Sujay Jakka

Svj0007@auburn.edu

COMP 5600

April 15, 2025

Assignment 4 Report

In this report, I will detail my algorithmic and design choices for Assignment 4. This report will consist of 6 sections, each dedicated to explaining my thought process for each question of the assignment. Furthermore, most of everything that is said in this report is also commented in my code to prevent any confusion.

Question 1

For Question 1, we had to complete the ValueIterationAgent class. Specifically, we had to implement the runValueIteration, computeActionFromValues, and computeQValueFromValues functions. The computeActionFromValues just returns the best action for a state by finding what action has the highest Q-value. Furthermore, the computeQValueFromValues function just returns the Q-value given a state and action. For the computeQValueFromValues function I basically followed the formula given in the assignment document and lecture where you do a weighted sum based on the reward given by transitioning to a successor state given the current state and action, the value of the successor state at the previous timestep times the discount factor, and the transition probability which is the probability of moving to the successor state given the current state and action. For this function I set my Q-value variable to 0 and iterated through every successor state and corresponding probability of the current state and action. I was able to do this through the getTransitionStatesAndProbs function. I added to my Q-value the product of the transition probability times the sum of the

reward and the product of the previous utility of the successor state times the discount factor. I then returned this Q-value. Now for the computeActionFromValues function I first check if the state being passed in as input is a terminal state. If it is a terminal state, that means no actions can be taken from it so we just return None as the best action. Then I initialize two variables one that keeps track of the best action and one that keeps track of the best Q-value. I iterate through all possible actions of the state which can be done with mdp.getPossibleActions and for each state, action pair I find the Q-value using the function we implemented before which is the computeQValueFromValues. Once I have the Q-value for the state, action pair I check if this is the highest Q-value seen so far, if it is I update my best action and Q-value variables. Lastly, I implemented the runValueIteration function which is in charge of actually running the different timesteps of the Value Iteration process. I first iterate through 0 to the inputted iterations, then I create a copy values dictionary from the current values dictionary. This is an important step as we want to make sure all of the states in iteration k + 1 only use values of states in iteration k. I then iterate through every state in the MDP and I skip the state if it is a terminal state as its value will always be 0. I then set the best Q-value for this state to be negative infinity. Next, I iterate through every possible action from this state and find the Q-value for this state, action pair. I update my best Q-value variable whenever I see an action with a higher Q-value. Lastly, I set my copied values dictionary, which represents the values for the current iteration, to the best Q-value seen for that state. After I iterate through all states, I set my self.values dictionary to the copied dictionary.

Question 2

For question 2, we are given a grid filled with terminal states in the bottom row each with a payoff of -10 and a terminal state in the center of the board with a payoff of +1 and lastly a terminal state in the last column of the middle row with a payoff of +10. Our goal is to find working discount, noise, and living reward parameters for 5 different policy types. The first policy type is preferring the close exit but risking the cliff which are the -10 payoff terminal states. I chose my discount factor to be 0.2, my noise value to be 0.0, and my living reward parameter to be 0.2. I chose my discount factor to be very low so that the agent prefers a smaller payoff sooner than a larger payoff down the line. Next, I chose the noise to be 0.0 so the agent actually takes the path of risking the cliff. If the noise is 0.0, the agent will not have anything to worry about as their action are now deterministic meaning they will not accidently land in a -10 terminal state. Lastly, I chose the living reward to be 0.2 which is pretty low so the agent understands that there is not much benefit to continue to live and to end the game quickly. For the next policy, we want to prefer the close exit but avoid the cliff. For this one I bumped up the noise to 0.3 so now there is a decent chance that the agent falls onto the cliff. This will cause the agent to avoid the cliff entirely. Next, I set my discount factor to 0.3 this is so that the agent does not prefer the +10 terminal state and decides to go for the closer exit. Lastly, I chose my living reward to be -0.1 to further discourage the farther higher payoff terminal state. For the third policy, they want us to produce an agent that prefers the distance exit but risks the cliff. This was pretty trivial. I just set my discount factor, noise, and living reward to 0.9, 0.0, and -0.1 each. I set my discount factor be high so the reward for reaching the high payoff terminal state does not erode much over time. I set 0.0 to be the noise, so the agent can risk being near the cliff without actually falling into the cliff. For the fourth policy, we prefer the distant exit but we want to avoid the cliff. Here I set my discount factor, noise, and living reward to be 0.9, 0.3, and 0.1

