Report LAB 1.5

Question 1 (6 points) Write a value iteration agent in **ValueIterationAgent**, which has been partially specified for you in **valueIterationAgents.py**. Your value iteration agent is an offline planner, not a reinforcement learning agent, and so the relevant training option is the number of iterations of value iteration it should run (option -i) in its initial planning phase. **ValueIterationAgent** takes an MDP on construction and runs value iteration for the specified number of iterations before the constructor returns.



**Initialization of the ValueIterationAgent:**

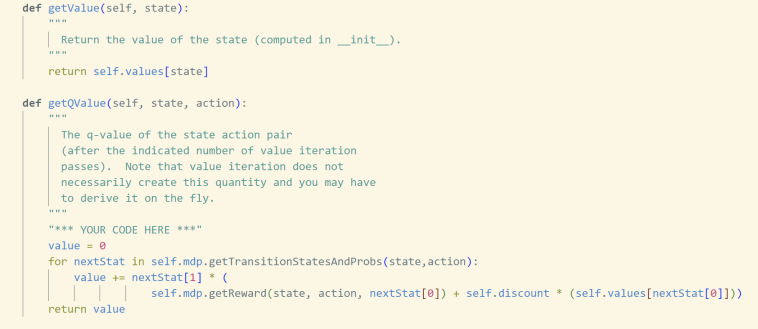
* The agent starts by obtaining all possible states from the MDP using mdp.getStates() and sets up a util.Counter to store the value estimates for each state.
* The value iteration process is executed for a specified number of iterations (self.iterations). During each iteration, the agent updates the value function for every state by evaluating the expected values of all possible actions.

**Updating the Value Function:**

* For each state, the agent calculates the maximum Q-value by considering all available actions using self.getQValue(state, action). The state’s value is updated to reflect the highest expected reward. If a state's computed value is greater than the current maximum (maxValue), maxValue is updated to this new value.

**Q-value Calculation:**

* The getQValue method is used to compute the expected utility of taking a specific action in a given state. It takes into account all possible next states, their transition probabilities, and the rewards associated with transitioning to these states. This value determines the optimal action to take from each state.

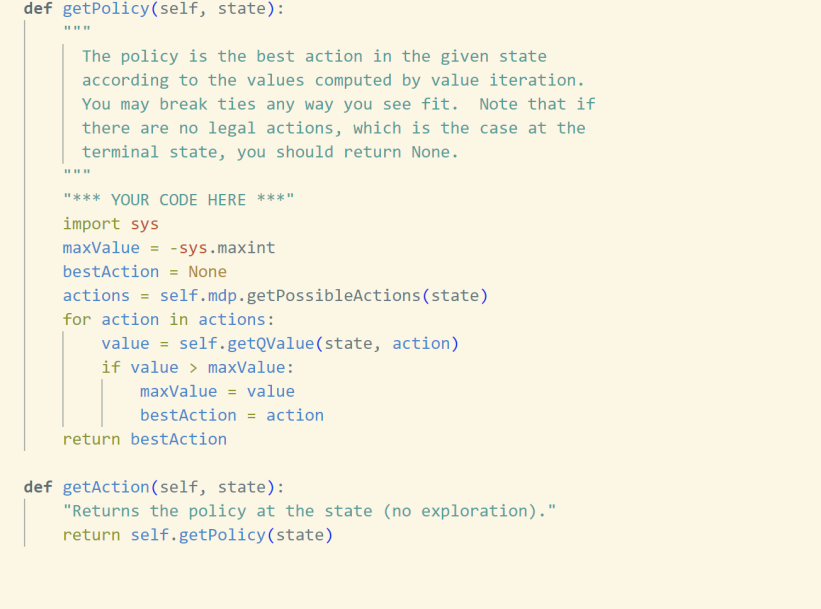


**getPolicy Method:**

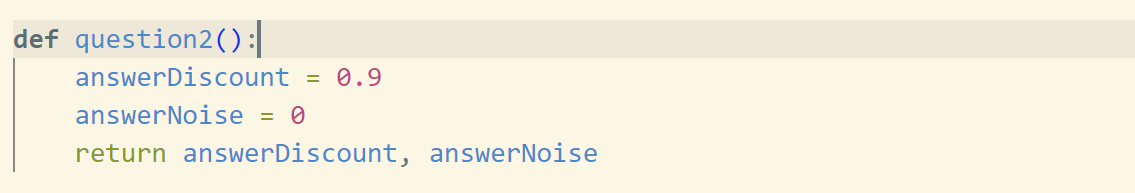
* The getPolicy method determines the best action to take in a given state based on the current value function.
* It iterates over all possible actions for the given state and calculates the Q-value for each action using self.getQValue(state, action).
* The action with the highest Q-value is selected as the optimal action. In case of ties (multiple actions having the same Q-value), one of the tied actions will be returned.
* If there are no legal actions (e.g., in a terminal state), the method returns None.

