[S&B Book] Chapter 4: Dynamic Programming

Tags

- The key idea of **Dynamic Programming** in reinforcement learning:

  The use of value functions to organize and structure the search for good policies.

▼ 4.1 Policy Evaluation (Prediction)

- ***Policy evaluation:*** How to compute the state-value function v_π for an arbitrary policy π. It is also been referred as the *prediction problem.*
- ***Iterative policy evaluation:***

  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 Bellman equation for v_π as an update rule:

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

  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.

- ***Expected update:***

  The iterative policy evaluation applies the same operation to each state s: it replaces the old value of s with a new value obtained from the old values of the successor states of s, and the expected immediate rewards, along all the one-step transitions possible under the policy being evaluated.

  The algorithms are called *expected* updates because they are based on **an expectation over all possible next states** rather than on a sample next state.

- Two methods of implementing policy evaluation:
  1. Two-array:

     Using two arrays, one for the old values $v_k(s)$, and one for the new values, $v_{k+1}(s)$. With two arrays, the new value can be computed one by one from the old values without the old values being changed.

  2. In-place:

     With each new value immediately overwriting the old one. Then, depending on the order in which the states are updated, sometimes new values are used instead of old ones.

The “in-place” method usually converges faster than the two-array version, because it uses new data as soon as they are available.

- **In-place iterative policy evaluation algorithm:**

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 $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

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

▼ 4.2 Policy Improvement

- The reason for computing the value function for a policy is to help find better policies.
- *Policy improvement theorem:*

  Let π and π' be any pair of deterministic policies such that, for all $s \in \mathcal{S}$, $q_\pi(s, \pi'(s)) \geq v_\pi(s)$. Then the policy π' must be as good as, or better than, π. That is, it must obtain greater or equal expected return for all states $s \in \mathcal{S}$: $v_{\pi'}(s) \geq v_\pi(s)$.

- ***Policy improvement:***

  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.

  The greedy policy takes action that looks best in the short term:

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

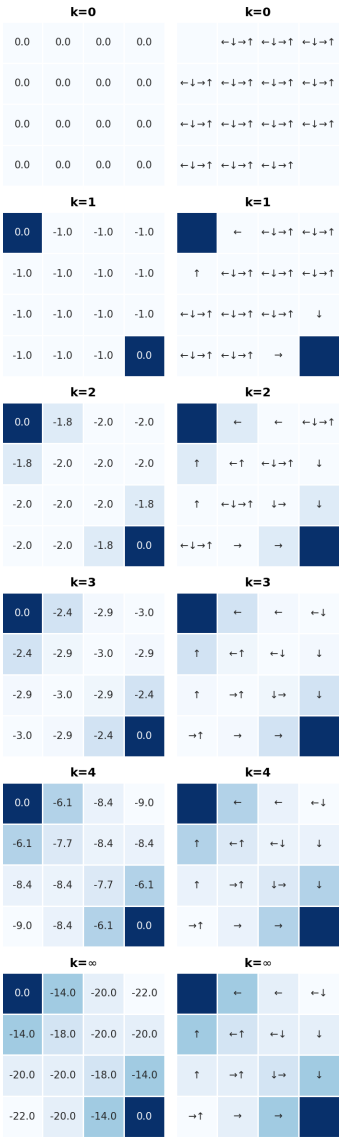
  Suppose the new greedy policy π' is as good as, but not better than, the old policy π, then $v_\pi = v_{\pi'}$, for all $s \in \mathcal{S}$:

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

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

Example

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).



Implementation

Reinforcement-Learning-2nd-Edition-Notes-Codes/example_4_1_policy_evaluation.py at main · terrence-ou/Reinforcement-Learning-2nd-Edition-Not

Notes and code implementations of examples and algorithms of the book Reinforcement Learning, 2nd Edition - Reinforcement-Learning-2nd-Edition-Notes-Codes/example_4_1_policy_evaluation.py at main ·...

https://github.com/terrence-ou/Reinforcement-Learning-2nd-Edition-Notes-Codes/blob/main/chapter_04_dynamic_programming/example_4_1_policy_evaluation.py

▼ 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 function:

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

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

- **This way of finding an optimal policy is called *policy iteration.***

- Algorithm:

