In our k-armed bandit problem, each of the k actions has an expected or mean reward given that that action is selected; let us call this the value of that action. We denote the action selected on time step t as A_t, and the corresponding reward as R_t. The value then of an arbitrary action a, denoted $q_*(a)$, is the expected reward given that a is selected:

$$q_*(a) \doteq \mathbb{E}[R_t \mid A_t = a].$$

[...] Whereas in bandit problems we estimated the value $q_*(a)$ of each action a, in MDPs we estimate the value $q_*(s, a)$ of each action a in each state s, or we estimate the value $v_*(s)$ of each state given optimal action selections. These state-dependent quantities are essential to accurately assigning credit for long-term consequences to individual action selections.
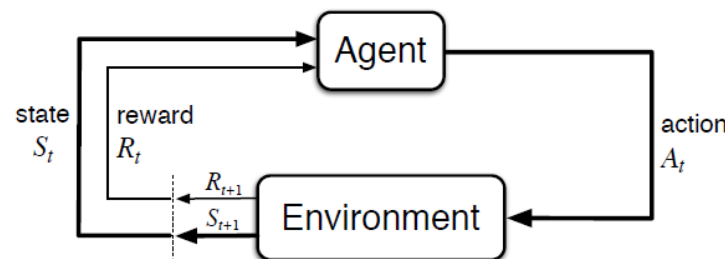

Figure 3.1: The agent–environment interaction in a Markov decision process.

In a *finite* MDP, the sets of states, actions, and rewards ($\mathcal{S}$, $\mathcal{A}$, and $\mathcal{R}$) all have a finite number of elements. In this case, the random variables R_t and S_t have well defined discrete probability distributions dependent only on the preceding state and action. That is, for particular values of these random variables, $s' \in \mathcal{S}$ and $r \in \mathcal{R}$, there is a probability of those values occurring at time t, given particular values of the preceding state and action:

$$p(s', r \mid s, a) \doteq \Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\}, \tag{3.2}$$

for all $s', s \in \mathcal{S}$, $r \in \mathcal{R}$, and $a \in \mathcal{A}(s)$. The function p defines the *dynamics* of the MDP.

The additional concept that we need is that of *discounting*. According to this approach, the agent tries to select actions so that the sum of the discounted rewards it receives over the future is maximized. In particular, it chooses A_t to maximize the expected *discounted return*:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{3.8}$$

where γ is a parameter, $0 \le \gamma \le 1$, called the *discount rate*.

$$
\begin{aligned}
G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \\
&= R_{t+1} + \gamma \left(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots \right) \\
&= R_{t+1} + \gamma G_{t+1}
\end{aligned}
\tag{3.9}
$$

The *value function* of a state s under a policy π, denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter. For MDPs, we can define v_π formally by

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s \right], \quad \text{for all } s \in \mathcal{S}, \tag{3.12}$$

We call the function v_π the *state-value function for policy π*.

Similarly, we define the value of taking action a in state s under a policy π, denoted $q_\pi(s, a)$, as the expected return starting from s, taking the action a, and thereafter following policy π:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a \right]. \tag{3.13}$$

We call q_π the *action-value function for policy π*.

A fundamental property of value functions used throughout reinforcement learning and dynamic programming is that they satisfy recursive relationships similar to that which we have already established for the return (3.9). For any policy π and any state s, the following consistency condition holds between the value of s and the value of its possible successor states:

Bellman optimality equation

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t=s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t=s] && \text{(by (3.9))} \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s',r|s,a)\Big[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1}=s']\Big] \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big], && \text{for all } s \in \mathcal{S}, \qquad (3.14)
\end{aligned}
$$

for all $s \in \mathcal{S}$. There is always at least one policy that is better than or equal to all other policies. This is an *optimal policy*. Although there may be more than one, we denote all the optimal policies by π_*. They share the same state-value function, called the *optimal state-value function*, denoted v_*, and defined as

$$
v_*(s) \doteq \max_\pi v_\pi(s), \qquad (3.15)
$$

Optimal policies also share the same *optimal action-value function*, denoted q_*, and defined as

$$
q_*(s,a) \doteq \max_\pi q_\pi(s,a), \qquad (3.16)
$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$. For the state–action pair (s,a), this function gives the expected return for taking action a in state s and thereafter following an optimal policy. Thus, we can write q_* in terms of v_* as follows:

$$
q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t=s, A_t=a]. \qquad (3.17)
$$

3

Because v_* is the value function for a policy, it must satisfy the self-consistency condition given by the Bellman equation for state values (3.14). Because it is the optimal value function, however, v_*'s consistency condition can be written in a special form without reference to any specific policy. This is the Bellman equation for v_*, or the *Bellman optimality equation.* Intuitively, the Bellman optimality equation expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state:

$$v_*(s) = \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s, a)$$

$$= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a]$$

$$= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \qquad \text{(by (3.9))}$$

$$= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \qquad (3.18)$$

$$= \max_a \sum_{s', r} p(s', r \mid s, a)[r + \gamma v_*(s')]. \qquad (3.19)$$

The last two equations are two forms of the Bellman optimality equation for v_*. The Bellman optimality equation for q_* is

$$q_*(s, a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a\right]$$

$$= \sum_{s', r} p(s', r \mid s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right]. \qquad (3.20)$$
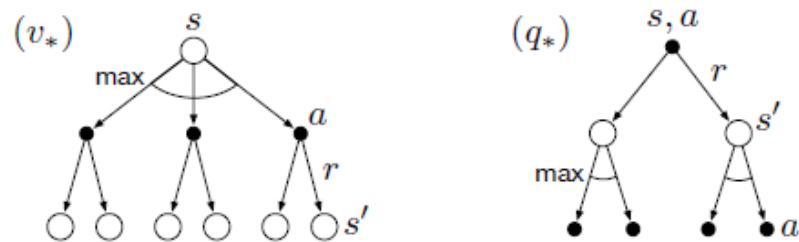

Figure 3.4: Backup diagrams for v_* and q_*

[...] DP algorithms are obtained by turning Bellman equations such as these into assignments, that is, into update rules for improving approximations of the desired value functions.

