Markov Decision Processes in Artificial Intelligence Solution

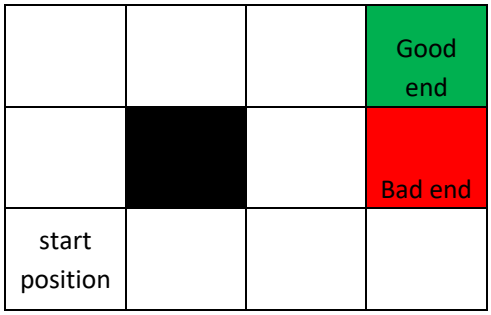
Author: Qi Li

I. Problems and Markov Decision Processes

1. Pac-Man game

This is a Pac-Man gain where the start position is bottom left and needs to find its shortest way to the green cell where the game ends. The Black cell is a wall and blocks the way, the red cell is a trap where the game ends bad.

This is an interesting problem because this is the foundation of artificial intelligence finding the correct path to its target while avoiding danger and roadblocks.

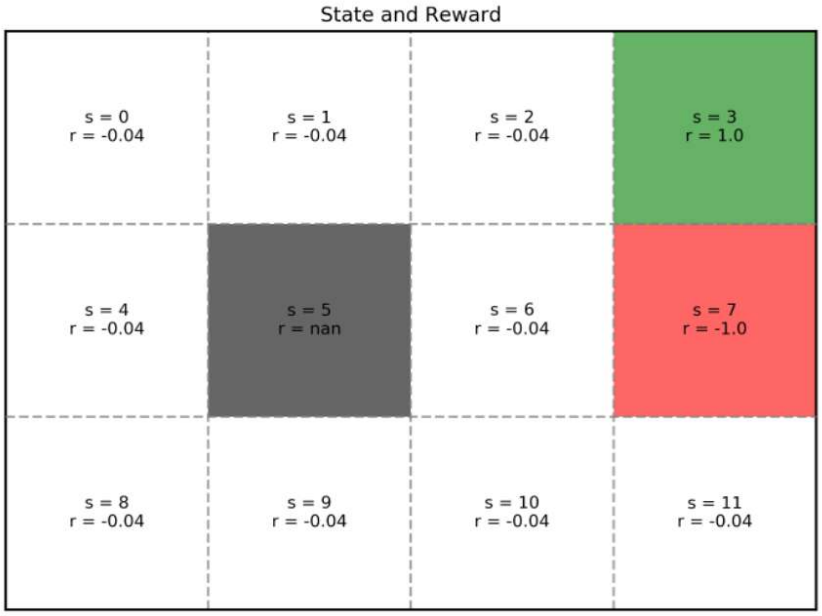


How it's related to Markov Decision processes:

(1) state: our Pac man can be in any of the 12 cells, being in each one represents a state

(2) action: our Pac man can move in 4 directions, up, down, left and right, but when it hits the wall or the boundary, it will be forced to stay at where it is

(3) reward: when moving to an empty state, reward is -0.04; when moving to the red and green, reward is -1 and 1 respectively.



2. 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

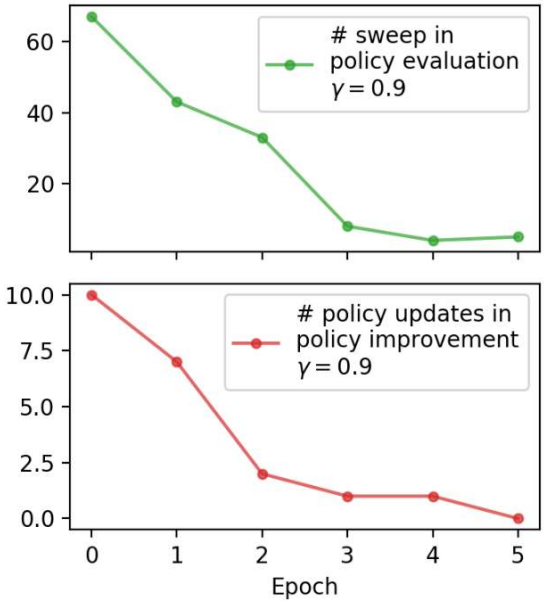
loses his stake. The game ends when the gambler wins by reaching his goal of 1000 dollars, or loses by running out of money. On each flip, the gambler must decide how many (integer) dollars to stake. The probability of heads is p, the probability of tails is (1-p).
How it's related to Markov Decision processes:
(1) state: the amount of money the gambler has
(2) action: the amount of money gambler is betting, in each round, he can bet as low as 1 dollar, or as high as the amount of money he has, or the amount of money he needs to reach his goal 1000, whichever is smaller
(3) reward: Zero on all transitions except those on which the gambler reaches his goal, when it is +1.

