Student Name: **A Z Hasnain Kabir**
Student ID: **200042102**

CSE 4618
Artificial Intelligence Lab
Lab 5

Question 1

We need to write a value iteration agent.

Analysis of the problem

We need to complete the class *ValueIterationAgent* inside *ValueIterationAgents.py*. The function *computeActionFromValues(state)* computes the best action according to values and *computeQValueFromValues(state, action)* returns Q-values of the (state, action) pair given by the value function.

Explanation of the solution

The value iteration algorithm runs for a specific number of iterations. For each state in MDP, it computes the Q-Values for all possible actions and updates the value of the state to be maximum Q-Value. In Q-Value calculation, it is the expected value of the immediate reward plus the discounted future value of the next state.

Question 2

We need to define an optimal policy for agent to cross the bridge.

Analysis of the problem

The agent will start near the low reward state. When we define one of the parameters Discount and Action optimally, it should cross the bridge on a straight path to the high rewards state.

Explanation of the solution

If we analyse the discount parameter, the agent will consider future rewards more heavily if the discount value is close to 1. On the other hand, if discount value is close to 0, it will choose immediate reward. So, the value of discount as 0.9 is ideal for achieving the given solution. In contrast, noise is similar to the chance of failure in each action. So if noise is kept significantly low, the agent that wants to move forward will move forward correctly. Therefore, we define noise as 0.01 and the agent has 99% chance that each action is successful.

Interesting findings

The value of noise needs to be significantly low, even if we define noise as low as 10% or even 5%, it will still not be enough.

Question 3

Define optimal policies for Gridworld and attempt to produce given outputs.

Analysis of the problem

In a Gridworld scenario, we should try and produce the given outputs by tuning the parameters discount, noise and given rewards.

- Prefer the close exit (+1), risking the cliff (-10)
- Prefer the close exit (+1), avoiding the cliff (-1)
- Prefer the distant exit (+10), risking the cliff (-10)
- Prefer the distant exit (+10), avoiding the cliff (-10)
- Avoid both exits and the cliff (so an episode should never terminate)

Explanation of the solution

The following outcomes can be achieved by defining the three parameters: A closer reward will be considered if discount value is close to 0 and distant reward will be considered if discount is closer to 1. A low value for noise will make the agent more likely to be successful when executing each of the action and vice versa. Living reward determines the reward for simply existing. For that reason, the agent will try to stay in the game more times than not if living reward is high and conversely, will try to exit the game faster if living reward is less.
With all of these in mind, we define the following parameters:
For reaching the closer exit while choosing the shorter path, we define discount as low (0.01), noise as low (0) and living reward as (0.01). For reaching the closer exit while choosing the longer path, we define discount as low (0.01), noise as low (0.01) and living reward as high (0.9). For reaching the distant exit while choosing the shorter path, we define discount as high (0.9), noise as low (0.01) and living reward as low (0.01). For reaching the distant exit while choosing the longer path, we define discount as high (0.9), noise as somewhat high (0.2) and living reward as somewhat high (0.1). Lastly, for avoiding both exits and choosing the longer path (in this case it should never terminate), we define discount as high (0.9), noise as high (0.9) and living reward as high (1).

Problems faced

In each case there exist some sort of threshold as to how high is a high value and how low is a low value. For instance 0.1 in many cases is not that much of a low value.

Solution to the problem

Different values are used and checked for each parameters to check just how high is a high value and just how low is a low value. Through this I could solve my problems.

Question 4

We need to implement asynchronous value iteration

Analysis of the problem

We need to write a value iteration agent in *AsynchronousValueIterationAgent* partially specified in *valueIterationAgents.py*.

Explanation of the solution

The solution is somewhat similar to value iteration agent except it checks each state in one iteration instead of iterating over all the states in each iteration. For example, in the first iteration, only the first state is updated whereas in the second iteration, only second state is updated.

Question 5

We need to implement Prioritized sweeping value iteration.

Analysis of the problem

A class called *PrioritizedSweepingValueIterationAgent* is defined which inherits *AsynchronousValueIterationAgent*. We are to implement this class accordingly.

Explanation of the solution

In this solution, we start by determining the predecessors of a given state. For every possible action from that predecessor we store the next states. We use a priority queue to store the difference between value of state and q-value based on a policy. Priority queue is used to prioritize those with the biggest changes. Throughout each iteration, states are revised based on their priority, with a particular emphasis on those demonstrating the most substantial disparities between current and anticipated values.

Interesting findings

Traditional value iteration involves updating the value of all states in each iteration, which can be computationally expensive for large state spaces. Prioritized sweeping focuses updates on states where the biggest changes in value are likely to occur, thus potentially reducing the number of updates needed.