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

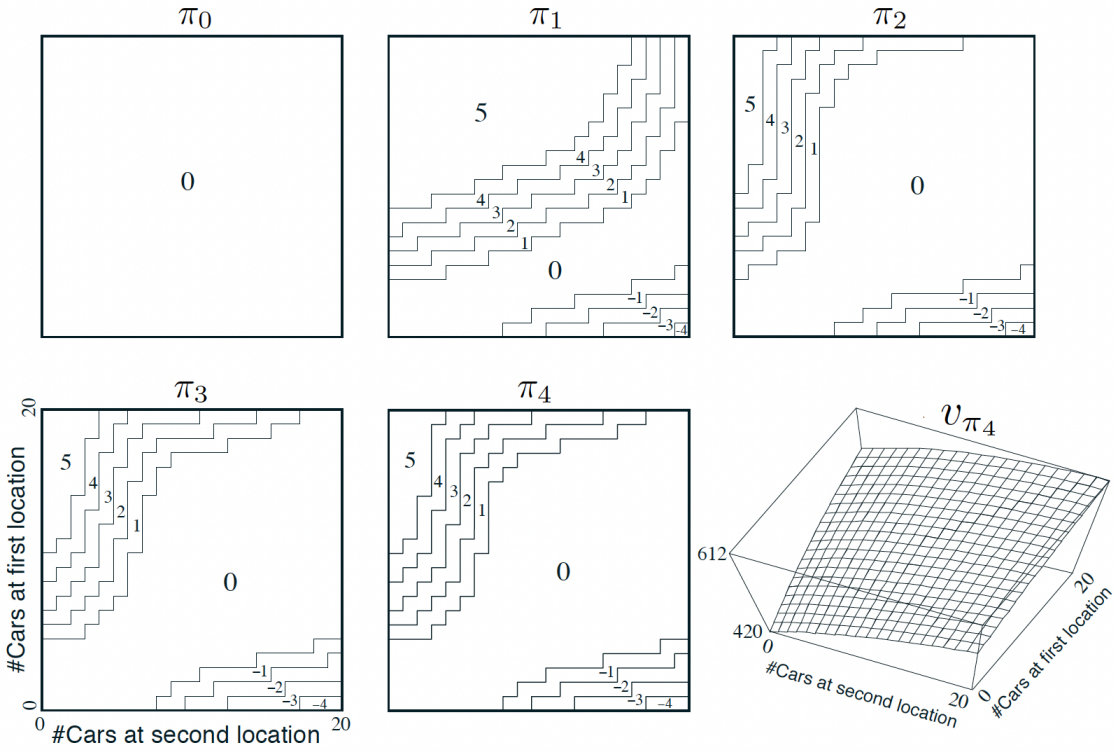
1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

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

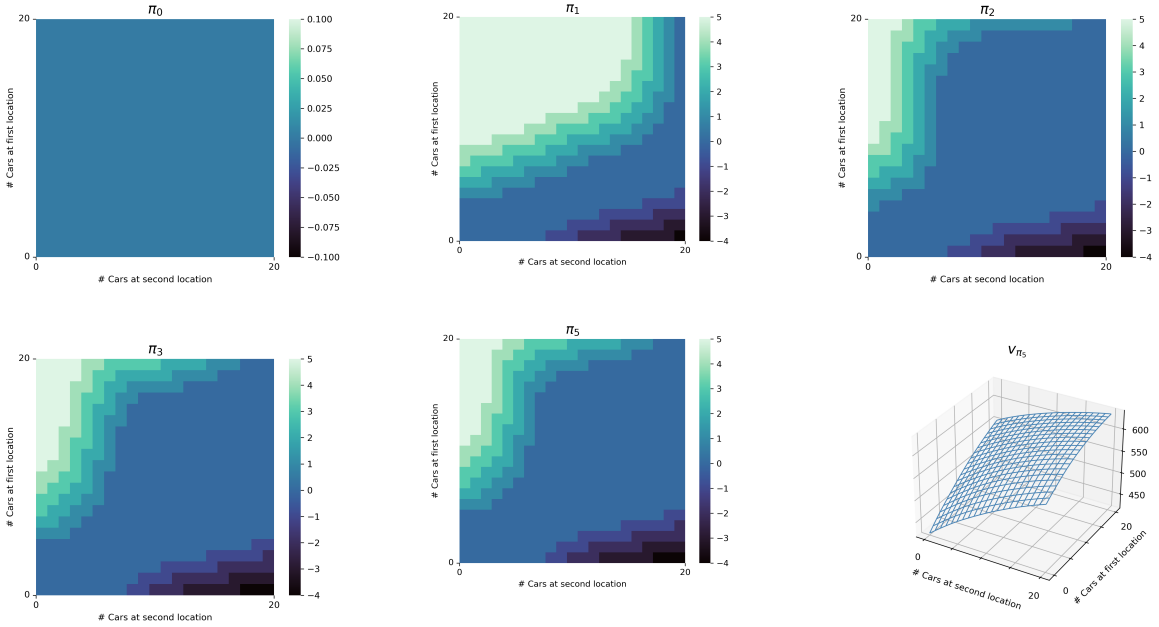
3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      old-action $\leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a) \big[r + \gamma V(s')\big]$
      If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   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:

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).



Implementation:



Reinforcement-Learning-2nd-Edition-Notes-Codes/example_4_2_JacksCarRental.py at main · terrence-ou/Reinforcement-Learning-2nd-Edition-Note

Notes and code implementations of examples and algorithms of the book Reinforcement Learning, 2nd Edition - Reinforcement-Learning-2nd-Edition-Notes-Codes/example_4_2_JacksCarRental.py at main · te...

https://github.com/terrence-ou/Reinforcement-Learning-2nd-Edition-Notes-Codes/blob/main/chapter_04_dynamic_programming/example_4_2_JacksCarRental.py

---

Exercises 4.5:

How would policy iteration be defined for action values? Give a complete algorithm for computing q_*, analogous to that on page 80 for computing v_*. Please pay special attention to this exercise, because the ideas involved will be used throughout the rest of the book.

Solution:

1. **Initialization**

   $Q(s, a) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}$ arbitrarily for all $s \in \mathcal{S}$

2. **Policy Evaluation**

   Loop:

   $\Delta \leftarrow 0$

   Loop for each $s \in \mathcal{S}$ and $a \in \mathcal{A}$:

   $q \leftarrow Q(s, a)$

   $Q(s, a) \leftarrow \sum_{s', r} p(s', r|s, a) \left[r + \gamma \sum_{a'} \pi(a'|s') Q(s', a')\right]$

   $\Delta \leftarrow \max(\Delta, |q - Q(s, a)|)$

   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. **Policy Improvement**

   $policy\_stable \leftarrow true$

   For each $s \in \mathcal{S}$:

   $old\_action \leftarrow \pi(s)$

   $\pi(s) \leftarrow \arg\max_a Q(s, a)$

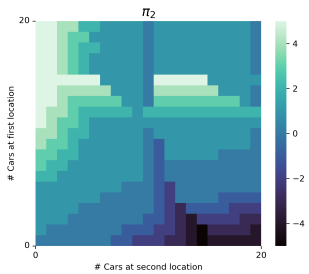
   if $old\_action \neq \pi(s)$, then $polict\_stable \leftarrow false$

