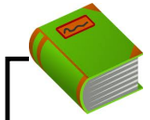
MDPs: policy evaluation

Evaluating a policy

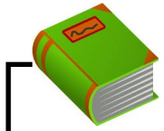


Definition: utility

Following a policy yields a **random path**.

The **utility** of a policy is the (discounted) sum of the rewards on the path (this is a random variable).

Path	Utility
[in; stay, 4, end]	4
[in; stay, 4, in; stay, 4, in; stay, 4, end]	12
[in; stay, 4, in; stay, 4, end]	8
[in; stay, 4, in; stay, 4, in; stay, 4, in; stay, 4, end]	16
...	...



Definition: value (expected utility)

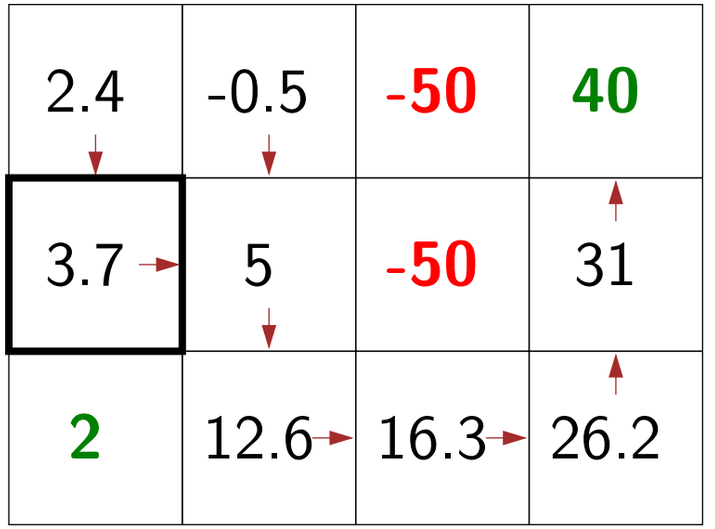
The **value** of a policy at a state is the **expected** utility.

Value: 12

- Now that we've defined an MDP (the input) and a policy (the output), let's turn to defining the evaluation metric for a policy — there are many of them, which one should we choose?
- Recall that we'd like to maximize the total rewards (utility), but this is a random variable, so we can't quite do that. Instead, we will instead maximize the **expected utility**, which we will refer to as **value** (of a policy).

Evaluating a policy: volcano crossing

Run (or press ctrl-enter)



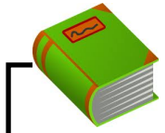
a	r	s
		(2,1)
E	-0.1	(2,1)
E	-0.1	(2,2)
S	-0.1	(3,2)
E	-0.1	(3,3)
E	-0.1	(3,4)
N	-0.1	(2,4)
N	39.9	(1,4)

Value: 3.73

Utility: 20.74

- To get an intuitive feel for the relationship between a value and utility, consider the volcano example. If you press `Run` multiple times, you will get random paths shown on the right leading to different utilities. Note that there is considerable variation in what happens.
- The expectation of this utility is the **value**.
- You can run multiple simulations by increasing `numEpisodes`. If you set `numEpisodes` to 1000, then you'll see the average utility converging to the value.

Discounting



Definition: utility

Path: $s_0, a_1 r_1 s_1, a_2 r_2 s_2, \ldots$ (action, reward, new state).

The **utility** with discount γ is

$$u_1 = r_1 + \gamma r_2 + \gamma^2 r_3 + \gamma^3 r_4 + \cdots$$

Discount $\gamma = 1$ (save for the future):

[stay, stay, stay, stay]: $4 + 4 + 4 + 4 = 16$

Discount $\gamma = 0$ (live in the moment):

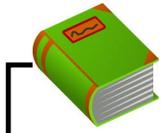
[stay, stay, stay, stay]: $4 + 0 \cdot (4 + \cdots) = 4$

Discount $\gamma = 0.5$ (balanced life):

[stay, stay, stay, stay]: $4 + \frac{1}{2} \cdot 4 + \frac{1}{4} \cdot 4 + \frac{1}{8} \cdot 4 = 7.5$

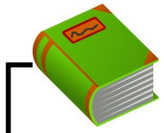
- There is an additional aspect to utility: **discounting**, which captures the fact that a reward today might be worth more than the same reward tomorrow. If the discount γ is small, then we favor the present more and downweight future rewards more.
- Note that the discounting parameter is applied exponentially to future rewards, so the distant future is always going to have a fairly small contribution to the utility (unless $\gamma = 1$).
- The terminology, though standard, is slightly confusing: a larger value of the discount parameter γ actually means that the future is discounted less.

