14 Markov Decision Process

In this chapter, we formally introduce the *Markov Decision Process (MDP)*, a way to formulate an RL environment. We then present ways to find the optimal strategy of an agent, provided that the agent knows the full details of the MDP — that is, knows everything about the environment.

14.1 Markov Decision Process (MDP)

Let's review the key ingredients of RL. We have the *agent*, who senses the environment and captures it as the current *state*. There is a finite number of actions available at any given state, and taking an action a in state s will cause a transition to s' with probability $p(s' \mid s, a)$. Each transition is accompanied by a reward $r(a \mid s, s_i) \in \mathbb{R}$. Finally, the goal of the agent is to maximize the expected reward via a sequence of actions.

A *Markov Decision Process (MDP)* is a formalization of these concepts. It is a *directed graph* which consists of four key features:

- A set S which contains all possible states
- A set A which contains all possible actions
- For each valid tuple of action a and states s_1, s_2, there is an assigned probability $p(s_2 \mid s_1, a)$ probability of transition to s_2 if action a is taken in s_1
- For each valid tuple of action a and states s_1, s_2, there is an assigned reward $r(a \mid s_1, s_2)$, which is obtained if action a is taken to transition from s_1 to s_2

If a designed MDP has M actions and N states, we can specify the MDP by a table of transition probabilities (with MN^2 numbers) and a table for rewards (with MN^2 numbers).

14.1.1 *Revisiting the Cake Eating Example*

Let's return to the case study on eating cake from Subsection 13.3, and formally express it through a MDP. The set of states is given as $S = \{0, 1, 2, 3\}$, where each state represents the number of slices left. The set of actions is given as $A = \{1, 2, 3\}$, where each action represents the number of slices you choose to eat on a given night. Notice that reward only depends on how many slices you take, not how many slices are left after your roommate goes through the fridge. That is, we can define the reward $r(a \mid s, *)$ for each $a \in A$ to be the same for every $s \in S$ where a is feasible. [1]

[1] We still need to include the previous state s because not all actions are feasible at each state. For example, you can't eat 2 slices when there is only 1 slice left.

Example 14.1.1. *Let's revisit Example 13.3.2 as a motivating example. If we let $a = 2$, $s_1 = 3$, and $s_2 = 0$, then the probability of the specified transition is $p(s_2 \mid s_1, a) = 0.5$. The associated reward is $r(a \mid s_1, s_2) = 1.5$ as discussed earlier.*

We are now ready to generalize to the full MDP, which is shown in Figure 14.1. Note that every transition is labeled with its probability, associated action, and associated reward.

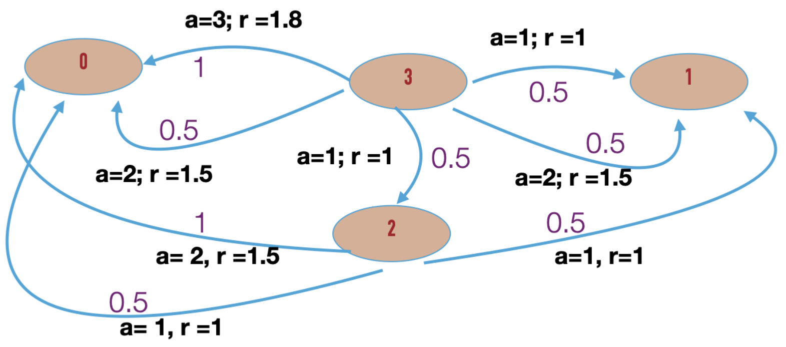


Figure 14.1: The full diagram of the cake problem when described as a MDP.

14.1.2 *Discounting the Future*

The MDP describing cake eating in the previous subsection was acyclic. [2] However, in general, MDPs can have directed cycles, and the agent's actions can allow it to continuously collect rewards along that cycle. For instance, continuing our cake theme, we may have a scenario in which you receive a fresh cake every 3 days. But now we run into a problem: how can we calculate the expected reward when there is an unbounded number of steps?

[2] This is also known as an *Episodic MDP*.

The solution lies in the concept of *future discounting*. The basic idea is to reduce, or *discount*, the amount of reward we get from future steps. In an MDP, we represent this through a discount factor $0 < \gamma \leq 1$ and an associated infinite sum. [3]

[3] This is related to notions of discounting commonly considered in economics.