The key idea of DP, and of reinforcement learning generally, is the use of value functions to organize and structure the search for good policies. In this chapter we show how DP can be used to compute the value functions defined in Chapter 3. As discussed there, we can easily obtain optimal policies once we have found the optimal value functions, v_* or q_*, which satisfy the Bellman optimality equations *optimal state-value function*

$$v_*(s) = \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a]$$

$$= \max_a \sum_{s',r} p(s',r \mid s,a)\Big[r + \gamma v_*(s')\Big], \tag{4.1}$$

or *optimal action-value function*

$$q_*(s,a) = \mathbb{E}\Big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \;\Big|\; S_t = s, A_t = a\Big]$$

$$= \sum_{s',r} p(s',r \mid s,a)\Big[r + \gamma \max_{a'} q_*(s', a')\Big], \tag{4.2}$$

4.1 Policy Evaluation (Prediction)

First we consider how to compute the state-value function v_π for an arbitrary policy π. This is called *policy evaluation* in the DP literature. We also refer to it as the *prediction problem*. Recall from Chapter 3 that, for all $s \in \mathcal{S}$,

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s]$$

$$= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \tag{from (3.9)}$$

$$= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \tag{4.3}$$

$$= \sum_a \pi(a \mid s) \sum_{s',r} p(s',r \mid s,a)\Big[r + \gamma v_\pi(s')\Big], \tag{4.4}$$

[...] Consider a sequence of approximate value functions $v_0, v_1, v_2, \ldots$, each mapping S^+ to $\mathbb{R}$ (the real numbers). The initial approximation, v_0, is chosen arbitrarily (except that the terminal state, if any, must be given value 0), and each successive approximation is obtained by using the Bellman equation for v_π (4.4) as an update rule:

$$
\begin{aligned}
v_{k+1}(s) &\doteq \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_k(s')\right],
\end{aligned}
\tag{4.5}
$$

for all $s \in S$. Clearly, $v_k = v_\pi$ is a fixed point for this update rule because the Bellman equation for v_π assures us of equality in this case. Indeed, the sequence $\{v_k\}$ can be shown in general to converge to v_π as $k \to \infty$ under the same conditions that guarantee the existence of v_π. This algorithm is called *iterative policy evaluation*.

A complete in-place version of iterative policy evaluation is shown in pseudocode in the box below. Note how it handles termination. Formally, iterative policy evaluation converges only in the limit, but in practice it must be halted short of this. The pseudocode tests the quantity $\max_{s \in S} |v_{k+1}(s) - v_k(s)|$ after each sweep and stops when it is sufficiently small.

Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input π, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in S^+$, arbitrarily except that $V(terminal) = 0$

Loop:
$\quad \Delta \leftarrow 0$
$\quad$ Loop for each $s \in S$:
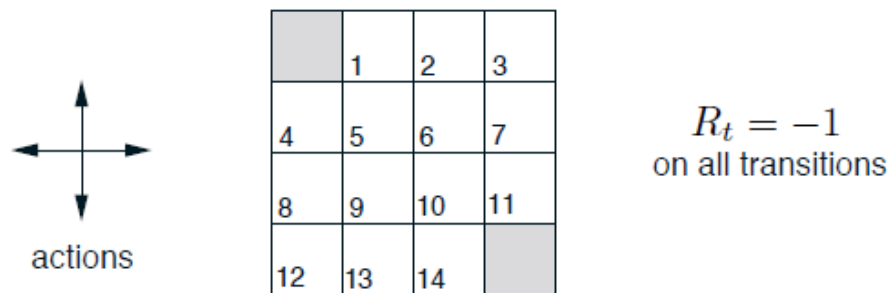$\quad\quad v \leftarrow V(s)$
$\quad\quad V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma V(s')\right]$
$\quad\quad \Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

$$v_{k+1}(s) \;\dot{=}\; \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s]$$

$$= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_k(s')\Big], \tag{4.5}$$

Example 4.1 Consider the 4×4 gridworld shown below.