II. Policy Iteration and Value Iteration

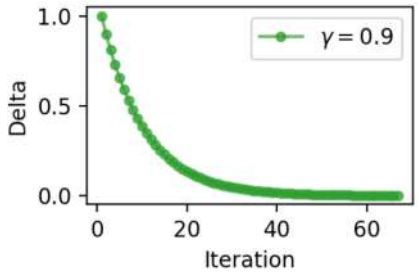
Policy Iteration and Value iteration are both dynamic programming algorithms to find optimal policy in a reinforcement learning environment.
In Policy Iteration, we start with a fixed policy, then it evaluates the policy until the changes of utility values of each state are marginal, then it improved the policy based on utility and loops through this process until policies converge.
In value iteration, we select the value function, choose the action that maximizes the utility of each state, and repeat the calculation of utility until the change is marginal.

1. Pac-Man Game

As Charles described in class, the process is stochastic, when the Pac-Man selects an action, there is 80% of chance it's going that direction, but 20% chance to go to 2 other directions. Also the default discount rate is 0.9 so it will take into account the future rewards more than a low discount rate.
The below graph shows the number of iterations needed in policy iteration. If the max value change of each state is less than 0.001 in the old policy and new policy, the iteration ends and we call this is converged. To achieve convergence, there are 6 policy improvements, and adding up all sweeps in each improvement, the total number of iteration is around 150. Also as policy improves. the number of iterations in policy evaluation is reduced because for each state the Pac-Man is in, it finds a better action in the current policy compared to the last policy. Eventually it will take very iterations to converge. The total run time for policy iteration is 0.04 second.

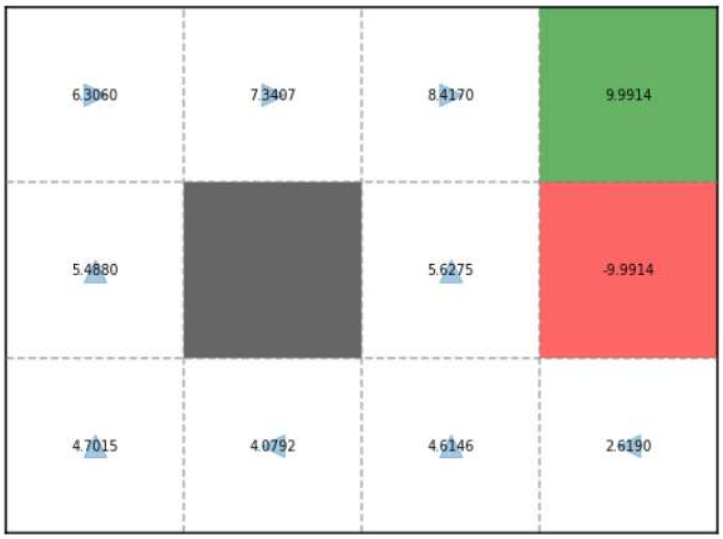


The below graph shows the number of iterations needed in value iteration. To achieve convergence to bring delta within 0.001 tolerance (convergence is defined above in policy iteration), 67 iterations are performed. The total run time is 0.02 second. The marginal delta decreases as iteration increases because we are having a better and better policy and do not have a high margin for utilities to decrease.



The result of policy iteration is:

The result of value iteration is:



According to above analysis, we see 3 things:

(1) Policy iteration is having more number of iterations compared to value iteration, 150 (6 policy improvements) and 67 respectively. This is because policy iteration consists of 2 parts, policy evaluation and policy improvement, every time when policy improves, a new round of policy evaluation starts. Depending on how good the random start policy is, number of policy improvements and policy evaluation may vary. But in each iteration, value iteration is more complex because it needs to take the max considering all possible actions.

(2) Policy iterations takes slightly more time than value iteration, 0.03 and 0.02 respectively. Although theoretically value iteration is more complex computationally than policy iterations due to its "max" argument to explore all different actions in a state, there are too few states and actions in this problem to show the difference in timing between both methods.