Definition 14.1.2 (Future Discounting). *If a reward r_t is received at time $t = 0, 1, 2, \ldots$, then the perceived value of these rewards r_d, or the **discounted reward**, at $t = 0$ is:*

$$r_d = r_0 + \gamma r_1 + \gamma^2 r_2 + \gamma^3 r_3 + \cdots$$

Example 14.1.3. *Consider the cake eating problem again and let r_t denote the reward we get on night t. If the reward is discounted by a factor of γ every night, the total expected discounted reward $E[total]$ can be rewritten as*

$$\mathbb{E}[total] = \mathbb{E}[r_1] + \gamma \cdot \mathbb{E}[r_2] + \gamma^2 \cdot \mathbb{E}[r_3]$$

Consider taking the action $a = 2$ on the first night. If $\gamma = 0.9$, then the expected discounted reward is

$$1.5 + 0.9 \cdot (0.5 \cdot 1 + 0.5 \cdot 0) = 1.95$$

This is the same as in Example 13.3.2 except the reward taken from the second night is discounted by a factor of 0.9. Now consider taking the action $a = 1$ on the first night and on the second night. If $\gamma = 0.9$, the expected discounted reward is

$$1 + 0.9 \cdot (0.5 \cdot 1 + 0.5 \cdot 1) + 0.9^2 \cdot (0.5^2 \cdot 1) = 2.1025$$

Here, we first take the reward of 1 on the first night without any discount factor. Then, we calculate the expected reward from the second night — 1 whether or not the roommate eats a slice — and discount it by a factor of 0.9. Finally, we calculate the expected reward from the third night — 1 only if the roommate did not eat any slice on the first two nights — and discount it by a factor of 0.9^2.

Note that in Definition 14.1.2, if each $r_t \in [-R, R]$ and if $\gamma < 1$, then the discounted reward of the infinite sequence has the following upper bound:

$$|r_d| \leq R(1 + \gamma + \gamma^2 + \cdots) = \frac{R}{1-\gamma} \tag{14.1}$$

(14.1) is derived by considering the formula for the sum of an infinite geometric series, which we can invoke if $\gamma < 1$. In general, γ is up to the system designer. A lower γ would imply that the agent places little importance on future rewards, whereas $\gamma = 1$ would imply that there is effectively no discounting.

14.2 Policy and Markov Reward Process

Now that we have discussed what an action is and what it does in an MDP, we want to specify what action an agent has to take in each state. This is known as a *policy*.

Example 14.2.1. *Consider again the cake eating MDP example without a discount factor. We already established through Example 13.3.2 and Example 13.3.5 that to maximize the expected reward, you need to eat one slice per day until all slices are gone. That is, in any state j where $j = 1, 2, 3$, you need to take action 1.*

In general, if S is the set of states, and A is the set of actions, then a *policy* (not necessarily the optimum) π can be defined as a function $\pi : S \to A$

Definition 14.2.2 (Policy)**.** *If S is the set of states, and A is the set of actions, any function $\pi : S \to A$ is called a **policy** that describes which action to take at each state. In particular, each state s should only be mapped to a valid action $a \in A_s$ at that state.*

Recall that if there are M actions and N states, there are at most MN^2 transitions in the graph of the MDP. Because a policy specifies one action per state, there are at most N^2 transitions that remain when we choose a specific policy. Therefore, it can be understood that a policy trims out the MDP.

14.2.1 Markov Reward Process (MRP)

When we have an MDP and a fixed policy, we have what is called a *Markov Reward Process (MRP)*. There are no more decisions to make; instead, all we need to do is take the action specified by the policy; probabilistically follow a transition into a new state; and collect the associated reward.

Example 14.2.3. *Let's revisit Figure 14.1. If we fix the policy to be $\pi(s) = 1$ for any $s \in S$, we can focus our attention to the action $a = 1$. Then there are three trajectories that will lead from state 3 to state 0, based on what the roommate does overnight. The first trajectory is $3 \to 1 \to 0$ with probability 0.5×1 and reward $1 + 1$. The second trajectory is $3 \to 2 \to 0$ with probability 0.5×0.5 and reward $1 + 1$. The last trajectory is $3 \to 2 \to 1 \to 0$ with probability $0.5 \times 0.5 \times 1$ and reward $1 + 1 + 1$.*

In general, when we fix a policy π and an initial state s, we can redraw the transition diagram of an MDP into a tree diagram for the MRP, where each node corresponds to a state, and each edge corresponds to a probabilistic transition. The top node represents the initial state, and each subsequent row of the tree represents the set of possible states after taking an action from their parent node.