$R_t = -1$ on all transitions

The nonterminal states are $\mathcal{S} = \{1, 2, \ldots, 14\}$. There are four actions possible in each state, $\mathcal{A} = \{\texttt{up}, \texttt{down}, \texttt{right}, \texttt{left}\}$, which deterministically cause the corresponding state transitions, except that actions that would take the agent off the grid in fact leave the state unchanged. Thus, for instance, $p(6, -1|5, \texttt{right}) = 1$, $p(7, -1|7, \texttt{right}) = 1$, and $p(10, r|5, \texttt{right}) = 0$ for all $r \in \mathcal{R}$. This is an undiscounted, episodic task. The reward is -1 on all transitions until the terminal state is reached. The terminal state is shaded in the figure (although it is shown in two places, it is formally one state). The expected reward function is thus $r(s, a, s') = -1$ for all states s, s' and actions a. Suppose the agent follows the equiprobable random policy (all actions equally likely). The left side of Figure 4.1 shows the sequence of value functions $\{v_k\}$ computed by iterative policy evaluation. The final estimate is in fact v_π, which in this case gives for each state the negation of the expected number of steps from that state until termination. ∎
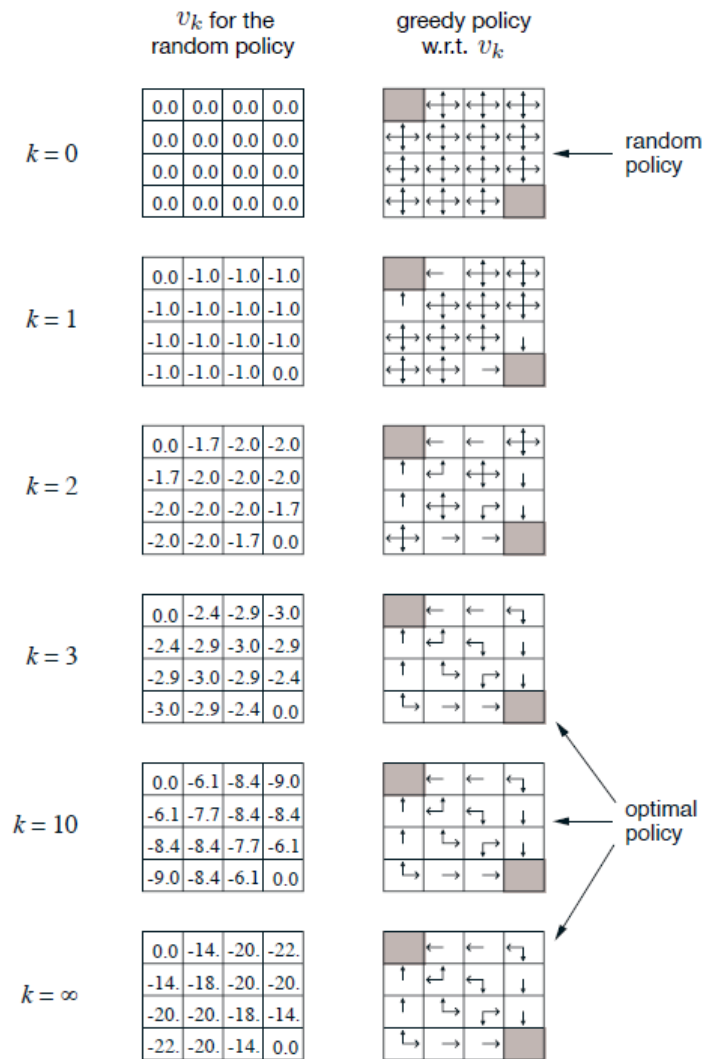

Figure 4.1: Convergence of iterative policy evaluation on a small gridworld. The left column is the sequence of approximations of the state-value function for the random policy (all actions equally likely). The right column is the sequence of greedy policies corresponding to the value function estimates (arrows are shown for all actions achieving the maximum, and the numbers shown are rounded to two significant digits). The last policy is guaranteed only to be an improvement over the random policy, but in this case it, and all policies after the third iteration, are optimal.

4.2   Policy Improvement

Our reason for computing the value function for a policy is to help find better policies. Suppose we have determined the value function v_π for an arbitrary deterministic policy π. For some state s we would like to know whether or not we should change the policy to deterministically choose an action $a \neq \pi(s)$. We know how good it is to follow the current policy from s—that is $v_\pi(s)$—but would it be better or worse to change to the new policy? One way to answer this question is to consider selecting a in s and thereafter following the existing policy, π. The value of this way of behaving is

$$
\begin{aligned}
q_\pi(s,a) &\doteq \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \qquad (4.6)\\
&= \sum_{s',r} p(s',r \mid s,a)\Big[r + \gamma v_\pi(s')\Big].
\end{aligned}
$$

The key criterion is whether this is greater than or less than $v_\pi(s)$. If it is greater—that is, if it is better to select a once in s and thereafter follow π than it would be to follow π all the time—then one would expect it to be better still to select a every time s is encountered, and that the new policy would in fact be a better one overall.

   That this is true is a special case of a general result called the *policy improvement theorem*. Let π and π' be any pair of deterministic policies such that, for all $s \in \mathcal{S}$,

$$
q_\pi(s, \pi'(s)) \geq v_\pi(s). \qquad (4.7)
$$

Then the policy π' must be as good as, or better than, π. That is, it must obtain greater or equal expected return from all states $s \in \mathcal{S}$:

$$
v_{\pi'}(s) \geq v_\pi(s). \qquad (4.8)
$$

Moreover, if there is strict inequality of (4.7) at any state, then there must be strict inequality of (4.8) at that state.

So far we have seen how, given a policy and its value function, we can easily evaluate a change in the policy at a single state. It is a natural extension to consider changes at *all* states, selecting at each state the action that appears best according to $q_\pi(s, a)$. In other words, to consider the new *greedy* policy, π', given by

$$
\begin{aligned}
\pi'(s) & \doteq \underset{a}{\arg\max}\, q_\pi(s, a) \\
& = \underset{a}{\arg\max}\, \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \qquad (4.9) \\
& = \underset{a}{\arg\max} \sum_{s', r} p(s', r \mid s, a) \Big[r + \gamma v_\pi(s') \Big],
\end{aligned}
$$

where $\arg\max_a$ denotes the value of a at which the expression that follows is maximized (with ties broken arbitrarily). The greedy policy takes the action that looks best in the short term—after one step of lookahead—according to v_π. By construction, the greedy policy meets the conditions of the policy improvement theorem (4.7), so we know that it is as good as, or better than, the original policy. The process of making a new policy that improves on an original policy, by making it greedy with respect to the value function of the original policy, is called *policy improvement.*

Suppose the new greedy policy, π', is as good as, but not better than, the old policy π. Then $v_\pi = v_{\pi'}$, and from (4.9) it follows that for all $s \in \mathcal{S}$:

$$
\begin{aligned}
v_{\pi'}(s) & = \max_a \mathbb{E}[R_{t+1} + \gamma v_{\pi'}(S_{t+1}) \mid S_t = s, A_t = a] \\
& = \max_a \sum_{s', r} p(s', r \mid s, a) \Big[r + \gamma v_{\pi'}(s') \Big].
\end{aligned}
$$

But this is the same as the Bellman optimality equation (4.1), and therefore, $v_{\pi'}$ must be v_*, and both π and π' must be optimal policies. Policy improvement thus must give us a strictly better policy except when the original policy is already optimal.