(3) Policy iteration and value iteration achieves the same optimal policy, but the values for each state are slightly different. This makes sense because both are methods to solve the Bellman equation and should have the same policy results and close utility values.

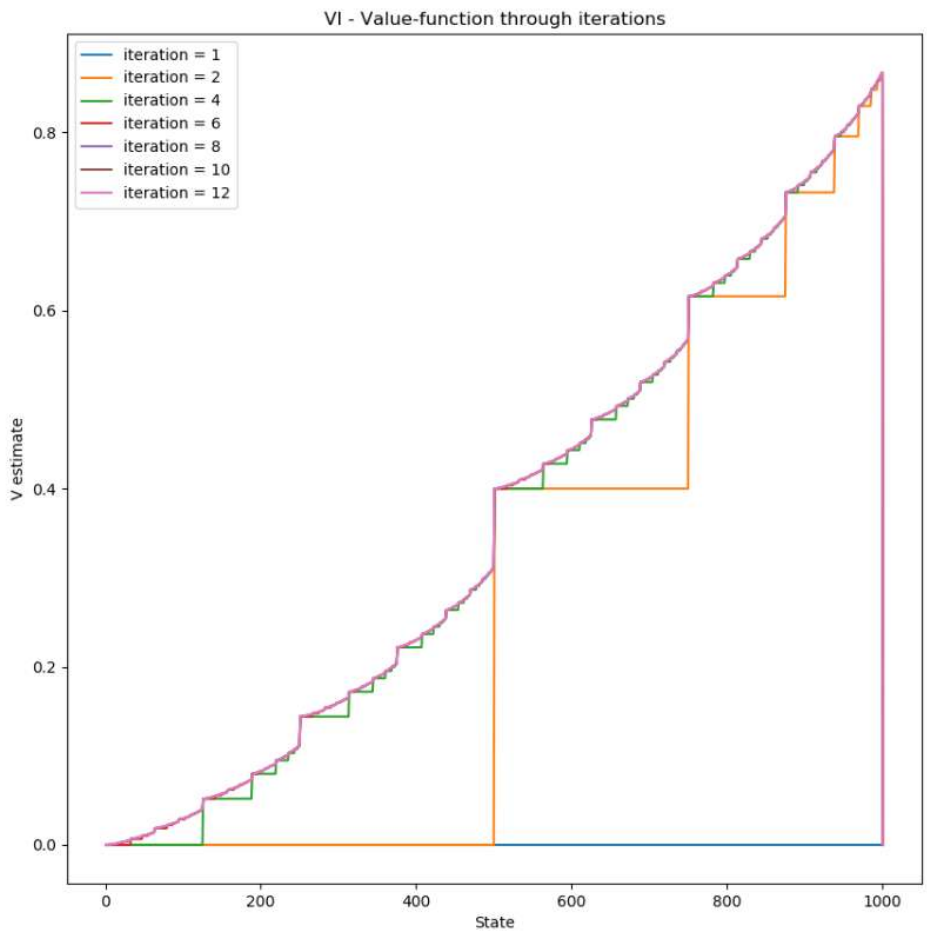
2. Gambler's Problem

The process is stochastic as well with 40% chance of winning and 60% chance of losing in every round the gambler plays.

Gamma is 0.9 to take into account further rewards in the future.

If the max value change of each state is less than 0.00001 in the old policy and new policy, the iteration ends and we call this is converged.

When Value iteration runs, it converges on 12^{th} iteration in 11 seconds. We can see below chart as iterations increase, values tend to become more stable instead of having large upward jumps. The reason is that the initial optimal actions are initiated as 0s and when the iterations begins, it starts to capture "milestones" like when state is 500, and as iterations go on, it starts to fill in more details between milestone states.



The best action from value iteration is shown below for each state, this is a seesaw shape of graph and there are huge jumps in "milestone" states we mentioned above, like 500 and 250. For state = 500, betting 500 to either win or lose has the best utility. Assume that at this state, we have 2 strategies, betting 500 (strategy 1), or betting 250, if we wins, we bet another 250 (strategy 2).