Example 14.2.4. *We revisit Example 14.2.3. We now transform Figure 14.1 into a tree diagram for the MRP as shown in Figure 14.2. The top node is the initial state 3. The second row of the tree is all states that can be achieved by taking the action 1 at state 3, and so on.*

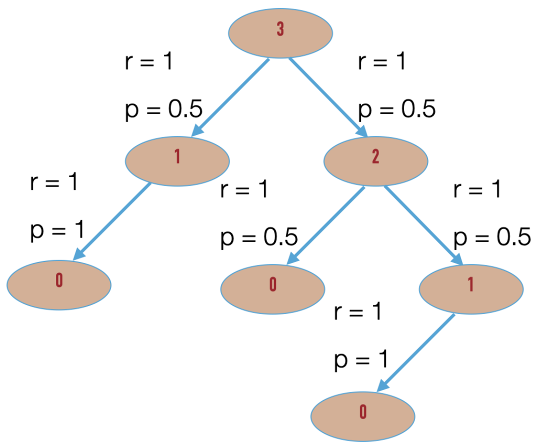


Figure 14.2: A tree representing the MRP in Example 14.2.3.

Note that in an MRP tree, the same state can appear multiple times, but each copy of the same state is identical — that is, the subtree rooted at each copy must be identical. In Figure 14.2, the state 1 appears three time in the tree. Every time it appears, it can only lead to state 0 with probability 1. This is simply the result of fixing a policy π — once we know the state we are in, we only have one choice for the action to take.

The policy also induces a *value function* on this tree. The value function assigns a value to each node of the tree, and each value intuitively measures how much reward the agent should expect to collect once the agent knows they have arrived at that node. By the observation from the previous paragraph, this expected reward should be the same for two nodes, if they are the copy of the same state. Therefore, we can equivalently define the value function for each state s instead. Formally, we define the value function as the following.

Definition 14.2.5 (Value Function). *$v_\pi(s)$, the **value of state** s **under the policy** π, is the expected discounted reward of a random trajectory starting from s. We can define this value by using the following recursive formula:*

$$v_\pi(s) = \sum_{s'} p(s' \mid s, \pi(s)) \cdot (r(\pi(s) \mid s, s') + \gamma v_\pi(s')) \tag{14.2}$$

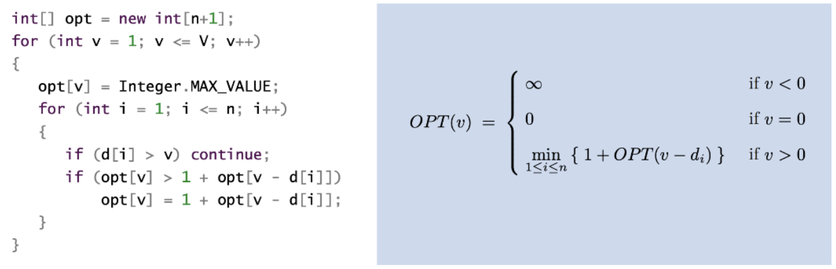
*Computing the value function as in (14.2) is also known as the **Bellman equation**.*

Let us unpack the intuition behind (14.2). Once we take action $\pi(s)$ at state s, it will bring us to state s' with probability $p(s' \mid s, a)$, immediately giving us a reward $r(a \mid s, s')$. Then, the expected reward from that point on is already captured by the value $v_\pi(s')$. We just need to apply the discount factor γ because we already took one time step to reach s' from s.

On the other hand, if we pick any random trajectory starting from s, its next node will be some state s' that is reachable from s. Therefore, the contribution of this particular trajectory to $v_\pi(s)$ is accounted for when we sum over that particular s'.