4.3 Policy Iteration

Once a policy, π, has been improved using v_π to yield a better policy, π', we can then compute $v_{\pi'}$ and improve it again to yield an even better π''. We can thus obtain a sequence of monotonically improving policies and value functions:

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*,$$

where $\xrightarrow{\text{E}}$ denotes a policy *evaluation* and $\xrightarrow{\text{I}}$ denotes a policy *improvement*.

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
   $\quad \Delta \leftarrow 0$
   $\quad$ Loop for each $s \in \mathcal{S}$:
   $\quad\quad v \leftarrow V(s)$
   $\quad\quad V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s))[r + \gamma V(s')]$
   $\quad\quad \Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   $\quad$ *old-action* $\leftarrow \pi(s)$
   $\quad \pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$
   $\quad$ If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Example 4.2: Jack's Car Rental Jack manages two locations for a nationwide car rental company. Each day, some number of customers arrive at each location to rent cars. If Jack has a car available, he rents it out and is credited $10 by the national company. If he is out of cars at that location, then the business is lost. Cars become available for renting the day after they are returned. To help ensure that cars are available where they are needed, Jack can move them between the two locations overnight, at a cost of $2 per car moved. We assume that the number of cars requested and returned at each location are Poisson random variables, meaning that the probability that the number is n is $\frac{\lambda^n}{n!}e^{-\lambda}$, where λ is the expected number. Suppose λ is 3 and 4 for rental requests at the first and second locations and 3 and 2 for returns. To simplify the problem slightly, we assume that there can be no more than 20 cars at each location (any additional cars are returned to the nationwide company, and thus disappear from the problem) and a maximum of five cars can be moved from one location to the other in one night. We take the discount rate to be $\gamma = 0.9$ and formulate this as a continuing finite MDP, where the time steps are days, the state is the number of cars at each location at the end of the day, and the actions are the net numbers of cars moved between the two locations overnight. Figure 4.2 shows the sequence of policies found by policy iteration starting from the policy that never moves any cars.

Policy iteration often converges in surprisingly few iterations, as illustrated in the example of Jack's car rental and in the example in Figure 4.1. The bottom-left diagram of Figure 4.1 shows the value function for the equiprobable random policy, and the bottom-right diagram shows a greedy policy for this value function. The policy improvement theorem assures us that these policies are better than the original random policy. In this case, however, these policies are not just better, but optimal, proceeding to the terminal states in the minimum number of steps. In this example, policy iteration would find the optimal policy after just one iteration.
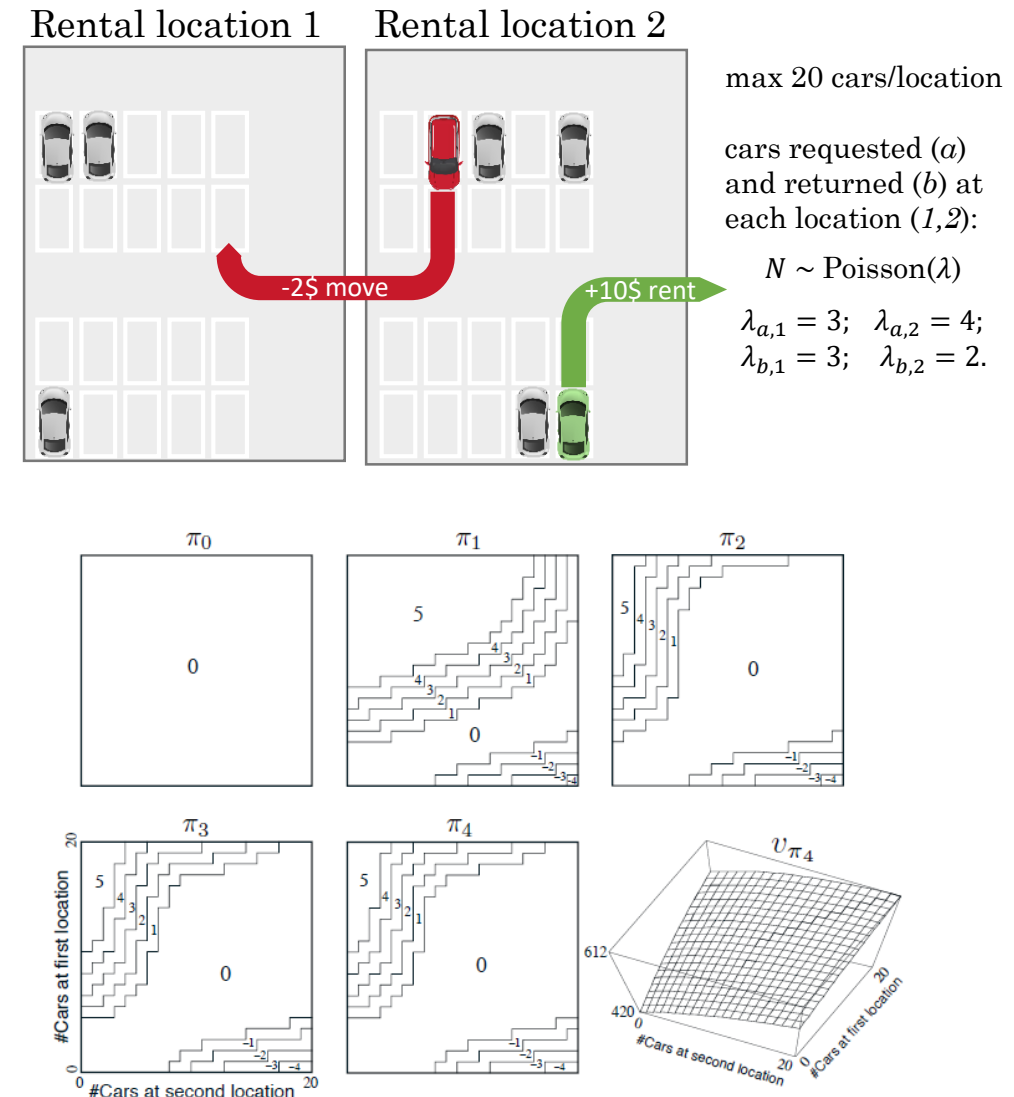

max 20 cars/location

cars requested (a) and returned (b) at each location ($1,2$):

$$N \sim \text{Poisson}(\lambda)$$

$\lambda_{a,1} = 3; \quad \lambda_{a,2} = 4;$
$\lambda_{b,1} = 3; \quad \lambda_{b,2} = 2.$