Winning probability = 40%

Losing probability = 60%

Value(strategy 1) = winning probability * 1+ losing probability * 0= 0.4*1 + 0.6* 0 = 0.4

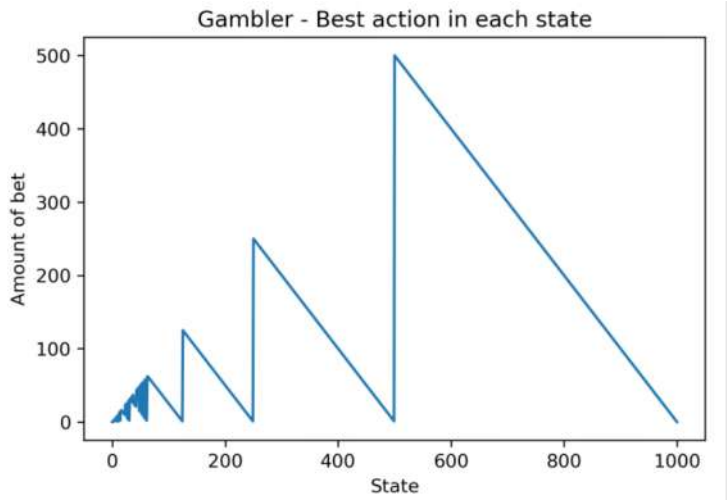
Value(strategy 2) = winning probability * Value(state = 750) + losing probability * Value(state = 250) = 0.4 * Value(state = 750) + 0.6 * Value(state = 250) <= 0.4 * (0.4 * 1 + 0.6 * 0) + 0.6 * 0.2(based on the value function above) = 0.28

Based on above example calculation, we can see the value of strategy 1 is better than strategy 2. Since strategy 2 is similar to other strategies that take smaller bets than 500, we can see that betting all when gambler has 500 is the optimal strategy.

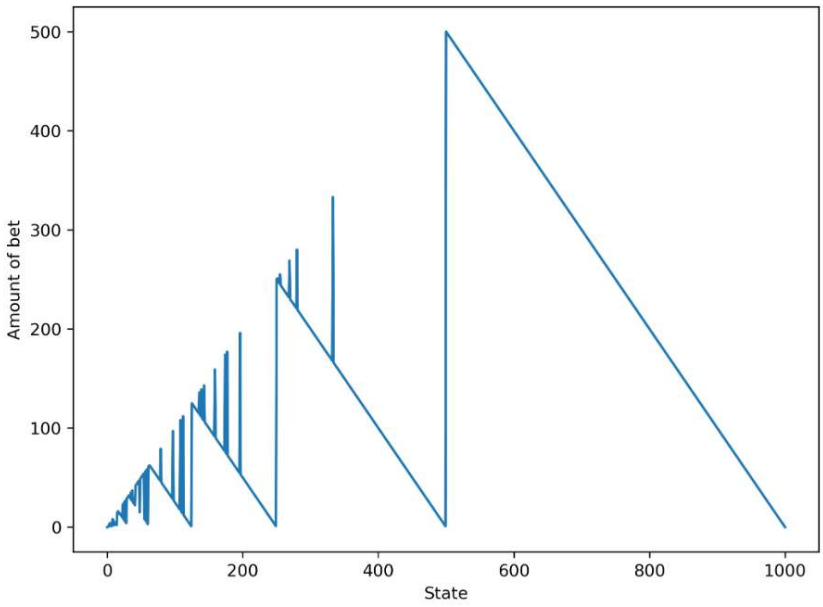
Another way to express why strategy 1 is better than all others is that the probability of winning is 40%, which means the odds are against the gambler and therefore gambler wants to bet as few times as possible.

When stake > 500, it's best for gambler to bet whatever it takes to get to 1000, because if the same reason above. The same reasoning can be applied to when gambler has stake of 250,

at that time max value for him is to bet all to reach 500.
When gambler has smaller stake less than 100, the betting is fluctuating a lot mainly because it's too small compared to ultimate goal of 1000 and can have many optimal policies in each state and the values usually don't have big differences.



When policy iteration runs, it converges on the 16th policy improvement, and in total there are 55 iterations, and ran in 86 seconds. Unlike value iteration, there are lots of spikes before capital is 300 mainly because when stake is small, there are many optimal options when stake is low. Unlike the Pac man's problem, gambler has much more action options in each state, and could potentially choose the ones with higher bets.



According to above analysis, we see 3 things:
(1) Policy iteration is having more number of iterations compared to value iteration, 55 (16 policy improvements) and 12 respectively. This is a large set of possible policies and we didn't start with a good random policy in the beginning, resulting in more iterations in policy iteration.
(2) Policy iterations takes much more time than value iteration, 86 and 11 respectively. I found

that more than 80% of time in policy iteration is spent in policy evaluation, and more specifically, solving the value amount. Therefore when the number of policy improves go up, there comes an increase amount of policy evaluation and thus lengthen the running time.
(3) Policy iteration and value iteration achieves a similar optimal policy, but policy iteration optimal solution generates a lot of spikes when gambler's capital is small. This is explained above, when stake is small there are many optimal solutions.

