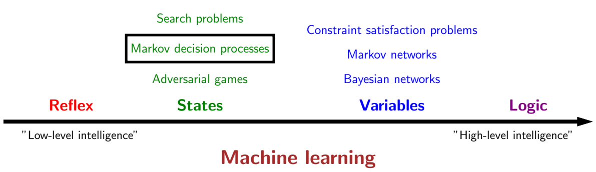
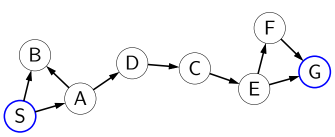
MDPs: overview

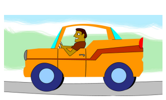
Course plan



So far: search problems



state s, action a $\xrightarrow{\textbf{deterministic}}$ state Succ(s, a)



- Last week, we looked at search problems, a powerful paradigm that can be used to solve a diverse range of problems ranging from word segmentation to package delivery to route finding. The key was to cast whatever problem we were interested in solving into the problem of finding the minimum cost path in a graph.
- However, search problems assume that taking an action a from a state s results **deterministically** in a unique successor state Succ(s, a).

Uncertainty in the real world

state s, action a $\xrightarrow{\text{random}}$ state s'_1

state s, action a $\xrightarrow{\text{random}}$ state s'_2

- In the real world, the deterministic successor assumption is often unrealistic, for there is **randomness**: taking an action might lead to any one of many possible states.
- One deep question here is how we can even hope to act optimally in the face of randomness? Certainly we can't just have a single deterministic plan, and talking about a minimum cost path doesn't make sense.
- Today, we will develop tools to tackle this more challenging setting. We will fortunately still be able to reuse many of the intuitions about search problems, in particular the notion of a state.

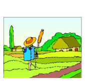
History

- MDPs: Mathematical Model for decision making under uncertainty.
- MDPs were first introduced in 1950s-60s.
- Ronald Howard's book on Dynamic Programming and Markov Processes
- The term 'Markov' refers to Andrey Markov as MDPs are extensions of Markov Chains, and they allow making decisions (taking actions or having choice).

Applications

Robotics: decide where to move, but actuators can fail, hit unseen obstacles, etc.

Resource allocation: decide what to produce, don't know the customer demand for various products



Agriculture: decide what to plant, but don't know weather and thus crop yield

- Randomness shows up in many places. They could be caused by limitations of the sensors and actuators of the robot (which we can control to some extent). Or they could be caused by market forces or nature, which we have no control over.
- We'll see that all of these sources of randomness can be handled in the same mathematical framework.

Volcano crossing



Run (or press ctrl-enter)

		-50	20
		-50	
2			

- Let us consider an example. You are exploring a South Pacific island, which is modeled as a 3x4 grid of states. From each state, you can take one of four actions to move to an adjacent state: north (N), east (E), south (S), or west (W). If you try to move off the grid, you remain in the same state. You start at (2,1). If you end up in either of the green or red squares, your journey ends, either in a lava lake (reward of -50) or in a safe area with either no view (2) or a fabulous view of the island (20). What do you do?
- If we have a deterministic search problem, then the obvious thing will be to go for the fabulous view, which yields a reward of 20. You can set `numIters` to 10 and press `Run`. Each state is labeled with the maximum expected utility (sum of rewards) one can get from that state (analogue of FutureCost in a search problem). We will define this quantity formally later. For now, look at the arrows, which represent the best action to take from each cell. Note that in some cases, there is a tie for the best, where some of the actions seem to be moving in the wrong direction. This is because there is no penalty for moving around indefinitely. If you change `moveReward` to -0.1, then you'll see the arrows point in the right direction.
- In reality, we are dealing with treacherous terrain, and there is on each action a probability `slipProb` of slipping, which results in moving in a random direction. Try setting `slipProb` to various values. For small values (e.g., 0.1), the optimal action is to still go for the fabulous view. For large values (e.g., 0.3), then it's better to go for the safe and boring 2. Play around with the other reward values to get intuition for the problem.
- Important: note that we are only specifying the dynamics of the world, not directly specifying the best action to take. The best actions are computed automatically from the algorithms we'll see shortly.