Figure 4.2: The sequence of policies found by policy iteration on Jack's car rental problem, and the final state-value function. The first five diagrams show, for each number of cars at each location at the end of the day, the number of cars to be moved from the first location to the second (negative numbers indicate transfers from the second location to the first). Each successive policy is a strict improvement over the previous policy, and the last policy is optimal. ■

4.4 Value Iteration

One drawback to policy iteration is that each of its iterations involves policy evaluation, which may itself be a protracted iterative computation requiring multiple sweeps through the state set. If policy evaluation is done iteratively, then convergence exactly to v_π occurs only in the limit. Must we wait for exact convergence, or can we stop short of that? The example in Figure 4.1 certainly suggests that it may be possible to truncate policy evaluation. In that example, policy evaluation iterations beyond the first three have no effect on the corresponding greedy policy.

In fact, the policy evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration. One important special case is when policy evaluation is stopped after just one sweep (one update of each state). This algorithm is called *value iteration*. It can be written as a particularly simple update operation that combines the policy improvement and truncated policy evaluation steps:

$$v_{k+1}(s) \doteq \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a]$$

$$= \max_a \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma v_k(s')\Big], \tag{4.10}$$

for all $s \in \mathcal{S}$. For arbitrary v_0, the sequence $\{v_k\}$ can be shown to converge to v_* under the same conditions that guarantee the existence of v_*.

Another way of understanding value iteration is by reference to the Bellman optimality equation (4.1). Note that value iteration is obtained simply by turning the Bellman optimality equation into an update rule. Also note how the value iteration update is identical to the policy evaluation update (4.5) except that it requires the maximum to be taken over all actions. Another way of seeing this close relationship is to compare the backup diagrams for these algorithms on page 59 (policy evaluation) and on the left of Figure 3.4 (value iteration). These two are the natural backup operations for computing v_π and v_*.

Finally, let us consider how value iteration terminates. Like policy evaluation, value iteration formally requires an infinite number of iterations to converge exactly to v_*. In practice, we stop once the value function changes by only a small amount in a sweep.
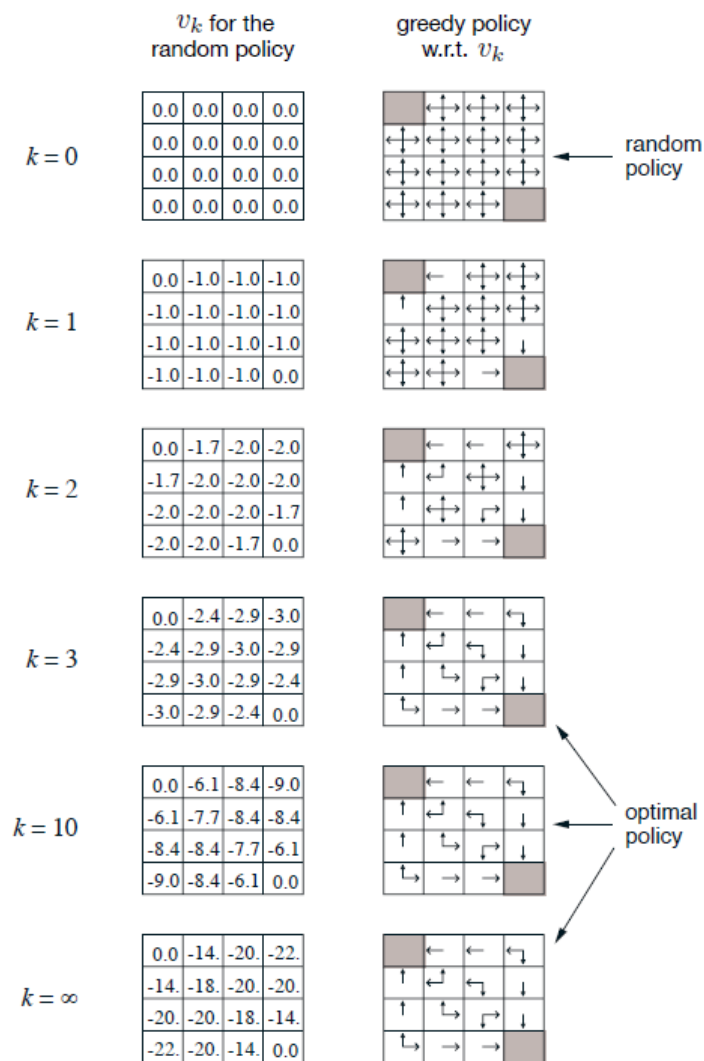

Figure 4.1: Convergence of iterative policy evaluation on a small gridworld. The left column is the sequence of approximations of the state-value function for the random policy (all actions equally likely). The right column is the sequence of greedy policies corresponding to the value function estimates (arrows are shown for all actions achieving the maximum, and the numbers shown are rounded to two significant digits). The last policy is guaranteed only to be an improvement over the random policy, but in this case it, and all policies after the third iteration, are optimal.