14.2.2 *Connection with Dynamic Programming*

In COS 226, you may have seen an implementation of a bottom-up dynamic programming.



```
int[] opt = new int[n+1];
for (int v = 1; v <= V; v++)
{
   opt[v] = Integer.MAX_VALUE;
   for (int i = 1; i <= n; i++)
   {
      if (d[i] > v) continue;
      if (opt[v] > 1 + opt[v - d[i]])
          opt[v] = 1 + opt[v - d[i]];
   }
}
```

$$OPT(v) = \begin{cases} \infty & \text{if } v < 0 \\ 0 & \text{if } v = 0 \\ \min\limits_{1 \le i \le n} \{\, 1 + OPT(v - d_i) \,\} & \text{if } v > 0 \end{cases}$$

Figure 14.3: A Dynamic Programming implementation of a coin changing problem that uses the bottom-up approach.

In such implementations, the algorithm divides the problem into subproblems arranged as directed acyclic graphs and compute "bottom-up." The MDP from the cake eating problem is acylic and our method using a look-ahead tree is similar to the dynamic programming algorithms. Therefore, it seems like we can apply a similar algorithm to the cake eating problem.

Example 14.2.6. *Consider Example 14.2.3 again, but now with a discount factor of* 0.9*. We will find the value $v_\pi(s)$ of each state s by going bottom-up from the tree in Figure 14.2. We start by noticing that $v_\pi(0) = 0$ as can be seen from the bottom row. Then from the third node of the third row, we can calculate*

$$v_\pi(1) = 1 \cdot (1 + 0.9 \cdot 0) = 1$$

From the second node of the second row, we can calculate

$$v_\pi(2) = 0.5 \cdot (1 + 0.9 \cdot 0) + 0.5 \cdot (1 + 0.9 \cdot 1) = 1.45$$

Finally, from the top node, we can calculate

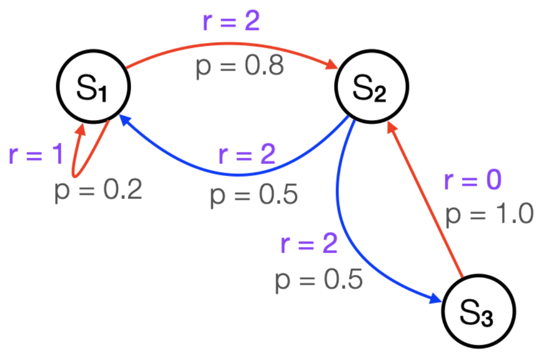
$$v_\pi(3) = 0.5 \cdot (1 + 0.9 \cdot 1) + 0.5 \cdot (1 + 0.9 \cdot 1.45) = 2.1025$$

But in general, the dynamic programming approach does not completely apply to MDP. The biggest assumption for dynamic programming algorithms is that the graph is *acyclic*, but MDPs are generally allowed to have directed cycles if we can return to the same state after a sequence of actions. Therefore, computing the expected reward for even a single policy π involves solving a system of linear equations.

Example 14.2.7. *Assume that we have three states s_1, s_2, s_3 and transitions as in Figure 14.2.2 with a discount factor of $\gamma = 0.7$. Then the value at each state is given as*

$$
\begin{aligned}
v_\pi(s_1) &= 0.2 \times (1 + 0.7 v_\pi(s_1)) + 0.8 \times (2 + 0.7 v_\pi(s_2)) \\
v_\pi(s_2) &= 0.5 \times (2 + 0.7 v_\pi(s_1)) + 0.5 \times (2 + 0.7 v_\pi(s_3)) \\
v_\pi(s_3) &= 1 \times (0 + 0.7 v_\pi(s_2))
\end{aligned}
$$

Unlike in Example 14.2.6, we cannot compute any of these values one by one because the values are interdependent in a cyclic manner. Instead, we need to solve the linear equation as a whole, which gives us the solution: $v_\pi(s_1) \approx 5.47, v_\pi(s_2) \approx 5.18, v_\pi(s_3) \approx 3.63$.