Roadmap

Modeling

Modeling MDP Problems

Algorithms

Policy Evaluation

Value Iteration

Learning

Intro to Reinforcement Learning

Model-Based Monte Carlo

Model-Free Monte Carlo

SARSA

Q-learning

Epsilon Greedy

Function Approximation

Dice game

Example: dice game

For each round $r = 1, 2, \ldots$

- You choose stay or quit.
- If quit, you get $\$10$ and we end the game.
- If stay, you get $\$4$ and then I roll a 6-sided dice.
  - If the dice results in 1 or 2, we end the game.
  - Otherwise, continue to the next round.

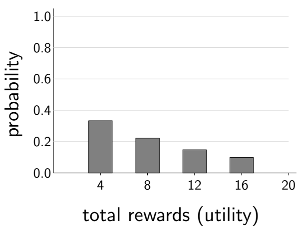
Start Stay Quit

Dice: Rewards: 0

- We'll see more volcanoes later, but let's start with a much simpler example: a dice game. What is the best strategy for this game?

Rewards

If follow policy "stay":



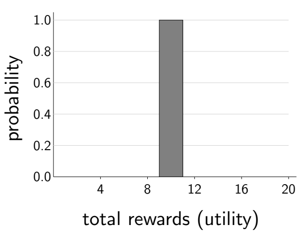
Expected utility:

$$\frac{1}{3}(4) + \frac{2}{3} \cdot \frac{1}{3}(8) + \frac{2}{3} \cdot \frac{2}{3} \cdot \frac{1}{3}(12) + \cdots = 12$$

- Let's suppose you always stay. Note that each outcome of the game will result in a different sequence of rewards, resulting in a **utility**, which is in this case just the sum of the rewards.
- We are interested in the **expected** utility, which you can compute to be 12.

Rewards

If follow policy "quit":



Expected utility:

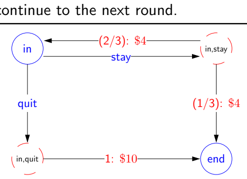
$$1(10) = 10$$

- If you quit, then you'll get a reward of 10 deterministically. Therefore, in expectation, the "stay" strategy is preferred, even though sometimes you'll get less than 10.

MDP for dice game

Example: dice game

For each round $r = 1, 2, \ldots$
- You choose stay or quit.
- If quit, you get $\$10$ and we end the game.
- If stay, you get $\$4$ and then I roll a 6-sided dice.
  - If the dice results in 1 or 2, we end the game.
  - Otherwise, continue to the next round.



- While we already solved this game directly, we'd like to develop a more general framework for thinking about not just this game, but also other problems such as the volcano crossing example. To that end, let us formalize the dice game as a **Markov decision process** (MDP).
- An MDP can be represented as a graph. The nodes in this graph include both **states** and **chance nodes**. Edges coming out of states are the possible actions from that state, which lead to chance nodes. Edges coming out of a chance nodes are the possible random outcomes of that action, which end up back in states. Our convention is to label these chance-to-state edges with the probability of a particular **transition** and the associated reward for traversing that edge.

Markov decision process

Definition: Markov decision process

States: the set of states

$s_{\text{start}} \in$ States: starting state

Actions(s): possible actions from state s

$T(s, a, s')$: probability of s' if take action a in state s

Reward(s, a, s'): reward for the transition (s, a, s')

IsEnd(s): whether at end of game

$0 \leq \gamma \leq 1$: discount factor (default: 1)

- A **Markov decision process** has a set of states States, a starting state s_{start}, and the set of actions Actions(s) from each state s.
- It also has a **transition distribution** T, which specifies for each state s and action a, a distribution over possible successor states s'. Specifically, we have that $\sum_{s'} T(s, a, s') = 1$ because T is a probability distribution (more on this later).
- Associated with each transition (s, a, s') is a reward, which could be either positive or negative.
- If we arrive in a state s for which IsEnd(s) is true, then the game is over.
- Finally, the discount factor γ is a quantity which specifies how much we value the future and will be discussed later.