respectively. I set my discount factor to be high similar to the previous policy so the farther exit's reward doesn't erode much over time. I then set my noise to be 0.3, so the agent has a pretty decent chance of falling into the cliff. Lastly, I set my living reward from -0.1 to 0.1 so the agent prefers to live meaning it will take the farther exit. For the last policy, the agent avoids both exits and the cliffs. This basically means that the episode should never terminate meaning the agent should never reach a terminal state. Here, I set my discount factor be extremely low(0.01) so the rewards gained from reaching the terminal state does not matter anymore. Next, I set my noise to 0.0 so the agent has no chance of falling into the cliff and reaching the terminal state. Lastly, I set my living reward to be 10 so the agent heavily prefers to live.

Question 3

For question 3, we are asked to implement the QLearningAgent class which means specifically implementing the update, computeValueFromQValues, getQValue, and computeActionFromQValues. Q-Learning is good for when we do not know the rewards of our actions or the transition probabilities. Q-Learning is a method that learns from trial and error. Before I implemented any of the functions, I created a dictionary using the util class which represents the Q-values. The default value for all keys are 0. For the getQValue function, I just return the value that corresponds to the tuple (state, action) in our Q-value dictionary. For the computeValueFromQValues method, I first get all legal actions by doing self.getLegalActions(state). I then check if there are any legal actions from this state, if there are not any legal actions then I return 0 as this is a terminal state. Next, I initialize a variable called best Q-value and I set this value to negative infinity. I then loop through each action and find the corresponding Q-value for the state, action pair. If it is higher than our current best Q-value, I

then update our Q-value. At the very end, I return best Q-value. For computeActionFromQValues it is very similar to computeValueFromQValues except if there are no legal action I return None as the best action. Furthermore, I have a variable now to also update the best action which is also called “best action”. I loop through each action in the legal actions and retrieve the Q-value. If the Q-value is better than the best Q-value seen so far, I update the best action and best Q-value variables. However, if it is equal to the best Q-value seen so far I choose a random action between the current action and the best seen action so far. I then return the best action. Lastly, for the update function I follow the formula from the lecture slides which is (1- alpha) * V(s) + (alpha * sample) where sample is equal to R(s, pi(s), s’) + discount factor * V(s’). I find V(s’) by calling the computeValueFromQValues on next state. I make sure to update my state, action pair in my dictionary before I exit the function.

Question 4

For question 4, we are asked to implement the getAction function in which we choose a random action with a probability of epsilon. Otherwise, we choose the best policy action. Implementing this was trivial, if there are no legal actions this means that this state is a terminal state and I just return None as there are no actions to take from a terminal state. With a probability of epsilon, I choose a random action using the random.choice function. Otherwise, I choose the best policy action using computeActionFromQValues function.

Question 5

For question 5, we are asked to test our Q-Learning agent by running 2000 training episodes and 100 training episodes. My implementation was able to successfully win 98% of the episodes. I made sure that my implementation considered unseen actions.

Question 6

For question 6, we are tasked to complete the ApproximateQAgent class. Specifically, we are asked to complete the getQValue and update functions. ApproximateQAgent is a subclass of QLearningAgent so all other functions are inherited except for the functions mentioned above that will be overridden. For the getQValue function, I simply return the dot product between the features of the given state, action pair and the weight dictionary. This follows the approximate Q-Learning formula given to us in the assignment details. Furthermore, for the update function I made sure to find the difference value by following the difference formula. Once I had the difference, I iterated through each feature in my feature vector and updated its corresponding weight using the difference, learning rate, and current weight value. Furthermore, I made sure that my methods in QLearning Agent call getQValue instead of accessing the Q-values directly. This will allow many functions in ApproximateQAgent to not have to be overridden.