Policy evaluation



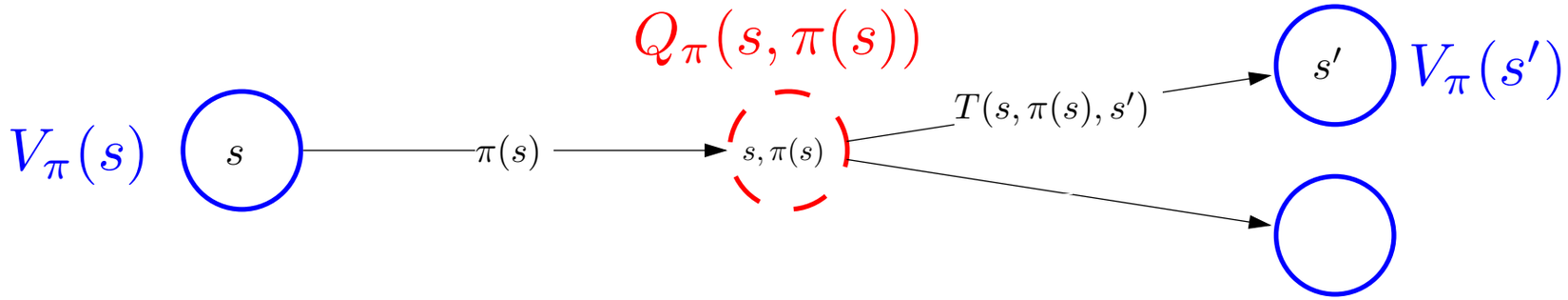
Definition: value of a policy

Let $V_\pi(s)$ be the expected utility received by following policy π from state s.



Definition: Q-value of a policy

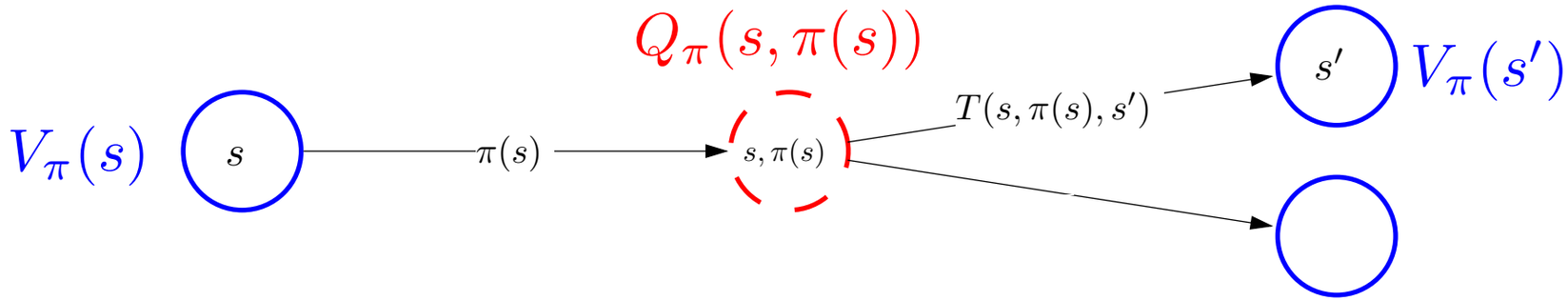
Let $Q_\pi(s, a)$ be the expected utility of taking action a from state s, and then following policy π.



- Associated with any policy π are two important quantities, the value of the policy $V_\pi(s)$ and the Q-value of a policy $Q_\pi(s, a)$.
- In terms of the MDP graph, one can think of the value $V_\pi(s)$ as labeling the state nodes, and the Q-value $Q_\pi(s, a)$ as labeling the chance nodes.
- This label refers to the expected utility if we were to start at that node and continue the dynamics of the game.