Search problems

Definition: search problem

States: the set of states

$s_{\text{start}} \in$ States: starting state

Actions(s): possible actions from state s

Succ(s, a): where we end up if take action a in state s

Cost(s, a): cost for taking action a in state s

IsEnd(s): whether at end

- Succ$(s, a) \Rightarrow T(s, a, s')$
- Cost$(s, a) \Rightarrow$ Reward(s, a, s')

- MDPs share many similarities with search problems, but there are differences (one main difference and one minor one).
- The main difference is the move from a deterministic successor function Succ(s, a) to transition probabilities over s'. We can think of the successor function Succ(s, a) as a special case of transition probabilities: $T(s, a, s') = \begin{cases} 1 & \text{if } s' = \text{Succ}(s, a) \\ 0 & \text{otherwise} \end{cases}$.
- A minor difference is that we've gone from minimizing costs to maximizing rewards. The two are really equivalent: you can negate one to get the other.

Transitions

Definition: transition probabilities

The **transition probabilities** $T(s, a, s')$ specify the probability of ending up in state s' when taking action a in state s.

Example: transition probabilities

s	a	s'	$T(s, a, s')$
in	quit	end	1
in	stay	in	$2/3$
in	stay	end	$1/3$

- Just to dwell on the major difference, transition probabilities, a bit more: for each state s and action a, the transition probabilities specifies a distribution over successor states s'.

Probabilities sum to one

Example: transition probabilities

s	a	s'	$T(s,a,s')$
in	quit	end	1
in	stay	in	$2/3$
in	stay	end	$1/3$

For each state s and action a:

$$\sum_{s' \in \text{States}} T(s,a,s') = 1$$

Successors: s' such that $T(s,a,s') > 0$

- This means that for each given s and a, if we sum the transition probability $T(s,a,s')$ over all possible successor states s', we get 1.
- If a transition to a particular s' is not possible, then $T(s,a,s') = 0$. We refer to the s' for which $T(s,a,s') > 0$ as the successors.
- Generally, the number of successors of a given (s,a) is much smaller than the total number of states. For instance, in a search problem, each (s,a) has exactly one successor.

What is a solution?

Search problem: path (sequence of actions)

MDP:

Definition: policy

A **policy** π is a mapping from each state $s \in$ States to an action $a \in$ Actions(s).

Example: volcano crossing

s	$\pi(s)$
(1,1)	S
(2,1)	E
(3,1)	N
...	...

- So we now know what an MDP is. What do we do with one? For search problems, we were trying to find the minimum cost **path**.
- However, fixed paths won't suffice for MDPs, because we don't know which states the random dice rolls are going to take us.
- Therefore, we define a **policy**, which specifies an action for every single state, not just the states along a path. This way, we have all our bases covered, and know what action to take no matter where we are.
- One might wonder if we ever need to take different actions from a given state. The answer is no, since like as in a search problem, the state contains all the information that we need to act optimally for the future. In more formal speak, the transitions and rewards satisfy the **Markov property**. Every time we end up in a state, we are faced with the exact same problem and therefore should take the same optimal action.

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, end]	12
[in; stay, 4, in; stay, 4, end]	8
[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)

2.4	-0.5	**-50**	**40**
3.7	5	**-50**	31
2	12.6	16.3	26.2

a	r	s
		(2,1)
E	-0.1	(2,2)
S	-0.1	(3,2)
E	-0.1	(3,3)
E	-50.1	(2,3)

Value: 3.73

Utility: -36.79

- 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: $s_0, a_1 r_1 s_1, a_2 r_2 s_2, \dots$ (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.

