1 MDP Overview

1.1 Intuition

There are many problems that can be modeled where an action, rather than producing a deterministic result, produces a distribution over possible outcomes. Consider for example a self driving car. The car is planning on merging into a busy interstate. After using the turn signal, perhaps there is probability P that other cars slow down and let it merge in. If $P = 1$, and cars always let them in, there is no need for a distribution. Generally, however $P \neq 1$. MDP's provide us with a way of approaching these problems.

MDP's again take advantage of the Markov Assumption: *All future states are independent of previous states given the current state.* MDP's provide a framework for modeling decision making where outcomes are partly random, and partly under the control of the decision maker.

1.2 Necessary Components

States: A set of states $\{s_1, s_2, ..., s_n\}$ in S.

Actions: A set of actions available from the states $\{a_1, a_2, ..., a_m\}$ in A.

Transition Function: $T(s, a, s')$ tells the probability of ending in state s' after taking action a from state s. Can be thought of like:

$$P(s'|a, s)$$

Reward Function: $R(s, a)$ the reward of taking action a from state s. In simple cases, we may only care about the state we started in, $R(s)$.

Start State: Because MDP must know what state they are in for their policy function, a start state must be provided.

Terminal State (optional): Some problems may terminate once a certain state is reached, though this is not always the case.

1.3 Output

Generally, the point of solving an MDP is to arrive at a policy that maximizes the expected sum of rewards. More formally, a policy is a mapping from states to actions. We will refer to this policy as π. Giving your policy a state should result in an action. For example

$$\pi(s_1) = a_4$$

2 Policy Determination

2.1 Intuition

Your first thought when trying to come up with a way to solve for a policy may be to use an expecti-max search tree. Model it such that every node is a state, and depending on your action, you arrive at different successors with probabilities defined in the transition function. Why might this be a bad idea?

Well, if you draw it out, you will see that your search tree is usually infinitely deep (assuming there is a cycle reachable from the start state), and many of identical sub-trees are repeated over and over again. This seems like a good candidate for dynamic programming.

2.2 Discounting

Discounting is the idea that we value future rewards less than current rewards. For example, would you rather have 100 dollars right now, or 110 in a month? Most people would say 100 right now. For mathematical convenience, and due to its presence in human decision making, let's add discounting. Rewards at the current state will be multiplied by factor 1, at next state by factor γ, at state in two steps by γ^2 and so on. This can be thought of as γ^d where d is the depth from the current node in the expectimax tree. In the general case γ is slightly less than 1.

2.3 Value Iteration

2.3.1 Value Update

The general equation for determining the value with $k+1$ steps left given the you already know values with k steps left is:

$$V_{k+1}(s) = max_a \left(\sum_{s'} T(s,a,s')[R(s,a) + \gamma V_k(s')] \right)$$

2.3.2 Intuition

I will not go back through the derivation because it is presented in the slides, but I will give some intuition. At every step we want to pick the expectation maximizing action (hence the max). Determining the expected value of every action, a weighted sum, is going to depend on all places we could end up given we take that action. As always with expectation, we take the weighted sum of the probability distribution times that distributions respective value. The values in this case are the current reward and future discounted rewards.

2.3.3 Algorithm

Initialize the value (reward) with 0 steps left to 0 for all states. Then calculate with 1 step left based on 0, with 2 steps left based on 1, etc, working backwards until convergence of values. Once these values have converged, finding policy is easy. Your policy becomes picking the highest expected value action based on converged values at every state.

2.4 Policy Iteration

2.4.1 Policy Update

We will break policy iteration into the following steps:

1. **Policy Evaluation**: calculate utilities for some fixed policy until convergence
2. **Policy Improvement**: update policy using one step look ahead with converged utilities from Policy Evaluation

For **Policy Evaluation**, you can either iterate to a fixed point, or solve a system of linear equations. Let's say you want to do the iterate method. I will denote $V_{k+1}^{\pi_i}(s)$ to be the value of state s using policy π_i after $k+1$ iterations. Using this, and since we already know our policy, we can state:

$$V_{k+1}^{\pi_i}(s) = \sum_{s'} T(s, \pi_i(s), s')[R(s, \pi_i(s)) + \gamma V_k^{\pi_i}(s')]$$

Like before, we can initialize to zero and repeat until convergence. If we want to get rid of the iterations, we get the following:

$$V^{\pi_i}(s) = \sum_{s'} T(s, \pi_i(s), s')[R(s, \pi_i(s)) + \gamma V^{\pi_i}(s')]$$

Note, each of these makes a linear equation. Only the values are unknown. Assuming n states, we now have n linear equations with n unknowns. We can solve this system of linear equations to arrive at values without iterating.

Once we know the values, we want to do **Policy Improvement**. For reference, policy improvement is the act of picking the best policy using the values we calculated in the previous step. In other words, we are calculating π_{i+1} using π_i, hence the name– Policy Iteration. The improvement step is as follows:

$$\pi_{i+1}(s) = \arg\max_a \sum_{s'} T(s, a, s')[R(s, a) + \gamma V^{\pi_i}(s')]$$

Note, this is an arg max not a max, meaning we care about the action that maximized the value, not the value that was maximized. Once the mapping from states to actions stays the same over an iteration, meaning $\pi_i = \pi_{i+1}$, we are done.