Policy evaluation

Plan: define recurrences relating value and Q-value

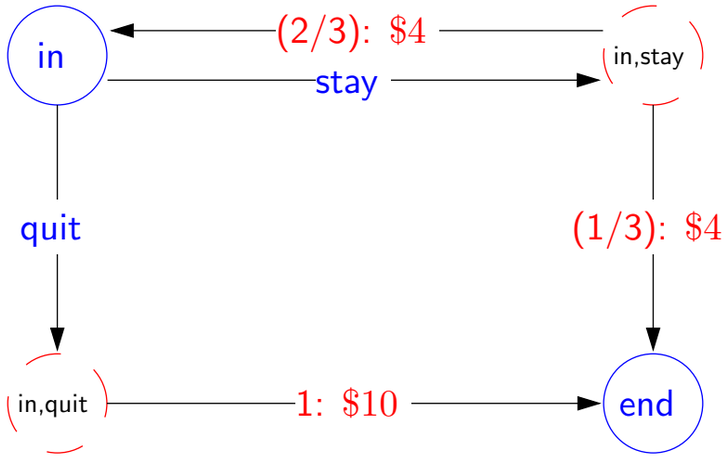


$$V_\pi(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ Q_\pi(s, \pi(s)) & \text{otherwise.} \end{cases}$$

$$Q_\pi(s, a) = \sum_{s'} T(s, a, s')[\text{Reward}(s, a, s') + \gamma V_\pi(s')]$$

- We will now write down some equations relating value and Q-value. Our eventual goal is to get to an algorithm for computing these values, but as we will see, writing down the relationships gets us most of the way there, just as writing down the recurrence for FutureCost directly lead to a dynamic programming algorithm for acyclic search problems.
- First, we get $V_\pi(s)$, the value of a state s, by just following the action edge specified by the policy and taking the Q-value $Q_\pi(s, \pi(s))$. (There's also a base case where IsEnd(s).)
- Second, we get $Q_\pi(s, a)$ by considering all possible transitions to successor states s' and taking the expectation over the immediate reward Reward(s, a, s') plus the discounted future reward $\gamma V_\pi(s')$.
- While we've defined the recurrence for the expected utility directly, we can derive the recurrence by applying the law of total expectation and invoking the Markov property. To do this, we need to set up some random variables: Let s_0 be the initial state, a_1 be the action that we take, r_1 be the reward we obtain, and s_1 be the state we end up in. Also define $u_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$ to be the utility of following policy π from time step t. Then $V_\pi(s) = \mathbb{E}[u_1 \mid s_0 = s]$, which (assuming s is not an end state) in turn equals $\sum_{s'} \mathbb{P}[s_1 = s' \mid s_0 = s, a_1 = \pi(s)]\mathbb{E}[u_1 \mid s_1 = s', s_0 = s, a_1 = \pi(s)]$. Note that $\mathbb{P}[s_1 = s' \mid s_0 = s, a_1 = \pi(s)] = T(s, \pi(s), s')$. Using the fact that $u_1 = r_1 + \gamma u_2$ and taking expectations, we get that $\mathbb{E}[u \mid s_1 = s', s_0 = s, a_1 = \pi(s)] =$ Reward$(s, \pi(s), s') + \gamma V_\pi(s')$. The rest follows from algebra.

Dice game



(assume $\gamma = 1$)

Let π be the "stay" policy: $\pi(\text{in}) = \text{stay}$.

$$V_\pi(\text{end}) = 0$$

$$V_\pi(\text{in}) = \frac{1}{3}(4 + V_\pi(\text{end})) + \frac{2}{3}(4 + V_\pi(\text{in}))$$

In this case, can solve in closed form:

$$V_\pi(\text{in}) = 12$$

- As an example, let's compute the values of the nodes in the dice game for the policy "stay".
- Note that the recurrence involves both $V_\pi(\text{in})$ on the left-hand side and the right-hand side. At least in this simple example, we can solve this recurrence easily to get the value.

Policy evaluation

Key idea: iterative algorithm

Start with arbitrary policy values and repeatedly apply recurrences to converge to true values.

Algorithm: policy evaluation

Initialize $V_\pi^{(0)}(s) \leftarrow 0$ for all states s.