14.3 *Optimal Policy*

Out of all choices for a policy, we are interested in the *optimal policy*, the one that maximizes the expected (discounted) reward. Surprisingly, it is known that there always exists a policy π^* that obtains the maximum expected reward from all initial states *simultaneously*; that is $\pi^* = \arg\max_\pi v_\pi(s)$ for every state s. [4]. The value function of the optimal policy is called the *optimal value function* and is often denoted as $v^*(s)$. Then we can express the optimal value function using (14.2) as:

$$v^*(s) = \max_\pi \sum_{s'} p(s' \mid s, \pi(s))(r(\pi(s) \mid s, s') + \gamma v_\pi(s'))$$

[4] If there are multiple such policies, we denote any one of them by π^*.

This is just restating the fact that the optimal value of state s is the maximum of all possible values $v_\pi(s)$ of s under a policy π — *i. e.*, the Bellman equation evaluated with the values $v_\pi(s')$ of each children node s' under that specific policy π.

But we can even go further than this result. It is known that the optimal value also satisfies the following:

$$v^*(s) = \max_\pi \sum_{s'} p(s' \mid s, \pi(s))(r(\pi(s) \mid s, s') + \gamma v^*(s')) \tag{14.3}$$

Notice that $v_\pi(s')$ in the summation has now been replaced with $v^*(s')$. This property, known as the *Bellman Optimality condition*, states that the optimal value is even the maximum when the Bellman equation is evaluated with the values $v^*(s')$, regardless of the choice of the policy π.

Notice that the right-hand side of (14.3) only depends on the choice of the action a of the given state s, not any other states. Therefore, we can rewrite (14.3) as:

$$v^*(s) = \max_{a \in A_s} \sum_{s'} p(s' \mid s, a)(r(a \mid s, s') + \gamma v^*(s')) \tag{14.4}$$

which also suggests that the optimal action at state s can be expressed as:

$$\pi^*(s) = \arg\max_{a \in A_s} \sum_{s'} p(s' \mid s, a)(r(a \mid s, s') + \gamma v^*(s')) \tag{14.5}$$

But the problem is: it is unclear how to turn this into an efficient algorithm. Computing the value $v^*(s)$ depends on the value $v^*(s')$, which can also depend on $v^*(s)$, which becomes recursive.

In this section, we present an iterative algorithm called the *value iteration* method which will be used to compute the optimal policy. Before we describe the algorithm, we unpack the underlying ideas.

14.3.1 Developing Intuition about Optimality: Gridworld

To develop intuition about how to find an optimum policy, let's consider a classic example called Gridworld. [5]

[5] Source: Sutton and Barton 2020, `https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf`

Example 14.3.1 (Gridworld)**.** *Consider a 5×5 grid. The set of states is given as the cells of this grid. At each state except for at $A = (1, 2)$ and $B = (1, 4)$, there are four available actions: move left/right/up/down, each with reward 0, except in the following setting: if the action will make you move off the grid. then the reward is -1, and you are made to stay at the same state instead.*

At A, there is only one action: move to $A' = (5, 2)$ with reward 10 and similarly at B, there is one action: move to $B' = (3, 4)$ with reward 5. [6] The discount factor is given as 0.9.

[6] The outgoing transition from A and B can be thought of as "wormholes."

How can we compute the reward for a policy in the example above? When beginners try to calculate the exact value using the above definitions, they quickly get bogged down in keeping track of too many variables, equations, and recurrences.

Instead, let's try to think intuitively about what an optimal policy **should** be trying to do. Since the wormholes are the only source of rewards, an optimal policy should be trying to utilize the wormholes as much as possible. Using this kind of intuition, we can design a

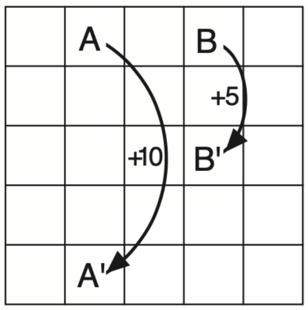


Figure 14.4: Visual representation of the Gridworld.

policy that looks at least near-optimal, and use its value as a **lower bound** for the optimal policy.