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
  | $\Delta \leftarrow 0$
  | Loop for each $s \in \mathcal{S}$:
  | $v \leftarrow V(s)$
  | $V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
  | $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
    $\pi(s) = \arg\max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$

Value iteration effectively combines, in each of its sweeps, one sweep of policy evaluation and one sweep of policy improvement. Faster convergence is often achieved by interposing multiple policy evaluation sweeps between each policy improvement sweep. In general, the entire class of truncated policy iteration algorithms can be thought of as sequences of sweeps, some of which use policy evaluation updates and some of which use value iteration updates. Because the max operation in (4.10) is the only difference between these updates, this just means that the max operation is added to some sweeps of policy evaluation. All of these algorithms converge to an optimal policy for discounted finite MDPs.

Example 4.3: Gambler's Problem A gambler has the opportunity to make bets on the outcomes of a sequence of coin flips. If the coin comes up heads, he wins as many dollars as he has staked on that flip; if it is tails, he loses his stake. The game ends when the gambler wins by reaching his goal of $100, or loses by running out of money. On each flip, the gambler must decide what portion of his capital to stake, in integer numbers of dollars. This problem can be formulated as an undiscounted, episodic, finite MDP. The state is the gambler's capital, $s \in \{1, 2, \dots, 99\}$ and the actions are stakes, $a \in \{0, 1, \dots, \min(s, 100 - s)\}$. The reward is zero on all transitions except those on which the gambler reaches his goal, when it is $+1$. The state-value function then gives the probability of winning from each state. A policy is a mapping from levels of capital to stakes. The optimal policy maximizes the probability of reaching the goal. Let p_h denote the probability of the coin coming up heads. If p_h is known, then the entire problem is known and it can be solved, for instance, by value iteration. Figure 4.3 shows the change in the value function over successive sweeps of value iteration, and the final policy found, for the case of $p_h = 0.4$. This policy is optimal, but not unique. In fact, there is a whole family of optimal policies, all corresponding to ties for the argmax action selection with respect to the optimal value function. Can you guess what the entire family looks like?
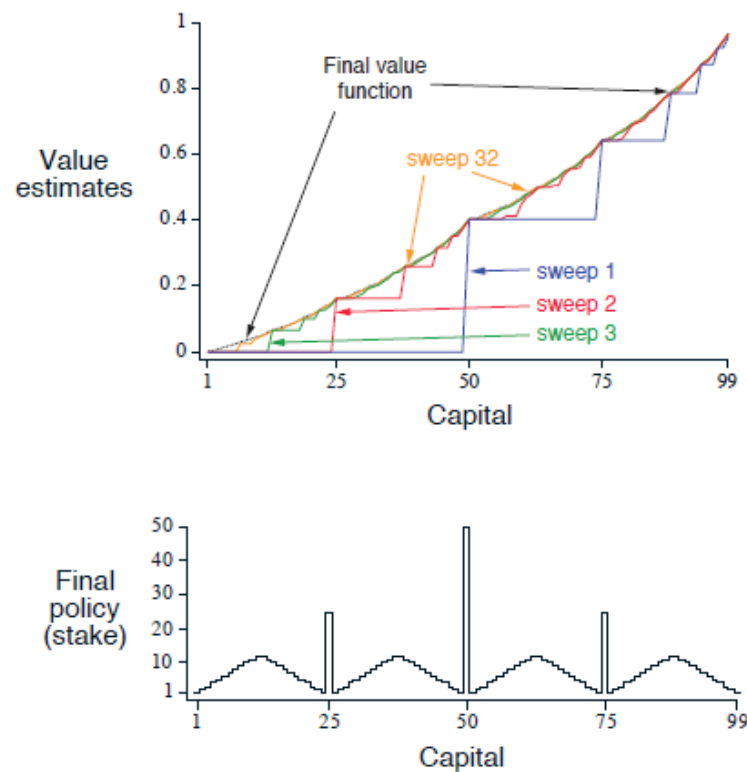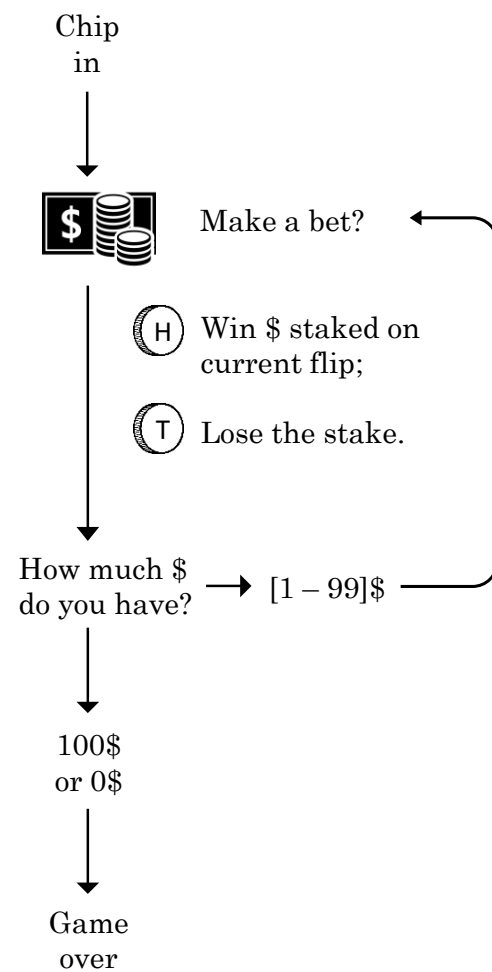
Chip
in

$ Make a bet?

(H) Win $ staked on current flip;

(T) Lose the stake.

How much $ do you have? → [1 − 99]$

100$ or 0$

Game over

Value estimates

Capital

Final policy (stake)

Capital

Figure 4.3: The solution to the gambler's problem for $p_h = 0.4$. The upper graph shows the value function found by successive sweeps of value iteration. The lower graph shows the final policy. ∎

4.5 Asynchronous Dynamic Programming

A major drawback to the DP methods that we have discussed so far is that they involve operations over the entire state set of the MDP, that is, they require sweeps of the state set. If the state set is very large, then even a single sweep can be prohibitively expensive. For example, the game of backgammon has over 10^{20} states. Even if we could perform the value iteration update on a million states per second, it would take over a thousand years to complete a single sweep.

