UNIT-IV

Dynamic Programming: Policy Evaluation, Policy Improvement, Policy Iteration, Value Iteration, Asynchronous Dynamic Programming, Generalized Policy Iteration, Efficiency of Dynamic Programming.

Dynamic Programming:

Bellman Optimality Equation

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v_*(s')\Big] \end{aligned} \tag{4.1}$$

or

$$\begin{aligned} q_*(s, a) &= \mathbb{E}\Big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \;\Big|\; S_t = s, A_t = a\Big] \\ &= \sum_{s',r} p(s', r|s, a)\Big[r + \gamma \max_{a'} q_*(s', a')\Big], \end{aligned} \tag{4.2}$$

for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, and $s' \in \mathcal{S}^+$. As we shall see, 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.

1. Policy Evaluation

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 $s \in \mathcal{S}$,

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi\big[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \;\big|\; S_t = s\big] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \qquad (4.3) \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v_\pi(s')\Big], \qquad (4.4) \end{aligned}$$

where $\pi(a|s)$ is the probability of taking action a in state s under policy π, and the expectations are subscripted by π to indicate that they are conditional on π being followed. The existence and uniqueness of v_π are guaranteed as long as either $\gamma < 1$ or eventual termination is guaranteed from all states under the policy π.

- If the environment's dynamics are completely known, then (4.4) is a system of |S| simultaneous linear equations in |S| unknowns (the $v_\pi(s)$, $s \in S$).
- In principle, its solution is a straightforward, if tedious, computation.

- For our purposes, iterative solution methods are most suitable. Consider a sequence of approximate value functions v0, v1, v2. . .
- Each mapping S^+ to $\mathbb{R}$. .
- The initial approximation, v0, is chosen arbitrarily and each successive approximation is obtained by using the Bellman equation for v_π (3.12) as an update rule:

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

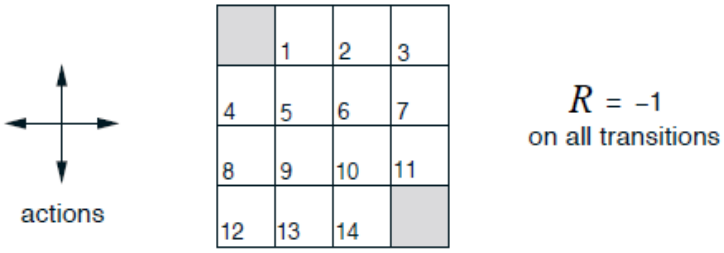
for all $s \in \mathcal{S}$. Clearly, $v_k = v_\pi$ is a fixed point for this update rule because the Bellman equation for v_π assures us of equality in this case. Indeed, the sequence $\{v_k\}$ can be shown in general to converge to v_π as $k \to \infty$ under the same conditions that guarantee the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

Input π, the policy to be evaluated
Initialize an array $V(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $V \approx v_\pi$

Figure 4.1: Iterative policy evaluation.

Example:

Example 4.1 Consider the 4×4 gridworld shown below.



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, $p(6|5, \texttt{right}) = 1$, $p(10|5, \texttt{right}) = 0$, and $p(7|7, \texttt{right}) = 1$. 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 4.2 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