First, let $v^*(s)$ denote the value $v_{\pi^*}(s)$ of state s for an optimal policy π^*. Since there is only one action to choose from at state A, we know that

$$v^*(A) = 10 + \gamma v^*(A') \tag{14.6}$$

Now, at the state A', one possible trajectory you can follow is "go up four steps" (each with reward 0) back to A. We know that the optimal value has to be at least as great as this value. That is

$$v^*(A') \geq \gamma^4 v^*(A) \tag{14.7}$$

Combining (14.6) and (14.7), we get

$$v^*(A) \geq 10 + \gamma^5 v^*(A)$$

If we solve for $v^*(A)$, we get

$$v^*(A) \geq \frac{10}{1-\gamma^5} \approx 24.4$$

The value iteration method discussed below is based on this intuition — we can provide a lower bound for the optimal policy by suggesting some potential policy. If we repeat this process, the lower bound for the optimal policy can only go up. At the end of the section, we will prove that this process converges to the actual optimal value.

14.3.2 Value Iteration Method

Value Iteration is a method guaranteed to find the optimal policy. At each step of the iteration, we are given a lower bound on the optimal values of each state s. Using the values of the immediate children nodes in the tree, we can compute an improved lower bound on $v^*(s)$.

Example 14.3.2. *See Figure 14.5. Suppose there are two actions to take at state s. The first action, labeled as blue, will lead to state s_1 with reward -1 with probability 0.5 and s_3 with reward -1 with probability 0.5. The second action, labeled as red, will lead to state s_2 with reward 2 with probability 1. The discount factor is given as 0.6. Now assume that someone tells us that they know a way to get expected reward of 12 starting from s_1, 1 from s_2, and 4 from s_3, regardless of the choice of initial action at s. In other words, the optimal values for these three states are lower bounded by: $v^*(s_1) \geq 12, v^*(s_2) \geq 1$ and $v^*(s_3) \geq 4$. Using this fact, we consider two strategies* [7] *— (1) first take action blue at state s and play optimally thereon based on the other person's knowledge; (2) first take action red at state s and play optimally thereon. The lower bound for the expected reward for each of the two strategies can be computed as:*

[7] This is not necessarily a *policy* because the second part of playing optimally may require you to return to state s and take an action that is inconsistent with your initial choice of action.

$$\begin{aligned} v_{blue}(s) &\geq 0.5 \times (-1 + 0.6 \times 12) + 0.5 \times (-1 + 0.6 \times 4) = 3.8 \\ v_{red}(s) &\geq 1.0 \times (2 + 0.6 \times 1) = 2.6 \end{aligned}$$

The Bellman Optimality condition in (14.4) guarantees that the optimal policy is at least as good as either of these strategies. Therefore $v^*(s)$ has to be larger than both v_{blue}, v_{red}; that is, $v^*(s) \geq 3.8$.

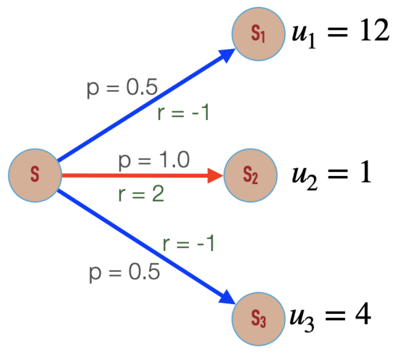


Figure 14.5: There are two actions you can take at state s, and you will end up in one of the three states: s_1, s_2, s_3.

In general, the value iteration algorithm looks like:

1. Initialize some values $v_0(s)$ for each state s such that we are guaranteed $v_0(s) \leq v^*(s)$
2. For each time step $k = 1, 2, \ldots$, and for each state s, use the values $v_k(s')$ of the immediate children s' to compute an updated value $v_{k+1}(s)$ such that $v_{k+1}(s) \leq v^*(s)$. [8]
3. When $k \to \infty$, each $v_k(s)$ will converge to the optimal value $v^*(s)$.

[8] These values $v_k(s)$ maintained by the algorithm is *not* necessarily associated with a specific policy. They are just a lower bound for the optimal value $v^*(s)$ that will be improved over time.

Recall from (14.1) that if all transition rewards are within $[-R, R]$, then the expected rewards at any state for any policy lies in $\left[-\frac{R}{1-\gamma}, \frac{R}{1-\gamma}\right]$.