3. Comparison of the 2 problems

Gambler's problem is an MDP problem with a much larger size than the Pac-man problem, in that it has much more states and much more actions to take in each state. But based on the results we can see both iteration methods can solve large problems in a reasonably amount of time with similar results. In both problems, PI runs more slowly than VI since PI has more expensive policy evaluation for value calculation. The paper Reinforcement Learning: A Survey by Kaelbling and Littman states when gamma is large PI should be faster than AI, but I think I didn't achieve the same results in practice because of a bad random start policy and also my algorithm implementation which has a lot of for loops and scales up the computational complexity for policy iteration.

III. Reinforcement Learning Algorithm: Q-Learning

Q-learning is an off policy reinforcement learning algorithm that seeks to find the best action to take given the current state. It's considered off-policy because the q-learning function learns from actions that are outside the current policy, like taking random actions, and therefore a policy isn't needed. More specifically, q-learning seeks to learn a policy that maximizes the total reward.

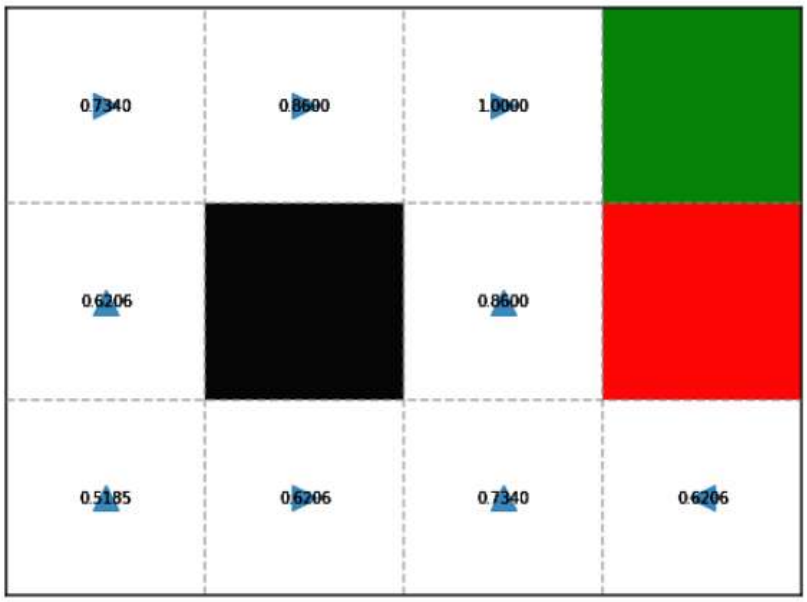
Gamma: discount rate for future reward

Learning rate: how much we accept new values versus old value

Epsilon: the percentage of time when we should take the best action (instead of a random action)

1. Pac-man Game

When applying q-learning in here, I use gamma = 0.9, epsilon = 0.8 since when doing value and policy iteration 80% of time we are taking designated action as well. Learning rate I chose 0.9 but I tried different learning rates and exploration strategy (changing epsilon) and all yields similar run time and run results. The run time is 0.09 second and the best action to take is shown below:



The only difference between q-learning results and the 2 MDP iterations is that the second cell in the bottom row changed from going right in q-learning. We can see q value in its left cell is lower than in the cell on the right, opposite of the values presented in value and policy iterations. But I think q-learning's result is more intuitively correct since going right takes only 3 steps to get to the green cell while going left takes 5 steps. This may be due to both MDP iterations are trying to avoid the red cell too much so it would rather take the farther path to get to the goal.

2. Gambler's problem

When applying Q-learning algorithm, I use gamma = 0.9, epsilon = 0.8 and learning rate of 0.9 so we allow 20% of chance for exploration. The run time is 104 seconds. If I use lower rates of learning rate it runs more slowly. Compared to both iterations above, it takes longer to run.

The result is shown below, and it's very close to value iteration results except that we have some small spikes when stake is very low, but much less than the spikes in policy iteration. One possible explanation is that there are many optimal strategies when stake is low, anther explanation is that Q-learning could get stuck in local optimum when stake is low.