**getAction Method:**

* The getAction method is a straightforward wrapper around getPolicy. It simply calls getPolicy to return the best action for the given state based on the current policy and value function. This method does not perform any exploration and relies entirely on the computed policy to decide the action.



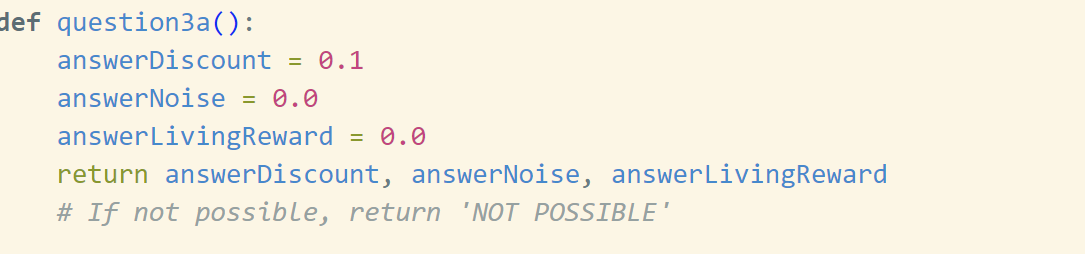
Question 2 (1 point) BridgeGrid is a grid world map with the a low-reward terminal state and a high-reward terminal state separated by a narrow "bridge", on either side of which is a chasm of high negative reward. The agent starts near the low-reward state. With the default discount of 0.9 and the default noise of 0.2, the optimal policy does not cross the bridge. Change only ONE of the discount and noise parameters so that the optimal policy causes the agent to attempt to cross the bridge. Put your answer in question2() of analysis.py. (Noise refers to how often an agent ends up in an unintended successor state when they perform an action.)



* The answerNoise is set to 0 to ensure that the agent's actions are deterministic, meaning there is no randomness in the environment's responses. This encourages the agent to cross the bridge with confidence, as it will reliably reach the intended destination, thereby increasing the likelihood of aiming for the high-reward terminal state.
* The answerDiscount (discount factor) remains at 0.9, as we still want the agent to consider future rewards and be motivated to reach high-value states.

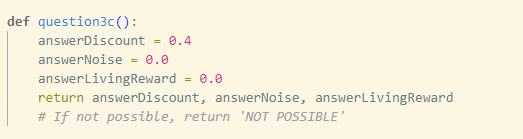
Question 3 (5 points) Consider the DiscountGrid layout, shown below. This grid has two terminal states with positive payoff (shown in green), a close exit with payoff +1 and a distant exit with payoff +10. The bottom row of the grid consists of terminal states with negative payoff (shown in red); each state in this "cliff" region has payoff -10. The starting state is the yellow square. We distinguish between two types of paths: (1) paths that "risk the cliff" and travel near the bottom row of the grid; these paths are shorter but risk earning a large negative payoff, and are represented by the red arrow in the figure below. (2) paths that "avoid the cliff" and travel along the top edge of the grid. These paths are longer but are less likely to incur huge negative payoffs. These paths are represented by the green arrow in the figure below.

1. **refer the close exit (+1), risking the cliff (-10)**:

- 

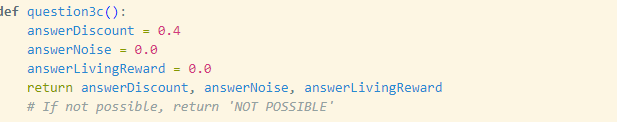
* **Low discount factor (answerDiscount = 0.1)**: This setting makes the agent more focused on immediate rewards rather than long-term gains, encouraging it to take the shorter, riskier route to reach its goal quickly.
* **Low noise (answerNoise = 0.0)**: With no noise, the agent can confidently take actions without worrying about unintended outcomes, making it less cautious.
* **Low living reward (answerLivingReward = 0.0)**: A neutral or slightly negative living reward ensures that the agent doesn't aim to stay alive indefinitely. It pushes the agent to reach a terminal state rather than aimlessly prolonging its existence.