Therefore, we can set the initial value $v_0(s) = -\frac{R}{1-\gamma}$ to be the lower bound for each state s. [9]

After the k-th iteration of the algorithm, we will maintain a value $v_k(s)$ for state s, where the condition $v_k(s) \leq v^*(s)$ is maintained as an invariant. Now at the $(k+1)$-th iteration, the algorithm will update the values at each state s as the following:

$$v_{k+1}(s) = \max_{\pi} \sum_{s'} p(s' \mid s, \pi(s)) \cdot \big(r(\pi(s) \mid s, s') + \gamma v_k(s')\big) \tag{14.8}$$

This is just the Bellman equation evaluated with the values $v_k(s')$ of each children node.

[9] Our proof assumes this special initialization where all $v_0(s) = -\frac{R}{1-\gamma}$ for all states s. It turns out the value iteration method converges to the optimal value for arbitrary initialization, but the proof is more complicated.

Example 14.3.3 (Example 14.3.1 revisited)**.** *Say we start the value iteration on the gridworld with all values equal to zero. Now let us compute $v_1(A)$, the value of A after the first iteration. Recall that A has only one action to choose from: moving to A'. Denote this action by a. Therefore,*

$$\begin{aligned} v_1(A) &= p(A' \mid A, a) \cdot \big(r(a \mid A, A') + \gamma v_0(A')\big) \\ &= 1.0 \cdot (10 + 0.9 \cdot 0) = 10 \end{aligned}$$

Problem 14.3.4 (Example 14.3.1 revisited)**.** *Start value iteration with all values equal to zero. What is $v_2((1,3))$, the value of $(1,3)$ after second iteration?*

14.3.3 *Why Does Value Iteration Find an Optimum Policy?*

We prove that the values $v_k(s)$ maintained by the value iteration method converge to the optimal values $v_\pi(s)$ in a finite number of steps. We break this proof down in two parts. We first prove that the invariant $v_k(s) \leq v^*(s)$ holds throughout the algorithm. Then we prove that in general, $v_{k+1}(s)$ is a tighter lower bound for $v^*(s)$ than $v_k(s)$.

Proposition 14.3.5. *For each time step $k = 1, 2, \ldots$, and for each state s, the invariant $v_k(s) \leq v^*(s)$ holds.*

Proof. Proof by mathematical induction. As discussed earlier, our choice of initial values $v_0(s) = -\frac{R}{1-\gamma}$ satisfies the invariant. Now assume that the invariant holds for some k. Now consider the update rule of the value iteration algorithm:

$$v_{k+1}(s) = \max_{\pi} \sum_{s'} p(s' \mid s, \pi(s)) \cdot \big(r(\pi(s) \mid s, s') + \gamma v_k(s')\big)$$

Notice that for any specific policy π and for any next state s', we have

$$\begin{aligned} &p(s' \mid s, \pi(s)) \cdot \big(r(\pi(s) \mid s, s') + \gamma v_k(s')\big) \\ \leq &p(s' \mid s, \pi(s)) \cdot \big(r(\pi(s) \mid s, s') + \gamma v^*(s')\big) \end{aligned}$$

because of the inductive hypothesis that $v_k(s') \le v^*(s')$. Therefore, if we sum over all state s', we have

$$\begin{aligned}&\sum_{s'} p(s' \mid s, \pi(s)) \cdot \big(r(\pi(s) \mid s, s') + \gamma v_k(s')\big)\\ \le&\sum_{s'} p(s' \mid s, \pi(s)) \cdot \big(r(\pi(s) \mid s, s') + \gamma v^*(s')\big)\end{aligned}$$

Since this inequality holds for every policy π, we have the following:

$$\begin{aligned}v_{k+1}(s) &= \max_{\pi}\sum_{s'} p(s' \mid s, \pi(s)) \cdot \big(r(\pi(s) \mid s, s') + \gamma v_k(s')\big)\\ &\le \max_{\pi}\sum_{s'} p(s' \mid s, \pi(s)) \cdot \big(r(\pi(s) \mid s, s') + \gamma v^*(s')\big) \quad = v^*(s)\end{aligned}$$

where we apply the Bellman Optimality condition (14.4) in the last equality. This concludes the inductive step, and it suffices for the proof. □

Now to prove that these values $v_k(s)$ eventually converge to $v^*(s)$, we introduce the following definition:

Definition 14.3.6. *The* ***residual*** *at s at the k-th iteration is defined as* $\delta_{s,k} = v^*(s) - v_k(s) > 0$.