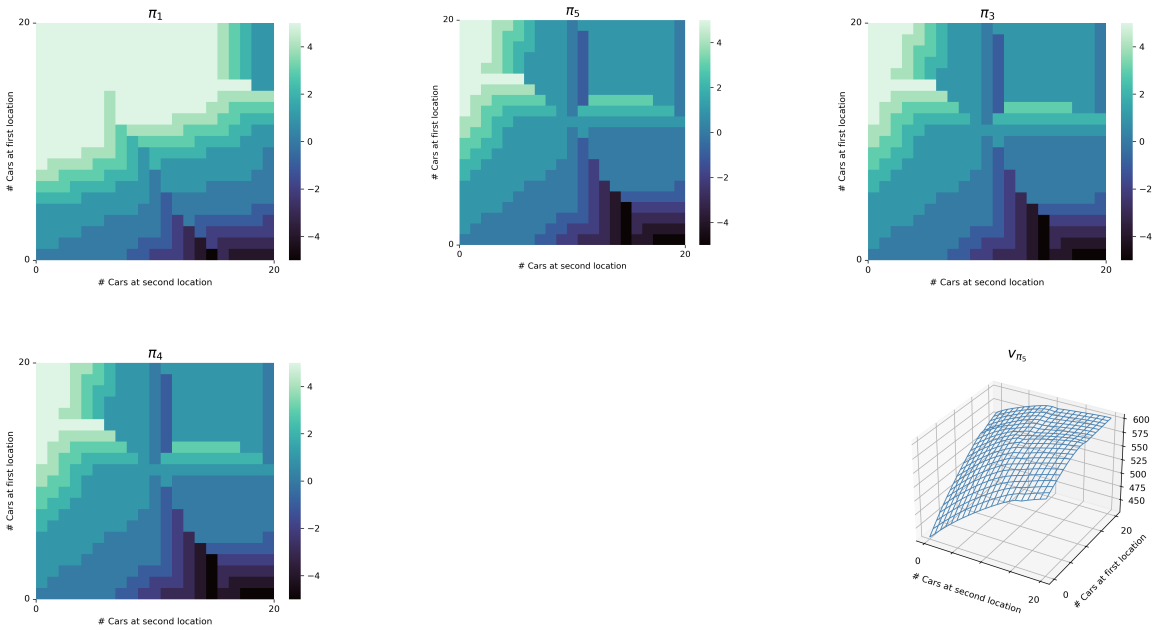
   If $policy\_stable$, then stop and return $Q \approx q_*$, and $\pi \approx \pi_*$; else go to 2

Exercise 4.7:

Write a program for policy iteration and re-solve Jack's car rental problem with the following changes. One of Jack's employees at the first location rides a bus home each night and lives near the second location. She is happy to shuttle one car to the second location for free. Each additional car still costs \$2, as do all cars moved in the other direction. In addition, Jack has limited parking space at each location. If more than 10 cars are kept overnight at a location (after any moving of cars), then an additional cost of \$4 must be incurred to use a second parking lot (independent of how many cars are kept there).

Solution:





▼ 4.4 Value Iteration

- ***Value iteration:***

> 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).

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

The value iteration can be understood as just turning the Bellman optimality equation $v_*(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')]$ into an update rule.

- The value iteration stops once the value function changes by only a small amount in a sweep.
- Algorithm

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 $V(terminal) = 0$

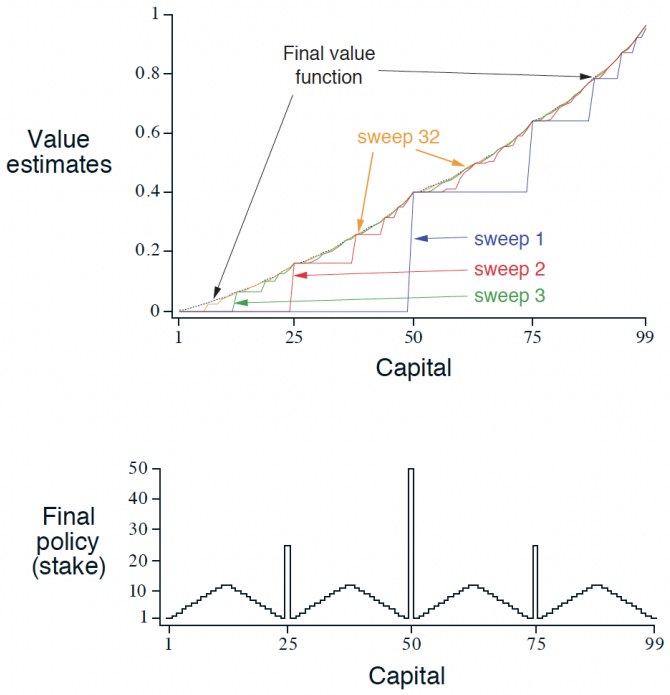
Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $v \leftarrow V(s)$
| $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$
| $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

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

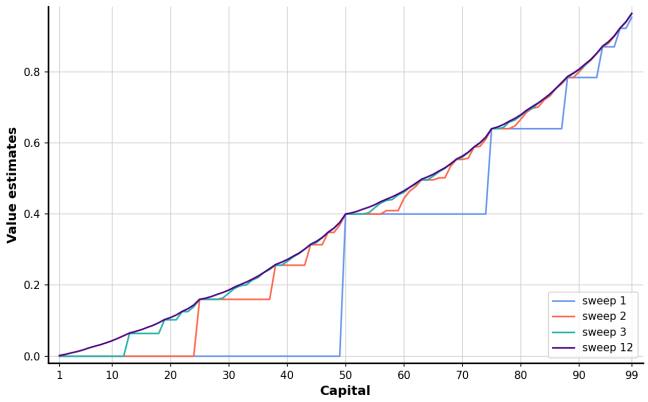
The value iteration effectively combines, in each of its sweeps, one sweep of policy evaluation and one sweep of policy improvement.