Asynchronous DP algorithms are in-place iterative DP algorithms that are not organized in terms of systematic sweeps of the state set. These algorithms update the values of states in any order whatsoever, using whatever values of other states happen to be available. The values of some states may be updated several times before the values of others are updated once. To converge correctly, however, an asynchronous algorithm must continue to update the values of all the states: it can't ignore any state after some point in the computation. Asynchronous DP algorithms allow great flexibility in selecting states to update.

For example, one version of asynchronous value iteration updates the value, in place, of only one state, s_k, on each step, k, using the value iteration update (4.10). If $0 \le \gamma < 1$, asymptotic convergence to v_* is guaranteed given only that all states occur in the sequence $\{s_k\}$ an infinite number of times (the sequence could even be random).[1] Similarly, it is possible to intermix policy evaluation and value iteration updates to produce a kind of asynchronous truncated policy iteration. Although the details of this and other more unusual DP algorithms are beyond the scope of this book, it is clear that a few different updates form building blocks that can be used flexibly in a wide variety of sweepless DP algorithms.

Of course, avoiding sweeps does not necessarily mean that we can get away with less computation. It just means that an algorithm does not need to get locked into any hopelessly long sweep before it can make progress improving a policy. We can try to take advantage of this flexibility by selecting the states to which we apply updates so as to improve the algorithm's rate of progress. We can try to order the updates to let value information propagate from state to state in an efficient way. Some states may not

15

need their values updated as often as others. We might even try to skip updating some states entirely if they are not relevant to optimal behavior. Some ideas for doing this are discussed in Chapter 8.

Asynchronous algorithms also make it easier to intermix computation with real-time interaction. To solve a given MDP, we can run an iterative DP algorithm *at the same time that an agent is actually experiencing the MDP*. The agent's experience can be used to determine the states to which the DP algorithm applies its updates. At the same time, the latest value and policy information from the DP algorithm can guide the agent's decision making. For example, we can apply updates to states as the agent visits them. This makes it possible to *focus* the DP algorithm's updates onto parts of the state set that are most relevant to the agent. This kind of focusing is a repeated theme in reinforcement learning.

4.6 Generalized Policy Iteration

Policy iteration consists of two simultaneous, interacting processes, one making the value function consistent with the current policy (policy evaluation), and the other making the policy greedy with respect to the current value function (policy improvement). In policy iteration, these two processes alternate, each completing before the other begins, but this is not really necessary. In value iteration, for example, only a single iteration of policy evaluation is performed in between each policy improvement. In asynchronous DP methods, the evaluation and improvement processes are interleaved at an even finer grain. In some cases a single state is updated in one process before returning to the other. As long as both processes continue to update all states, the ultimate result is typically the same—convergence to the optimal value function and an optimal policy.

We use the term *generalized policy iteration* (GPI) to refer to the general idea of letting policy-evaluation and policy-improvement processes interact, independent of the granularity and other details of the two processes. Almost all reinforcement learning methods are well described as GPI. That is, all have identifiable policies and value functions, with the policy always being improved with respect to the value function and the value function always being driven toward the value function for the policy, as suggested by the diagram to the right. If both the evaluation process and the improvement process stabilize, that is, no longer produce changes, then the value function and policy must be optimal. The value function stabilizes only when it is consistent with the current policy, and the policy stabilizes only when it is greedy with respect to the current value function.

evaluation

$V \rightsquigarrow v_\pi$

π V

$\pi \rightsquigarrow \text{greedy}(V)$

improvement

$\pi_* \rightleftharpoons v_*$

Thus, both processes stabilize only when a policy has been found that is greedy with respect to its own evaluation function. This implies that the Bellman optimality equation (4.1) holds, and thus that the policy and the value function are optimal.

17
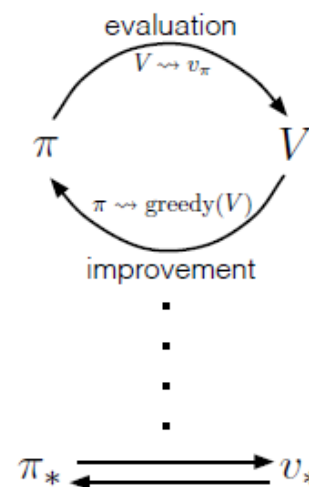
The evaluation and improvement processes in GPI can be viewed as both competing and cooperating. They compete in the sense that they pull in opposing directions. Making the policy greedy with respect to the value function typically makes the value function incorrect for the changed policy, and making the value function consistent with the policy typically causes that policy no longer to be greedy. In the long run, however, these two processes interact to find a single joint solution: the optimal value function and an optimal policy.

One might also think of the interaction between the evaluation and improvement processes in GPI in terms of two constraints or goals—for example, as two lines in a two-dimensional space as suggested by the diagram to the right. Although the real geometry is much more complicated than this, the diagram suggests what happens in the real case. Each process drives the value function or policy toward one of the lines representing a solution to one of the two goals. The goals interact because the two lines are not orthogonal. Driving directly toward one goal causes some movement away from the other goal. Inevitably, however, the joint process is brought closer to the overall goal of optimality. The arrows in this diagram correspond to the behavior of policy iteration in that each takes the system all the way to achieving one of the two goals completely. In GPI one could also take smaller, incomplete steps toward each goal. In either case, the two processes together achieve the overall goal of optimality even though neither is attempting to achieve it directly.
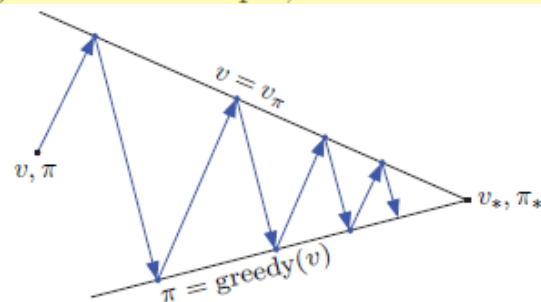
4.7 Efficiency of Dynamic Programming