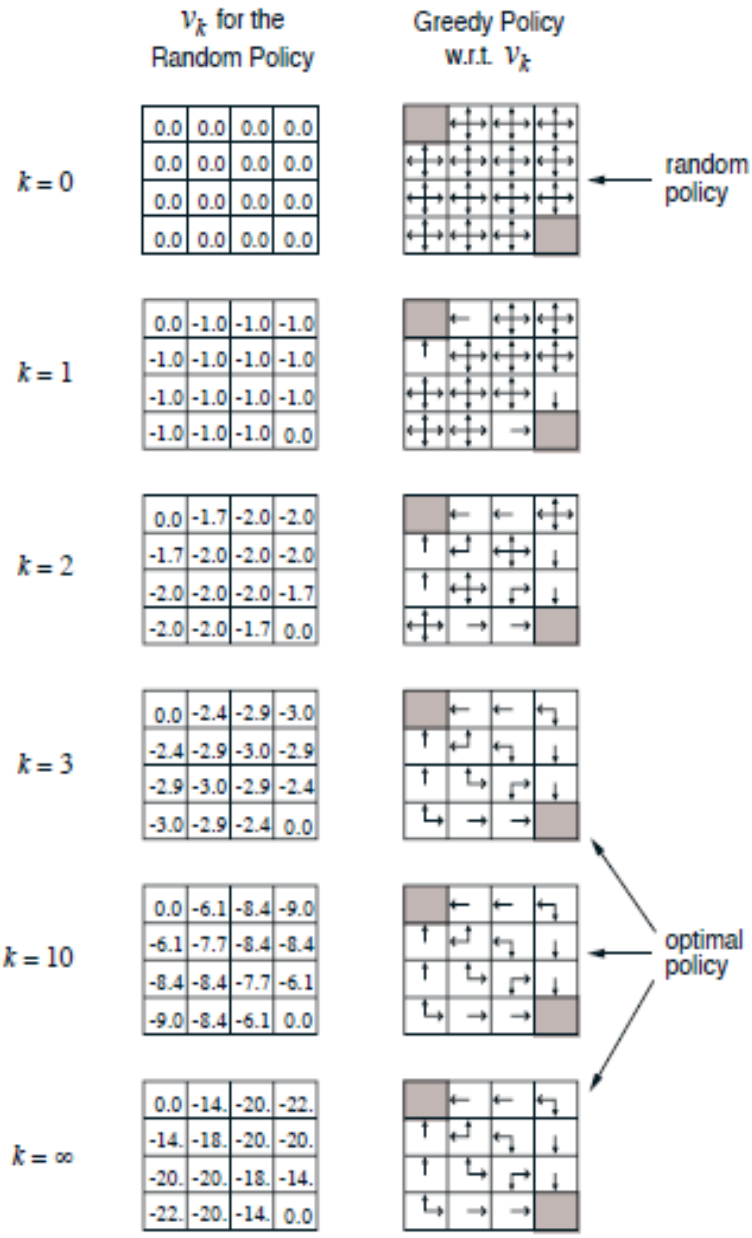


Figure 4.2: Convergence of iterative policy evaluation on a small gridworld. The left column is the sequence of approximations of the state-value function for the random policy (all actions equal). The right column is the sequence of greedy policies corresponding to the value function estimates (arrows are shown for all actions achieving the maximum). 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.

termination.

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

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

- 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 $s \in \mathcal{S}$,

$$q_\pi(s, \pi'(s)) \geq v_\pi(s). \tag{4.7}$$

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 $s \in \mathcal{S}$:

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

The idea behind the proof of the policy improvement theorem is easy to understand. Starting from (4.7), we keep expanding the q_π side and reapplying

(4.7) until we get $v_{\pi'}(s)$:

$$\begin{aligned}
v_\pi(s) &\leq q_\pi(s, \pi'(s)) \\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) \mid S_t = s] \\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma \mathbb{E}_{\pi'}[R_{t+2} + \gamma v_\pi(S_{t+2})] \mid S_t = s] \\
&= \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2}) \mid S_t = s\right] \\
&\leq \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 v_\pi(S_{t+3}) \mid S_t = s\right] \\
&\vdots \\
&\leq \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \mid S_t = s\right] \\
&= v_{\pi'}(s).
\end{aligned}$$

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 to a particular action. It is a natural extension to consider changes at *all* states and to *all* possible actions, 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

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

- where argmax_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 look ahead—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 $s \in \mathcal{S}$:

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

So far in this section we have considered the special case of deterministic policies. In the general case, a stochastic policy π specifies probabilities, $\pi(a|s)$, for taking each action, a, in each state, s. We will not go through the details, but in fact all the ideas of this section extend easily to stochastic policies. In particular, the policy improvement theorem carries through as stated for the stochastic case, under the natural definition:

$$q_\pi(s, \pi'(s)) = \sum_a \pi'(a|s) q_\pi(s, a).$$

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*. Each policy is guaranteed to be a strict improvement over the previous one

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

2. Policy Evaluation
   Repeat
      $\Delta \leftarrow 0$
      For each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\big[r + \gamma V(s')\big]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

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

Figure 4.3: Policy iteration (using iterative policy evaluation) for v∗. This algorithm has a subtle bug, in that it may never terminate if the policy continually switches between two or more policies that are equally good. The bug can be fixed by adding additional flags, but it makes the pseudocode so ugly that it is not worth it. :-)

Because a finite MDP has only a finite number of policies, this process must converge to an optimal policy and optimal value function in a finite number of iterations.

This way of finding an optimal policy is called policy iteration. A complete algorithm is given in Figure 4.3.