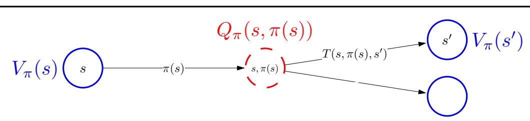
Policy evaluation

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

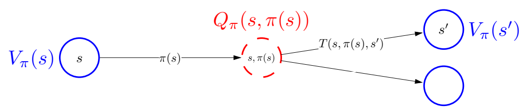
Let $Q_\pi(s, a)$ be the expected utility of taking action a from state s, and then following policy π.



- 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

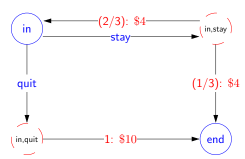


$$V_\pi(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ Q_\pi(s,\pi(s)) & \text{otherwise.} \end{cases}$$

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

- 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 $\mathbb{P}[s_1 = s' \mid s_0 = s, a_1 = \pi(s)] = T(s'|s,\pi(s))$. Using the fact that $u_1 = r_1 + \gamma u_2$ and taking expectations, we get that $\mathbb{E}[u \mid s_1 = s', s_0 = s, a_1 = \pi(s)] = \text{Reward}(s,\pi(s),s') + \gamma V_\pi(s')$. The rest follows from algebra.

Dice game



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:

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

- 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:

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

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 $t = 1, \dots, t_{\text{PE}}$:
For each state s:

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

MDP complexity

S states
A actions per state
S' successors (number of s' with $T(s' \mid s, a) > 0$)

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: $\pi(\text{in}) = \text{stay}$.

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

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

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.

Optimal value and policy

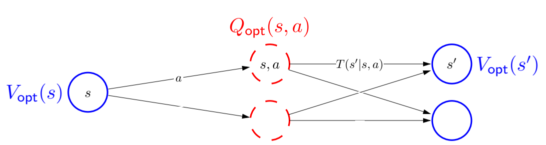
Goal: try to get directly at maximum expected utility

Definition: optimal value

The **optimal value** $V_{\text{opt}}(s)$ is the maximum value attained by any policy.

- We will write down a bunch of recurrences which look exactly like policy evaluation, but instead of having V_π and Q_π with respect to a fixed policy π, we will have V_{opt} and Q_{opt}, which are with respect to the optimal policy.

Optimal values and Q-values



Optimal value if take action a in state s:

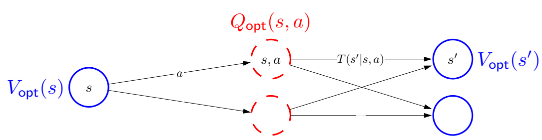
$$Q_{\text{opt}}(s,a) = \sum_{s'} T(s,a,s')[\text{Reward}(s,a,s') + \gamma V_{\text{opt}}(s')].$$

Optimal value from state s:

$$V_{\text{opt}}(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} Q_{\text{opt}}(s,a) & \text{otherwise.} \end{cases}$$

- The recurrences for V_{opt} and Q_{opt} are identical to the ones for policy evaluation with one difference: in computing V_{opt}, instead of taking the action from the fixed policy π, we take the best action, the one that results in the largest $Q_{\text{opt}}(s,a)$.

Optimal policies



Given Q_{opt}, read off the optimal policy:

$$\pi_{\text{opt}}(s) = \arg \max_{a \in \text{Actions}(s)} Q_{\text{opt}}(s,a)$$

- So far, we have focused on computing the value of the optimal policy, but what is the actual policy? It turns out that this is pretty easy to compute.
- Suppose you're at a state s. $Q_{\text{opt}}(s,a)$ tells you the value of taking action a from state s. So the optimal action is simply to take the action a with the largest value of $Q_{\text{opt}}(s,a)$.