DP may not be practical for very large problems, but compared with other methods for solving MDPs, DP methods are actually quite efficient. If we ignore a few technical details, then, in the worst case, the time that DP methods take to find an optimal policy is polynomial in the number of states and actions. If n and k denote the number of states and actions, this means that a DP method takes a number of computational operations that is less than some polynomial function of n and k. A DP method is guaranteed to find an optimal policy in polynomial time even though the total number of (deterministic) policies is k^n. In this sense, DP is exponentially faster than any direct search in policy space could be, because direct search would have to exhaustively examine each policy to provide the same guarantee. Linear programming methods can also be used to solve MDPs, and in some cases their worst-case convergence guarantees are better than those of DP methods. But linear programming methods become impractical at a much smaller number of states than do DP methods (by a factor of about 100). For the largest problems, only DP methods are feasible.

DP is sometimes thought to be of limited applicability because of the *curse of dimensionality*, the fact that the number of states often grows exponentially with the number of state variables. Large state sets do create difficulties, but these are inherent difficulties of the problem, not of DP as a solution method. In fact, DP is comparatively better suited to handling large state spaces than competing methods such as direct search and linear programming.

In practice, DP methods can be used with today's computers to solve MDPs with millions of states. Both policy iteration and value iteration are widely used, and it is not clear which, if either, is better in general. In practice, these methods usually converge much faster than their theoretical worst-case run times, particularly if they are started with good initial value functions or policies.

On problems with large state spaces, *asynchronous* DP methods are often preferred. To complete even one sweep of a synchronous method requires computation and memory for every state. For some problems, even this much memory and computation is impractical, yet the problem is still potentially solvable because relatively few states occur along optimal solution trajectories. Asynchronous methods and other variations of GPI can be applied in such cases and may find good or optimal policies much faster than synchronous methods can.

4.8   Summary

In this chapter we have become familiar with the basic ideas and algorithms of dynamic programming as they relate to solving finite MDPs. *Policy evaluation* refers to the (typically) iterative computation of the value function for a given policy. *Policy improvement* refers to the computation of an improved policy given the value function for that policy. Putting these two computations together, we obtain *policy iteration* and *value iteration*, the two most popular DP methods. Either of these can be used to reliably compute optimal policies and value functions for finite MDPs given complete knowledge of the MDP.

Classical DP methods operate in sweeps through the state set, performing an *expected update* operation on each state. Each such operation updates the value of one state based on the values of all possible successor states and their probabilities of occurring. Expected updates are closely related to Bellman equations: they are little more than these equations turned into assignment statements. When the updates no longer result in any changes in value, convergence has occurred to values that satisfy the corresponding Bellman equation. Just as there are four primary value functions (v_π, v_*, q_π, and q_*), there are four corresponding Bellman equations and four corresponding expected updates. An intuitive view of the operation of DP updates is given by their *backup diagrams*.

Insight into DP methods and, in fact, into almost all reinforcement learning methods, can be gained by viewing them as *generalized policy iteration* (GPI). GPI is the general idea of two interacting processes revolving around an approximate policy and an approximate value function. One process takes the policy as given and performs some form of policy evaluation, changing the value function to be more like the true value function for the policy. The other process takes the value function as given and performs some form of policy improvement, changing the policy to make it better, assuming that the value function is its value function. Although each process changes the basis for the other, overall they work together to find a joint solution: a policy and value function that are unchanged by either process and, consequently, are optimal. In some cases, GPI can be proved to converge, most notably for the classical DP methods that we have presented in this chapter. In other cases convergence has not been proved, but still the idea of GPI improves our understanding of the methods.

It is not necessary to perform DP methods in complete sweeps through the state set. *Asynchronous DP* methods are in-place iterative methods that update states in an arbitrary order, perhaps stochastically determined and using out-of-date information. Many of these methods can be viewed as fine-grained forms of GPI.

Finally, we note one last special property of DP methods. All of them update estimates of the values of states based on estimates of the values of successor states. That is, they update estimates on the basis of other estimates. We call this general idea *bootstrapping*. Many reinforcement learning methods perform bootstrapping, even those that do not require, as DP requires, a complete and accurate model of the environment. In the next chapter we explore reinforcement learning methods that do not require a model and do not bootstrap. In the chapter after that we explore methods that do not require a model but do bootstrap. These key features and properties are separable, yet can be mixed in interesting combinations.

SUMMARY

- **4.1 Policy Evaluation (Prediction)**
  - Iterative Policy Evaluation
  - **Ex 4.1** 4×4 gridworld game

- **4.2 Policy Improvement**
  - Greedy Policy Improvement

- **4.3 Policy Iteration**
  - **Ex 4.2** Jack's car rental

- **4.4 Value Iteration**
  - **Ex 4.3** Gambler's problem

- **4.5 Asynchronous Dynamic Programming**

- **4.6 Generalized Policy Iteration**

- **4.7 Efficiency of Dynamic Programming**

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big],$$

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_k(s')\Big],$$

$$q_\pi(s,\pi'(s)) \ge v_\pi(s) \quad v_{\pi'}(s) \ge v_\pi(s).$$

$$\pi'(s) = \arg\max_a \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big],$$

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*,$$

evaluation

$V \rightsquigarrow v_\pi$

π V

$\pi \rightsquigarrow \text{greedy}(V)$

improvement

$\pi_* \rightleftarrows v_*$