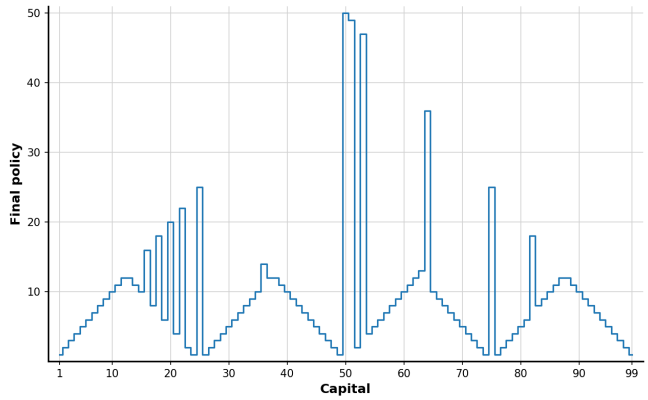
Example 4.3: Gambler's problem

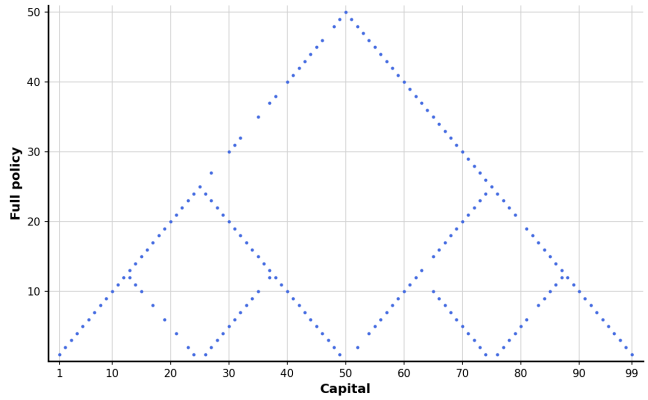
A gambler has the opportunity to make bets on the outcomes of a sequence of coin flips. If the coin comes up heads, he wins as many dollars as he has staked on that flip; if it is tails, he loses his stake. The game ends when the gambler wins by reaching his goal of $100, or loses by running out of money. On each flip, the gambler must decide what portion of his capital to stake, in integer numbers of dollars.



Implementation:







Reinforcement-Learning-2nd-Edition-Notes-Codes/chapter_04_dynamic_programming/example_4_3_gambler.py at main · terrence-ou/Reinforcemen

Notes and code implementations of examples and algorithms of the book Reinforcement Learning, 2nd Edition - Reinforcement-Learning-2nd-Edition-Notes-Codes/chapter_04_dynamic_programming/example_4_3...

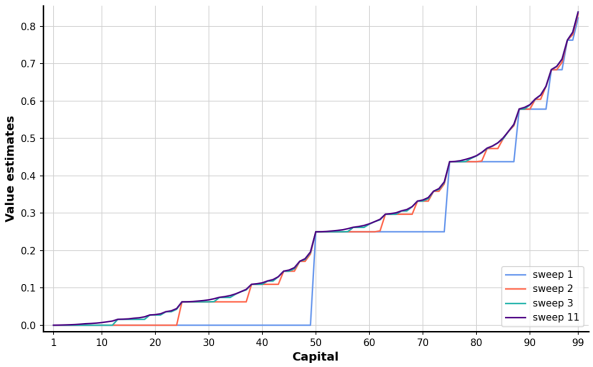
https://github.com/terrence-ou/Reinforcement-Learning-2nd-Edition-Notes-Codes/blob/main/chapter_04_dynamic_programming/example_4_3_gambler.py