Notice that as long as the residuals at the k-th iteration converge to 0, the values $v_k(s)$ also converge to $v^*(s)$. Since the residuals take finite values when the algorithm is initiated, it suffices to prove that the residuals decrease non-trivially in every iteration. [10]

[10] Our exposition of Value Iteration with our particular initialization is new. The usual textbook description requires a slightly more complicated argument.

Proposition 14.3.7. *If the largest residual at iteration k is denoted as* $\delta_k = \max_s \delta_{s,k}$*, then the largest residual δ_{k+1} at iteration $k+1$ satisfies* $\delta_{k+1} \le \gamma\delta_k$

Proof. Let a^* be the action at s under the optimum policy π^*. Then by (14.2),

$$v^*(s) = \sum_{s'} p(s' \mid s, a^*)(r(a^* \mid s, s') + \gamma v^*(s')) \tag{14.9}$$

Note that taking the action a^* is always an option at the $(k+1)$-th iteration, so the $v_{k+1}(s)$, the maximum value across all actions has to be greater than or equal to the value computed with the action a^*; that is,

$$\begin{aligned}v_{k+1}(s) &= \max_{\pi}\sum_{s'} p(s' \mid s, \pi(s))(r(\pi(s) \mid s, s') + \gamma v_k(s'))\\ &\ge \sum_{s'} p(s' \mid s, a^*)(r(a^* \mid s, s') + \gamma v_k(s')) \qquad (14.10)\end{aligned}$$

Subtracting (14.10) from (14.9), we get

$$v^*(s) - v_{k+1}(s) \le \gamma \left(\sum_{s'} p(s' \mid s, a^*)(v^*(s') - v_k(s')) \right)$$

By the definition of δ_k, each of $v^*(s') - v_k(s') = \delta_{s',k} \le \delta_k$. Therefore,

$$v^*(s) - v_{k+1}(s) \le \gamma \delta_k \sum_{s'} p(s' \mid s, a^*)$$

Finally note that $\sum_{s'} p(s' \mid s, a^*) = 1$ because p is a probability distribution. □

Theorem 14.3.8. *For each $s \in S$, $v_k(s)$ converges to $v^*(s)$ when $k \to \infty$.*

Proof. By Proposition 14.3.5 and Proposition 14.3.7,

$$|v^*(s) - v_k(s)| = v^*(s) - v_k(s) \le \delta_k \le \gamma^k \delta_0$$

which converges to 0 when k goes to infinity. □

14.3.4 Retrieving Optimal Policy from the v^*'s

One important thing to note is that the value iteration method finds the optimal *value* of each state, not the optimal *policy*. So we need an extra step to retrieve the optimal policy from the output of the value iteration algorithm. This can be done by considering the Bellman Optimality condition. For each state s, define $\pi^*(s) = a^*$ such that

$$a^* = \arg\max_{a \in A_s} \sum_{s'} p(s' \mid s, a)(r(a \mid s, s') + \gamma v^*(s')) \qquad \text{(14.5 revisited)}$$

where $v^*(s)$ is the value that the value iteration algorithm converges to. If there are multiple actions a that satisfy the equation above, arbitrarily choose an action.

Example 14.3.9 (Example 14.3.1 revisited)**.** *Say we ran the value iteration algorithm on the Gridworld. The output of the algorithm (the optimal values of each state) is given in Table 14.1.*

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

Table 14.1: Optimal values $v^*(s)$ of the Gridworld.

Consider the state $A' = (5, 2)$. There are four actions to take: left-/right/up/down. Each action would yield the following values when evaluat-

ing the Bellman equation:

$$\begin{aligned}
v_{left}(A') &= 0 + 0.9 \times 14.4 = 13.0 \\
v_{right}(A') &= 0 + 0.9 \times 14.4 = 13.0 \\
v_{up}(A') &= 0 + 0.9 \times 17.8 = 16.0 \\
v_{down}(A') &= -1 + 0.9 \times 16.0 = 13.4
\end{aligned}$$

The only action that maximizes the value is the action "go up." Therefore, we can conclude that the optimal policy π^* will adopt the action "go up" for the state A'.

Problem 14.3.10 (Example 14.3.1 revisited)**.** *Verify that an optimal policy can assign either the action "go up" or the action "go left" for the state* $(5, 3)$.