1. Prefer the close exit (+1), but avoiding the cliff (-10)



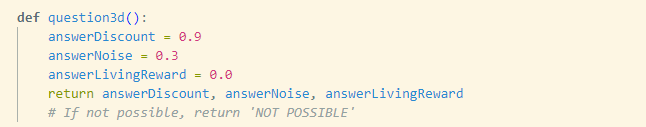
* **Moderate discount factor (answerDiscount = 0.4)**: This setting encourages the agent to consider future rewards, but not to the extent of being overly cautious. The agent balances between immediate and future rewards.
* **Low noise (answerNoise = 0.0)**: With no noise, the agent is confident in its actions, ensuring it can take the intended path without worrying about unintended movements.
* **Neutral living reward (answerLivingReward = 0.0)**: A neutral living reward prevents the agent from lingering indefinitely in non-terminal states, pushing it toward completing its goal without unnecessary hesitation.

1. Prefer the distant exit (+10), risking the cliff (-10)



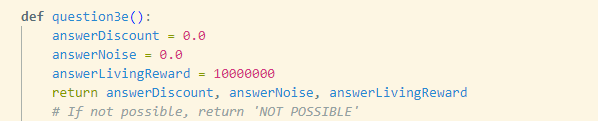
* In this scenario, the agent is expected to aim for the distant exit, even if it means taking a risk with the cliff, which means the agent should focus on maximizing large future rewards.
* **Moderate discount factor (answerDiscount = 0.4)**: This value helps the agent consider the distant exit (+10) as an attractive option while balancing the risk, ensuring future rewards are still important.
* **Low noise (answerNoise = 0.0)**: Ensures that the agent's actions are deterministic, allowing it to confidently navigate the risky path without being overly cautious.
* **Neutral living reward (answerLivingReward = 0.0)**: This setting encourages the agent to reach the exit promptly, rather than staying indefinitely in the environment.

1. Prefer the distant exit (+10), avoiding the cliff (-10)



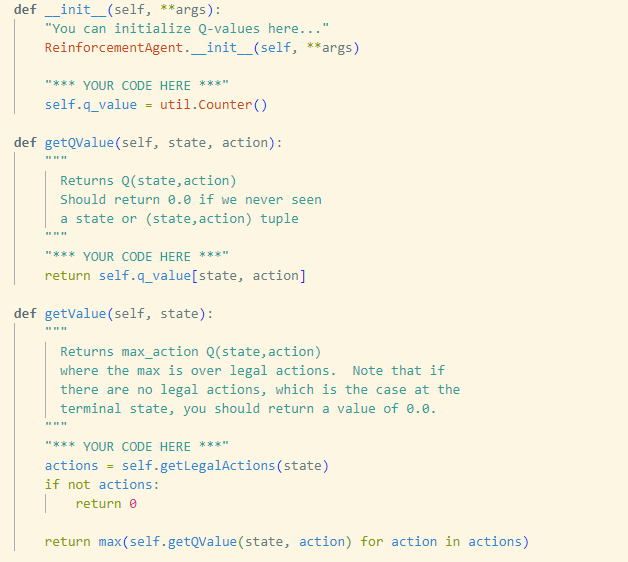
* The agent should aim for the distant exit while also being cautious to avoid the cliff, meaning it must strike a balance between prioritizing future rewards and maintaining safety.
* **High discount factor (answerDiscount = 0.9)**: This ensures that the agent places significant value on the distant +10 exit, emphasizing future rewards.
* **Moderate noise (answerNoise = 0.3)**: This introduces some uncertainty, making the agent more cautious and encouraging it to avoid risky paths.
* **Neutral living reward (answerLivingReward = 0.0)**: This ensures that the agent doesn't linger in the environment, motivating it to reach the exit instead of staying alive indefinitely.

