U_k(s') \right) \\
&\qquad - \max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) \big(U_k(s') - \delta\big) \right) \\
&= \delta\gamma
\end{aligned}
$$

Similarly,

$$
\begin{aligned}
U_{k+1}(s) - U_k(s) &> \max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U_k(s') \right) \\
&\qquad - \max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) \big(U_k(s') + \delta\big) \right) \\
&= -\delta\gamma
\end{aligned}
$$

The accumulated improvement after infinite iterations is thus bounded by

$$
\|U^*(s) - U_k(s)\|_\infty < \sum_{i=1}^{\infty} \delta\gamma^i = \frac{\delta\gamma}{1-\gamma}
$$

A Bellman residual of δ thus guarantees that the optimal value function obtained by value iteration is within $\delta\gamma/(1-\gamma)$ of U^*.

Exercise 7.9. Suppose that we run policy evaluation on an expert policy to obtain a value function. If acting greedily with respect to that value function is equivalent to the expert policy, what can we deduce about the expert policy?

Solution: We know from the Bellman optimality equation that greedy lookahead on an optimal value function is stationary. If the greedy policy matches the expert policy, then both policies are optimal.

Exercise 7.10. Show how an LQR problem with a quadratic reward function $R(\mathbf{s}, \mathbf{a}) = \mathbf{s}^\top \mathbf{R}_s \mathbf{s} + \mathbf{a}^\top \mathbf{R}_a \mathbf{a}$ can be reformulated so that the reward function includes linear terms in $\mathbf{s}$ and $\mathbf{a}$.

Solution: We can introduce an additional state dimension that is always equal to 1, yielding a new system with linear dynamics:

$$\begin{bmatrix} \mathbf{s}' \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{T}_s & \mathbf{0} \\ \mathbf{0}^\top & 1 \end{bmatrix} \begin{bmatrix} \mathbf{s} \\ 1 \end{bmatrix} + \mathbf{T}_a \mathbf{a}$$

The reward function of the augmented system can now have linear state reward terms:

$$\begin{bmatrix} \mathbf{s} \\ 1 \end{bmatrix}^\top \mathbf{R}_{\text{augmented}} \begin{bmatrix} \mathbf{s} \\ 1 \end{bmatrix} = \mathbf{s}^\top \mathbf{R}_s \mathbf{s} + 2\mathbf{r}_{s,\text{linear}}^\top \mathbf{s} + r_{s,\text{scalar}}$$

Similarly, we can include an additional action dimension that is always 1 in order to obtain linear action reward terms.

Exercise 7.11. Why does the optimal policy obtained in example 7.4 produce actions with greater magnitude when the horizon is greater?

Solution: The problem in example 7.4 has quadratic reward that penalizes deviations from the origin. The longer the horizon, the greater the negative reward that can be accumulated, making it more worthwhile to reach the origin sooner.

Exercise 7.12. Prove that iterative policy evaluation converges to the solution of equation (7.6).

Solution: Consider iterative policy evaluation applied to a policy π as given in equation (7.5):

$$U_{k+1}^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s' \mid s, \pi(s)) U_k^\pi(s')$$

Let us define an operator B_π and rewrite this as $U^\pi_{k+1} = B_\pi U^\pi_k$. We can show that B_π is a contraction mapping:

$$\begin{aligned} B_\pi U^\pi(s) &= R(s, \pi(s)) + \gamma \sum_{s'} T(s' \mid s, \pi(s)) U^\pi(s') \\ &= R(s, \pi(s)) + \gamma \sum_{s'} T(s' \mid s, \pi(s)) \big(U^\pi(s') - \hat{U}^\pi(s') + \hat{U}^\pi(s')\big) \\ &= B_\pi \hat{U}^\pi(s) + \gamma \sum_{s'} T(s' \mid s, \pi(s)) \big(U^\pi(s') - \hat{U}^\pi(s')\big) \\ &\leq B_\pi \hat{U}^\pi(s) + \gamma \|U^\pi - \hat{U}^\pi\|_\infty \end{aligned}$$

Hence, $\|B_\pi U^\pi - B_\pi \hat{U}^\pi\|_\infty \leq \alpha \|U^\pi - \hat{U}^\pi\|_\infty$ for $\alpha = \gamma$, implying that B_π is a contraction mapping. As discussed in appendix A.15, $\lim_{t\to\infty} B^t_\pi U^\pi_1$ converges to a unique fixed point U^π, for which $U^\pi = B_\pi U^\pi$.

Exercise 7.13. Prove that value iteration converges to a unique solution.

Solution: The value iteration update (equation (7.16)) is

$$U^{k+1}(s) = \max_a \left(R(s, a) + \gamma \sum_{s'} T(s' \mid s, a) U_k(s') \right)$$

We will denote the Bellman operator as B and rewrite an application of the Bellman backup as $U_{k+1} = BU_k$. As with the previous problem, if B is a contraction mapping, then repeated application of B to U will converge to a unique fixed point.