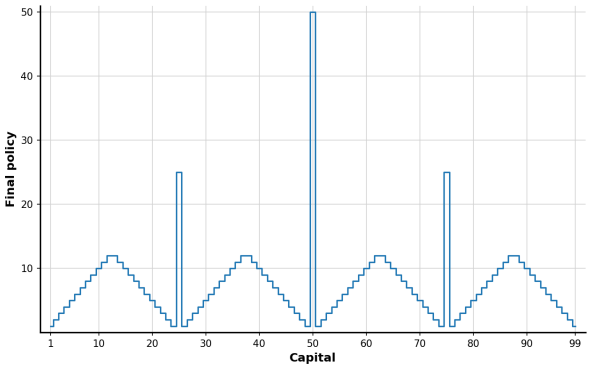
Exercise 4.9

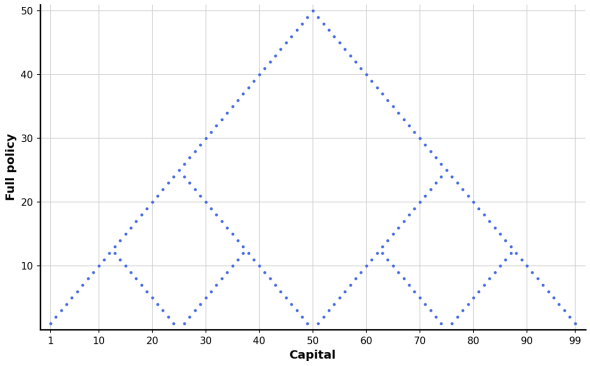
Implement value iteration for the gambler's problem and solve it for $p_h = 0.25$ and $p_h = 0.55$.

Solution:

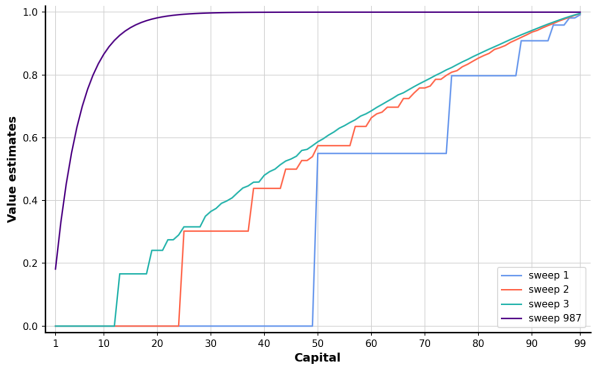
$p_h = 0.25$

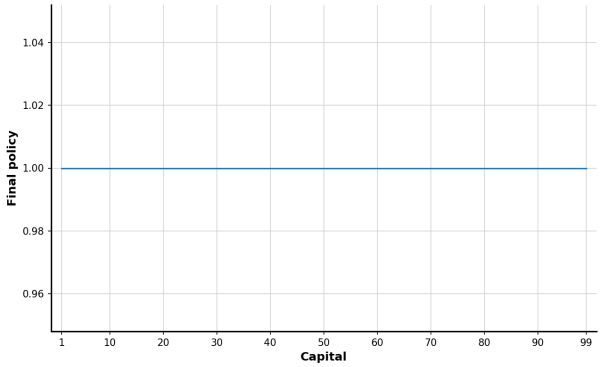






$p_h = 0.55$:





Reinforcement-Learning-2nd-Edition-Notes-Codes/chapter_04_dynamic_programming/example_4_3_gambler.py at main · terrence-ou/Reinforcemen

Notes and code implementations of examples and algorithms of the book Reinforcement Learning, 2nd Edition - Reinforcement-Learning-2nd-Edition-Notes-Codes/chapter_04_dynamic_programming/example_4_3...

https://github.com/terrence-ou/Reinforcement-Learning-2nd-Edition-Notes-Codes/blob/main/chapter_04_dynamic_programming/example_4_3_gambler.py

▼ 4.5 Asynchronous Dynamic Programming

- Limitations of PI and VI: If the state set is very large, like over 20^{10} states, then even a single sweep can be prohibitively expensive.
- ***Asynchronous DP algorithms:***

  In-place iterative DL algorithms that are not organized in terms of systematic sweeps of the state set.

  However, to converge correctly, an asynchronous algorithm must continue to update the value 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.

▼ 4.6 Generalized Policy Iteration

- ***Generalized policy iteration (GPI):***

  It refers to the general idea of letting policy-evaluation and policy-improvement processes interact; Almost all reinforcement learning methods are well described as GPI.

- Competing and cooperating:

  The evaluation and improvement processes in GPI 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.