1. Avoid both exits and the cliff (so an episode should never terminate)

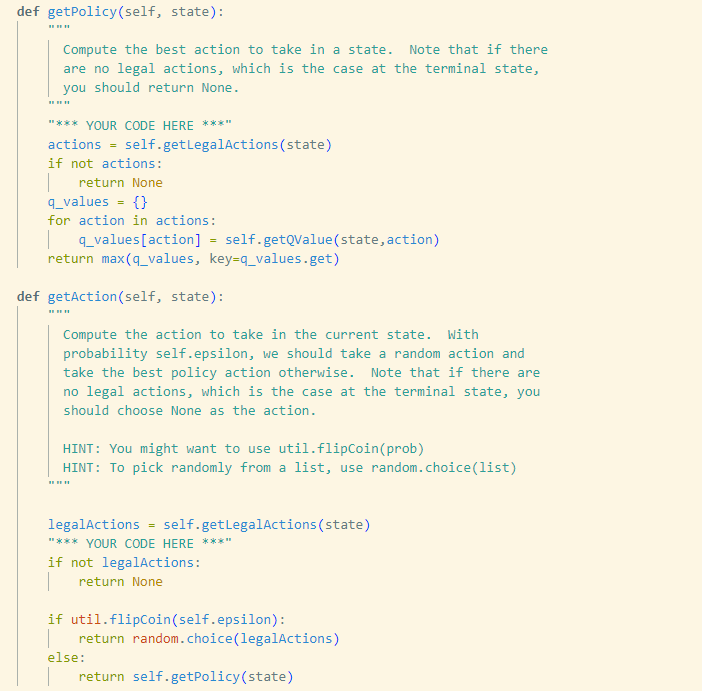
- 

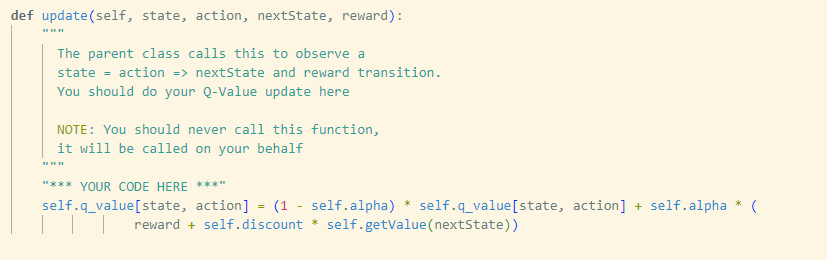
* The agent is designed to avoid both the exits and the cliff, meaning it should aim to avoid terminating the episode altogether.
* **Zero discount factor (answerDiscount = 0.0)**: This ensures the agent doesn't value future rewards, making it less inclined to exit.
* **Zero noise (answerNoise = 0.0)**: Guarantees that the agent's actions are deterministic, allowing it to move confidently without worrying about unintended movements.
* **High positive living reward (answerLivingReward = 1000000)**: This large positive living reward strongly encourages the agent to keep moving and exploring indefinitely, rather than seeking an exit.

Question 4 (5 points) You will now write a Q-learning agent, which does very little on construction, but instead learns by trial and error from interactions with the environment through its update(state, action, nextState, reward) method. A stub of a Q-learner is specified in QLearningAgent in qlearningAgents.py, and you can select it with the option '-a q'. For this question, you must implement the update, getValue, getQValue, and getPolicy methods.

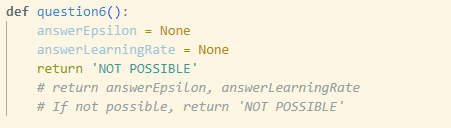


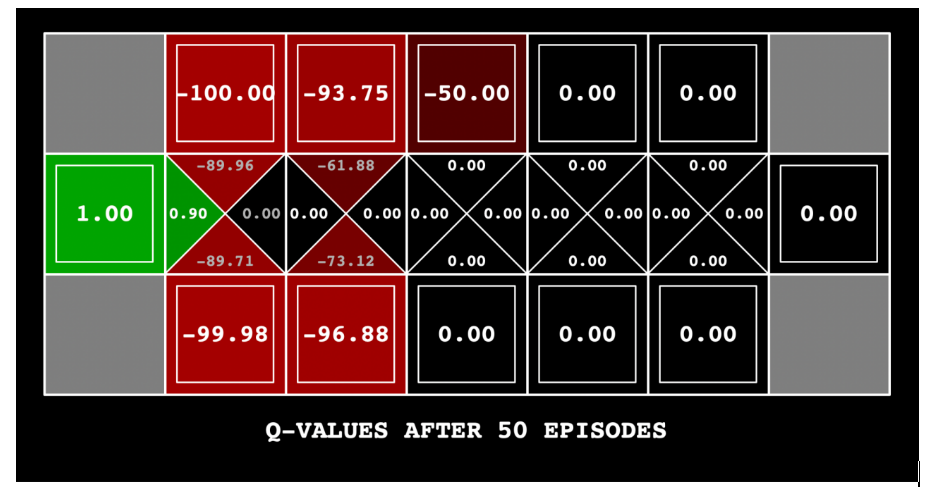
Question 5 (2 points) Complete your Q-learning agent by implementing epsilon-greedy action selection in getAction, meaning it chooses random actions an epsilon fraction of the time, and follows its current best Q-values otherwise.