Value iteration

Algorithm: value iteration [Bellman, 1957]

Initialize $V_{\text{opt}}^{(0)}(s) \leftarrow 0$ for all states s.
For iteration $t = 1, \ldots, t_{\text{VI}}$:
  For each state s:
  $$V_{\text{opt}}^{(t)}(s) \leftarrow \max_{a \in \text{Actions}(s)} \underbrace{\sum_{s'} T(s,a,s')[\text{Reward}(s,a,s') + \gamma V_{\text{opt}}^{(t-1)}(s')]}_{Q_{\text{opt}}^{(t-1)}(s,a)}$$

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

- By now, you should be able to go from recurrences to algorithms easily. Following the recipe, we simply iterate some number of iterations, go through each state s and then replace the equality in the recurrence with the assignment operator.
- Value iteration is also guaranteed to converge to the optimal value.
- What about the optimal policy? We get it as a byproduct. The optimal value $V_{\text{opt}}(s)$ is computed by taking a max over actions. If we take the argmax, then we get the optimal policy $\pi_{\text{opt}}(s)$.

Value iteration: dice game

s	end	in
$V_{\text{opt}}^{(t)}$	0.00	12.00 ($t = 100$ iterations)
$\pi_{\text{opt}}(s)$	-	stay

- Let us demonstrate value iteration on the dice game. Initially, the optimal policy is "quit", but as we run value iteration longer, it switches to "stay".

Value iteration: volcano crossing

Run (or press ctrl-enter)

		-50	20
		-50	
2			

- As another example, consider the volcano crossing. Initially, the optimal policy and value correspond to going to the safe and boring 2. But as you increase `numIters`, notice how the value of the far away 20 propagates across the grid to the starting point.
- To see this propagation even more clearly, set `slipProb` to 0.

Convergence

Theorem: convergence

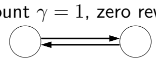
Suppose either

- discount $\gamma < 1$, or
- MDP graph is acyclic.

Then value iteration converges to the correct answer.

Example: non-convergence

discount $\gamma = 1$, zero rewards



- Let us state more formally the conditions under which any of these algorithms that we talked about will work. A sufficient condition is that either the discount γ must be strictly less than 1 or the MDP graph is acyclic.
- We can reinterpret the discount $\gamma < 1$ condition as introducing a new transition from each state to a special end state with probability $(1-\gamma)$, multiplying all the other transition probabilities by γ, and setting the discount to 1. The interpretation is that with probability $1-\gamma$, the MDP terminates at any state.
- In this view, we just need that a sampled path be finite with probability 1.
- We won't prove this theorem, but will instead give a counterexample to show that things can go badly if we have a cyclic graph and $\gamma = 1$. In the graph, whatever we initialize value iteration, value iteration will terminate immediately with the same value. In some sense, this isn't really the fault of value iteration, but it's because all paths are of infinite length. In some sense, if you were to simulate from this MDP, you would never terminate, so we would never find out what your utility was at the end.

Summary of algorithms

- Policy evaluation: (MDP, π) $\rightarrow V_\pi$
- Value iteration: MDP $\rightarrow (Q_{\text{opt}}, \pi_{\text{opt}})$

Unifying idea

Algorithms:

- Search DP computes $\text{FutureCost}(s)$
- Policy evaluation computes policy value $V_\pi(s)$
- Value iteration computes optimal value $V_{\text{opt}}(s)$

Recipe:

- Write down recurrence (e.g., $V_\pi(s) = \cdots V_\pi(s') \cdots$)
- Turn into iterative algorithm (replace mathematical equality with assignment operator)

- There are two key ideas in this lecture. First, the policy π, value V_π, and Q-value Q_π are the three key quantities of MDPs, and they are related via a number of recurrences which can be easily gotten by just thinking about their interpretations.
- Second, given recurrences that depend on each other for the values you're trying to compute, it's easy to turn these recurrences into algorithms that iterate between those recurrences until convergence.