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 4.4 shows the sequence of policies found by policy iteration starting from the policy that never moves any cars. ■

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.
- 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 backup of each state).
- This algorithm is called value iteration. It can be written as a particularly simple backup operation that combines the Figure 4.4:

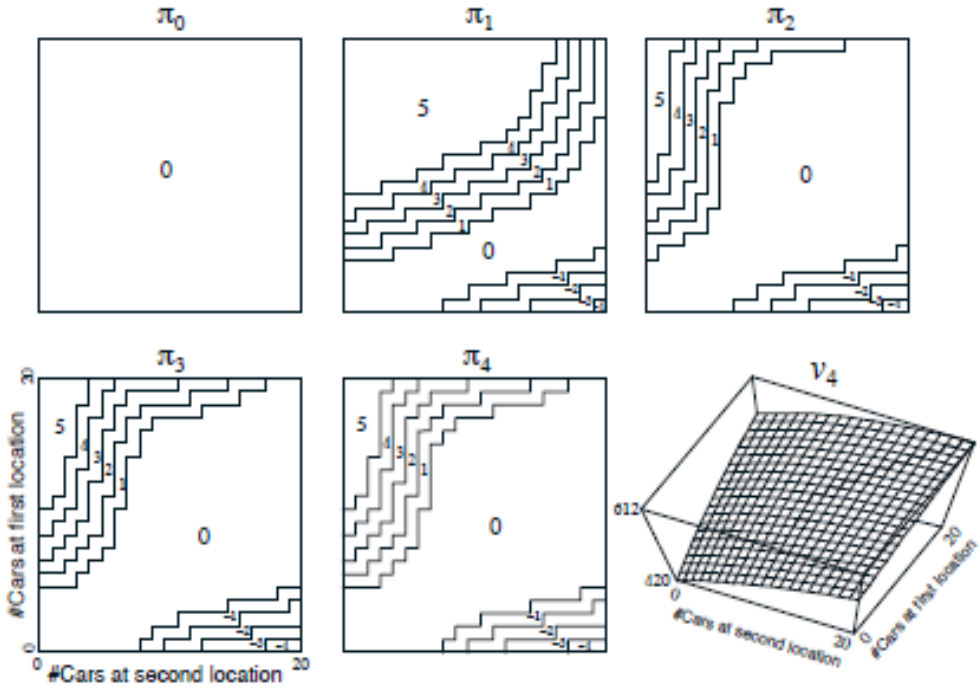


Figure 4.4: The sequence of policies found by policy iteration on Jack's car rental problem, and the final state-value function. The first five diagrams show, for each number of cars at each location at the end of the day, the number of cars to be moved from the first location to the second (negative 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.

policy improvement and truncated policy evaluation steps:

$$\begin{aligned} v_{k+1}(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a] \qquad (4.10) \\ &= \max_a \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v_k(s')\Big], \end{aligned}$$

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 backup is identical to the policy evaluation backup (4.5) except that it requires the maximum to be taken over all actions.

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 4.5 gives a complete value iteration algorithm with this kind of termination condition.

Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
  $\Delta \leftarrow 0$
  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 $\Delta < \theta$ (a small positive number)

Output a deterministic policy, π, such that
  $\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$

Figure 4.5: Value iteration.

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, $s \in \{1, 2, \ldots, 99\}$ and the actions are stakes, $a \in \{0, 1, \ldots, \min(s, 100 - s)\}$. 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 4.6 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? ■

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 1020 states.
- Even if we could perform the value iteration backup 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 back up 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 backed up several times before the values of others are backed up once.
- To converge correctly, however, an asynchronous algorithm must continue to backup 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 which backup operations are applied.