2.4.2 Intuition

In most cases, we simply want a policy. While iterating to a fixed point on values can help us solve for this, we may actually arrive at our optimal policy before the values converge. Policy iteration realizes this, and provides a framework for arriving at the optimal policy without needing to wait for values to converge. Only the policy must converge.

In summary: You start by defining a policy. Then, using that policy you derive values of every state. Now that state values have changed, you update your policy. Repeat this until your policy stays the same throughout the iteration. Once this happens, you know you have arrived at the optimal policy.

3 Example

3.1 Problem Statement

You are hired as a user experience analyst for a popular website provider. Prior to your arrival, the software team had been working on a help dialogue box that it wants to start integrating with its website. The team was having trouble figuring out when they should display this dialogue, and they need your help.

Being a graduate of CS270, you decide to use MDP's. You have user test subjects come in for a study. Assume they wear a headpiece which tells you whether they are Happy, Confused, or Annoyed. You are relaying information to the software team during the study telling them when to display and hide the help box, and you are asking your test subjects to provide feedback on their experience, positive or negative at discrete times.

Throughout this experiment, you collect a bunch of data. Now, you are ready to formulate your MDP and determine the optimal policy.

3.2 MDP Definition

There are three **states** a user can be in: Happy, Annoyed, Confused.

There are two **actions** you can take from each of those states: Launch Help Popup, or Dont Launch.

The **transition function** is as follows:

T(happy, dont launch, happy) = 0.8
T(happy, dont launch, confused) = 0.2
T(happy, help popup, annoyed) = 0.6
T(happy, help popup, happy) = 0.4

T(confused, dont launch, happy) = 0.1
T(confused, dont launch, confused) = 0.9
T(confused, help popup, annoyed) = 0.2
T(confused, help popup, happy) = 0.8

T(annoyed, dont launch, annoyed) = 0.1
T(annoyed, dont launch, confused) = 0.9
T(annoyed, help popup, annoyed) = 1.0

Lastly, the reward will simply be the reward for the state they started in (s):

R(Happy) = 5, R(Confused) = -1, R(Annoyed) = -3

3.3 Solving

For the proposes of demonstration, I will only go through one iteration of these algorithms. The ideas can of course be extended to many iterations until convergence is met. Additionally, assume $\gamma = 0.9$.

3.3.1 Value Iteration

Assume $V_0(s)$ is 0 for all states.

$V_1(Happy)$ = max(
  dont launch: 0.8(5 + 0.9(0)) + 0.2(5 + 0.9(0)) = 5
  popup: 0.6(5 + 0.9(0)) + 0.4(5 + 0.9(0)) = 5
) = 5

$V_1(Conf)$ = max(
  dont launch: 0.1(-1 + 0.9(0)) + 0.9(-1 + 0.9(0)) = -1
  popup: 0.8(-1 + 0.9(0)) + 0.2(-1 + 0.9(0)) = -1
) = -1

$V_1(Ann)$ = max(
  dont launch: 0.1(-3 + 0.9(0)) + 0.9(-3 + 0.9(0)) = -3
  popup: 1(-3 + 0.9(0)) = -3
) = -3

Notice: the values for every state s after 1 step are the same as $R(s)$ no matter what action you take. Why? (Because $V_0(s) = 0$). Will this also happen in the next step? (No). Try walking through one more step of value iteration to see for yourself.

3.3.2 Policy Iteration

Assume the initial policy is always don't launch. Here are the linear equations we get:

V(Happy) = 0.8[5 + 0.9V(Happy)] + 0.2[5 + 0.9V(Confused)]
V(Confused) = 0.1[-1 + 0.9V(Happy)] + 0.9[-1 + 0.9V(Confused)]
V(Annoyed) = 0.1[-3 + 0.9V(Annoyed)] + 0.9[-3 + 0.9V(Confused)]

After solving the system of linear equations, we get: V(Happy) = 20.81, V(Confused) = 4.59, V(Annoyed) = 0.79

Now lets run policy improvement:

$V_1(Happy)$ = arg max(
  dont launch: 0.8(5 + 0.9 * 20.81) + 0.2(5 + 0.9 * 4.59) = 20.8
  popup: 0.6(5 + 0.9 * 0.79) + 0.4(5 + 0.9 * 20.81) = 12.9
) = dont launch

$V_1(Conf)$ = arg max(
  dont launch: 0.1(-1 + 0.9 * 20.81) + 0.9(-1 + 0.9 * 4.59) = 4.6
  popup: 0.8(-1 + 0.9 * 20.81) + 0.2(-1 + 0.9 * 0.79) = 14.1
) = popup

$V_1(Ann)$ = arg max(
  don't launch: 0.1(-3 + 0.9 * 0.79) + 0.9(-3 + 0.9 * 4.59) = 0.79
  popup: 1(-3 + 0.9 * 0.79) = -2.2
) = don't launch

Our new policy π_1 is:

$\pi_1(Happy)$ = dont launch
$\pi_1(Conf)$ = popup
$\pi_1(Ann)$ = dont launch

Note, the policy changed so we must run the iteration again, until convergence.

3.4 Extensions

3.4.1 Unknown state

What if we didn't know what state the user was in? Formulate this as a POMDP

3.4.2 Discount Factor

Consider an MDP where one of the next states has high reward, but all states after that one are horrible (much worse than previous was good)! How might

our choice of discount factor be important here? What do large/small discount factor prevent/allow?