We can show that B is a contraction mapping:

$$\begin{aligned} BU(s) &= \max_a \left(R(s, a) + \gamma \sum_{s'} T(s' \mid s, a) U(s') \right) \\ &= \max_a \left(R(s, a) + \gamma \sum_{s'} T(s' \mid s, a) \big(U(s') - \hat{U}(s') + \hat{U}(s')\big) \right) \\ &\leq B\hat{U}(s) + \gamma \max_a \sum_{s'} T(s' \mid s, a) \big(U(s') - \hat{U}(s')\big) \\ &\leq B\hat{U}(s) + \alpha \|U - \hat{U}\|_\infty \end{aligned}$$

for $\alpha = \gamma \max_s \max_a \sum_{s'} T(s' \mid s, a)$, with $0 \leq \alpha < 1$. Hence, $\|BU - B\hat{U}\|_\infty \leq \alpha \|U - \hat{U}\|_\infty$, which implies that B is a contraction mapping.

Exercise 7.14. Show that the point to which value iteration converges corresponds to the optimal value function.

Solution: Let U be the value function produced by value iteration. We want to show that $U = U^*$. At convergence, we have $BU = U$. Let U_0 be a value function that maps all states to 0. For any policy π, it follows from the definition of B_π that $B_\pi U_0 \leq BU_0$. Similarly, $B^t_\pi U_0 \leq B^t U_0$. Because $B^t_{\pi^*} U_0 \to U^*$ and $B^t U_0 \to U$ as $t \to \infty$, it follows that $U^* \leq U$, which can be the case only if $U = U^*$.

Exercise 7.15. Suppose that we have a linear Gaussian problem with disturbance $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \mathbf{\Sigma})$ and quadratic reward. Show that the scalar term in the utility function has the form:

$$q_{h+1} = \sum_{i=1}^{h} \mathbb{E}_{\mathbf{w}}\left[\mathbf{w}^\top \mathbf{V}_i \mathbf{w}\right] = \sum_{i=1}^{h} \mathrm{Tr}(\mathbf{\Sigma}\mathbf{V}_i)$$

You may want to use the *trace trick*:

$$\mathbf{x}^\top \mathbf{A}\mathbf{x} = \mathrm{Tr}\left(\mathbf{x}^\top \mathbf{A}\mathbf{x}\right) = \mathrm{Tr}\left(\mathbf{A}\mathbf{x}\mathbf{x}^\top\right)$$

Solution: This equation is true if $\mathbb{E}_{\mathbf{w}}\left[\mathbf{w}^\top \mathbf{V}_i \mathbf{w}\right] = \mathrm{Tr}(\mathbf{\Sigma}\mathbf{V}_i)$. Our derivation is

$$\begin{aligned}
\mathop{\mathbb{E}}_{\mathbf{w}\sim\mathcal{N}(\mathbf{0},\mathbf{\Sigma})}\left[\mathbf{w}^\top \mathbf{V}_i \mathbf{w}\right] &= \mathop{\mathbb{E}}_{\mathbf{w}\sim\mathcal{N}(\mathbf{0},\mathbf{\Sigma})}\left[\mathrm{Tr}\left(\mathbf{w}^\top \mathbf{V}_i \mathbf{w}\right)\right] \\
&= \mathop{\mathbb{E}}_{\mathbf{w}\sim\mathcal{N}(\mathbf{0},\mathbf{\Sigma})}\left[\mathrm{Tr}\left(\mathbf{V}_i \mathbf{w}\mathbf{w}^\top\right)\right] \\
&= \mathrm{Tr}\left(\mathop{\mathbb{E}}_{\mathbf{w}\sim\mathcal{N}(\mathbf{0},\mathbf{\Sigma})}\left[\mathbf{V}_i \mathbf{w}\mathbf{w}^\top\right]\right) \\
&= \mathrm{Tr}\left(\mathbf{V}_i \mathop{\mathbb{E}}_{\mathbf{w}\sim\mathcal{N}(\mathbf{0},\mathbf{\Sigma})}\left[\mathbf{w}\mathbf{w}^\top\right]\right) \\
&= \mathrm{Tr}(\mathbf{V}_i \mathbf{\Sigma}) \\
&= \mathrm{Tr}(\mathbf{\Sigma}\mathbf{V}_i)
\end{aligned}$$

Exercise 7.16. What is the role of the scalar term q in the LQR optimal value function, as given in equation (7.31)?

$$q_{h+1} = \sum_{i=1}^{h} \mathbb{E}_{\mathbf{w}}\left[\mathbf{w}^\top \mathbf{V}_i \mathbf{w}\right]$$

Solution: A matrix $\mathbf{M}$ is positive definite if, for all nonzero $\mathbf{x}$, $\mathbf{x}^\top \mathbf{M}\mathbf{x} > 0$. In equation (7.31), every $\mathbf{V}_i$ is negative semidefinite, so $\mathbf{w}^\top \mathbf{V}\mathbf{w} \leq 0$ for all $\mathbf{w}$. Thus, these q terms are guaranteed to be nonpositive. This should be expected, as it is impossible to obtain positive reward in LQR problems, and we seek instead to minimize cost.

The q scalars are offsets in the quadratic optimal value function:

$$U(\mathbf{s}) = \mathbf{s}^\top \mathbf{V}\mathbf{s} + q$$

Each q represents the baseline reward around which the $\mathbf{s}^\top \mathbf{V}\mathbf{s}$ term fluctuates. We know that $\mathbf{V}$ is negative definite, so $\mathbf{s}^\top \mathbf{V}\mathbf{s} \leq 0$, and q thus represents the expected reward that one could obtain if one were at the origin, $\mathbf{s} = \mathbf{0}$.