For iteration $t = 1, \ldots, t_{\text{PE}}$:

For each state s:

$$V_\pi^{(t)}(s) \leftarrow \underbrace{\sum_{s'} T(s, \pi(s), s')[\text{Reward}(s, \pi(s), s') + \gamma V_\pi^{(t-1)}(s')]}_{Q^{(t-1)}(s, \pi(s))}$$

- But for a much larger MDP with 100000 states, how do we efficiently compute the value of a policy?
- One option is the following: observe that the recurrences define a system of linear equations, where the variables are $V_\pi(s)$ for each state s and there is an equation for each state. So we could solve the system of linear equations by computing a matrix inverse. However, inverting a 100000×100000 matrix is expensive in general.
- There is an even simpler approach called **policy evaluation**. We've already seen examples of iterative algorithms in machine learning: the basic idea is to start with something crude, and refine it over time.
- Policy iteration starts with a vector of all zeros for the initial values $V_\pi^{(0)}$. Each iteration, we loop over all the states and apply the two recurrences that we had before. The equations look hairier because of the superscript (t), which simply denotes the value of at iteration t of the algorithm.

Policy evaluation implementation

How many iterations (t_{PE})? Repeat until values don't change much:

$$\max_{s \in \mathsf{States}} |V_\pi^{(t)}(s) - V_\pi^{(t-1)}(s)| \leq \epsilon$$

Don't store $V_\pi^{(t)}$ for each iteration t, need only last two:

$$V_\pi^{(t)} \text{ and } V_\pi^{(t-1)}$$

- Some implementation notes: a good strategy for determining how many iterations to run policy evaluation is based on how accurate the result is. Rather than set some fixed number of iterations (e.g, 100), we instead set an error tolerance (e.g., $\epsilon = 0.01$), and iterate until the maximum change between values of any state s from one iteration (t) to the previous $(t-1)$ is at most ϵ.
- The second note is that while the algorithm is stated as computing $V_\pi^{(t)}$ for each iteration t, we actually only need to keep track of the last two values. This is important for saving memory.

Complexity

Algorithm: policy evaluation

Initialize $V_\pi^{(0)}(s) \leftarrow 0$ for all states s.

For iteration $t = 1, \ldots, t_{\text{PE}}$:

For each state s:

$$V_\pi^{(t)}(s) \leftarrow \underbrace{\sum_{s'} T(s, \pi(s), s')[\text{Reward}(s, \pi(s), s') + \gamma V_\pi^{(t-1)}(s')]}_{Q^{(t-1)}(s, \pi(s))}$$

MDP complexity

S states

A actions per state

S' successors (number of s' with $T(s, a, s') > 0$)

Time: $O(t_{\text{PE}} S S')$

- Computing the running time of policy evaluation is straightforward: for each of the t_{PE} iterations, we need to enumerate through each of the S states, and for each one of those, loop over the successors S'. Note that we don't have a dependence on the number of actions A because we have a fixed policy $\pi(s)$ and we only need to look at the action specified by the policy.
- Advanced: Here, we have to iterate t_{PE} time steps to reach a target level of error ϵ. It turns out that t_{PE} doesn't actually have to be very large for very small errors. Specifically, the error decreases exponentially fast as we increase the number of iterations. In other words, to cut the error in half, we only have to run a constant number of more iterations.
- Advanced: For acyclic graphs (for example, the MDP for Blackjack), we just need to do one iteration (not t_{PE}) provided that we process the nodes in reverse topological order of the graph. This is the same setup as we had for dynamic programming in search problems, only the equations are different.

Policy evaluation on dice game

Let π be the "stay" policy: $\pi(\text{in}) = \text{stay}$.

$$V_\pi^{(t)}(\text{end}) = 0$$

$$V_\pi^{(t)}(\text{in}) = \tfrac{1}{3}(4 + V_\pi^{(t-1)}(\text{end})) + \tfrac{2}{3}(4 + V_\pi^{(t-1)}(\text{in}))$$

s	end	in	
$V_\pi^{(t)}$	0.00	12.00	($t = 100$ iterations)

Converges to $V_\pi(\text{in}) = 12$.

- Let us run policy evaluation on the dice game. The value converges very quickly to the correct answer.

Summary so far

- MDP: graph with states, chance nodes, transition probabilities, rewards
- Policy: mapping from state to action (solution to MDP)
- Value of policy: expected utility over random paths
- Policy evaluation: iterative algorithm to compute value of policy

- Let's summarize: we have defined an MDP, which we should think of a graph where the nodes are states and chance nodes. Because of randomness, solving an MDP means generating policies, not just paths. A policy is evaluated based on its value: the expected utility obtained over random paths. Finally, we saw that policy evaluation provides a simple way to compute the value of a policy.