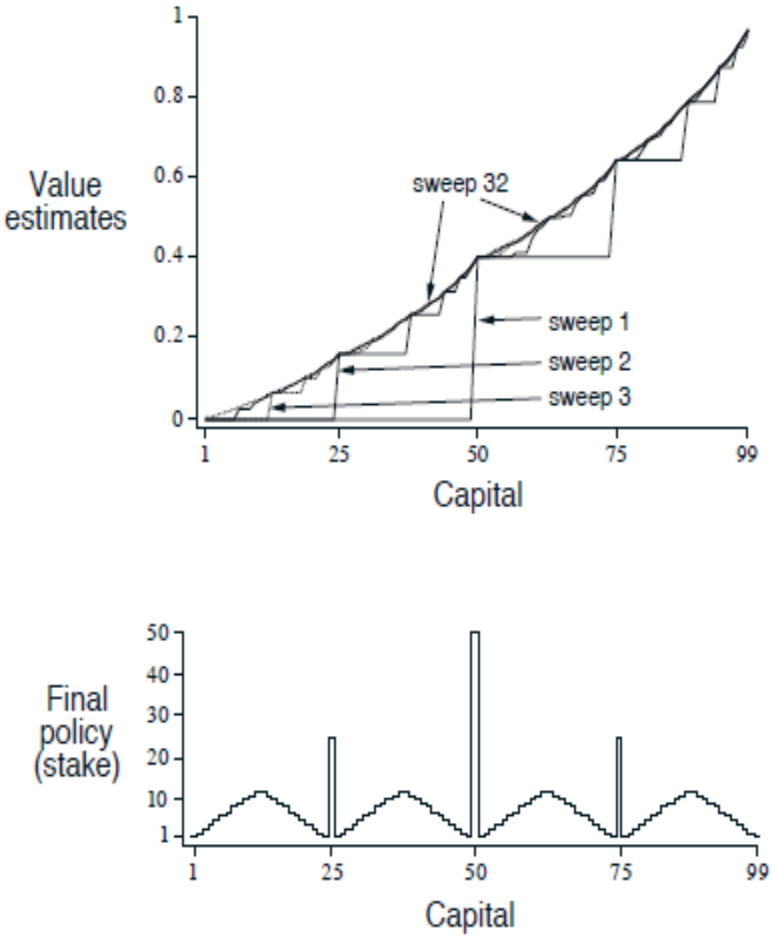


Figure 4.6: The solution to the gambler's problem for $p_h = 0.4$. The upper graph shows the value function found by successive sweeps of value iteration. The lower graph shows the final policy.

- For example, one version of asynchronous value iteration backs up the value, in place, of only one state, sk, on each step, k, using the value iteration backup (4.10).
- $0 \leq \gamma < 1$, asymptotic convergence to $v*$ is guaranteed given only that all states occur in the sequence {sk} an infinite number of times (the sequence could even be stochastic).
- (In the undiscounted episodic case, it is possible that there are some orderings of backups that do not result in convergence, but it is relatively easy to avoid these.) Similarly, it is possible to intermix policy evaluation and value iteration backups 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 backups form building blocks that can be used flexibly in a wide variety of sweepless DP algorithms.
- 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 backups.
- 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 backups to states as the agent visits them.
- This makes it possible to focus the DP algorithm's backups onto parts of the state set that are most relevant to the agent.
- This kind of focusing is a repeated theme in reinforcement learning.

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.
- This overall schema for GPI is illustrated in Figure 4.7.

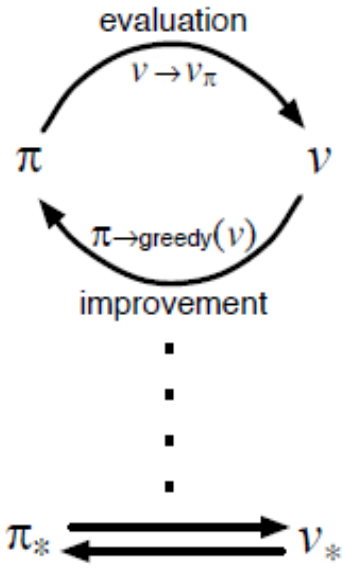
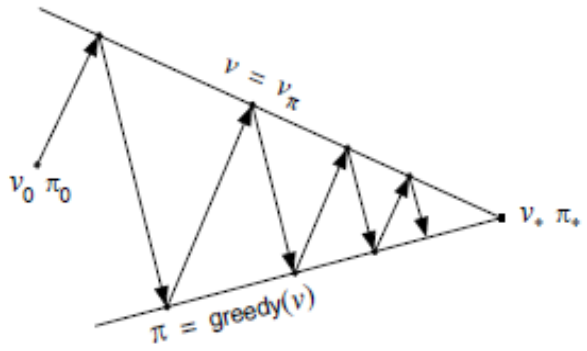


Figure 4.7: Generalized policy iteration: Value and policy functions interact until they are optimal and thus consistent with each other.



- It is easy to see that 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.
- 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.
- 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.

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 the (worst case) time DP methods take to find an optimal policy is polynomial in the number of states and actions.
- If n and m 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 m.
- A DP method is guaranteed to find an optimal policy in polynomial time even though the total number of (deterministic) policies is mn.
- 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 (Bellman, 1957a), 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 are impractical, yet the problem is still potentially solvable because only a 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.