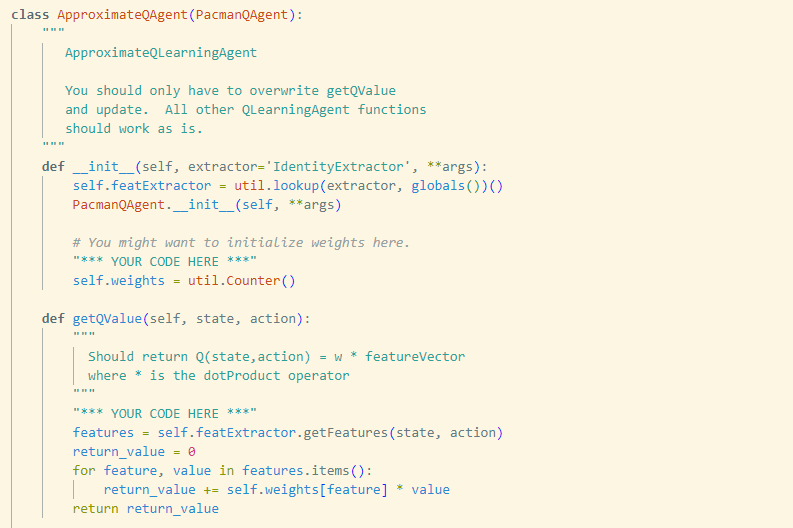


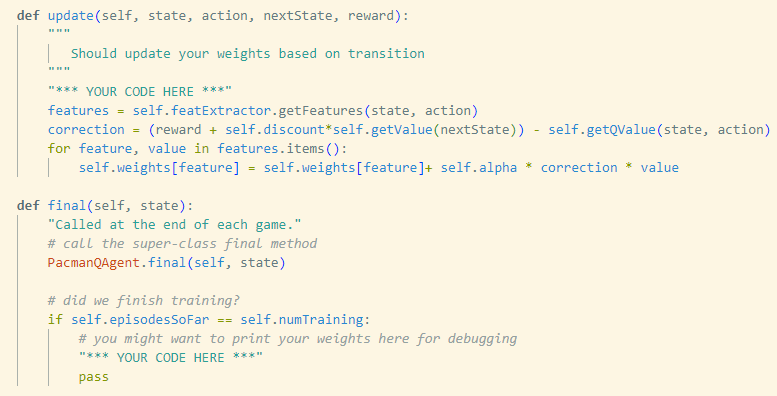
Question 6 (1 points) First, train a completely random Q-learner with the default learning rate on the noiseless BridgeGrid for 50 episodes and observe whether it finds the optimal policy.





Question 9 (3 points) Implement an approximate Q-learning agent that learns weights for features of states, where many states might share the same features. Write your implementation in ApproximateQAgent class in qlearningAgents.py, which is a subclass of PacmanQAgent.





**Q-Value Approximation**

* The ApproximateQAgent employs a linear approximation method for Q-values, enabling it to generalize effectively in larger or continuous state spaces. Instead of working with every possible state-action pair, the agent uses a feature-based representation to approximate the Q-values.

**Feature Extraction**

* The agent uses featExtractor to extract features from each state-action pair. These features provide a structured and often domain-specific representation of the state-action space, making the learning process more efficient and faster than handling raw state-action pairs.

**Weight Updates**

* The agent updates its weights incrementally based on observed transitions. After each transition (state, action, reward, next state), the agent adjusts the weights using the difference between the predicted Q-value (self.getQValue) and the target Q-value (calculated using the reward and the estimated future value from the next state). Over time, these weights converge to values that produce more accurate Q-value approximations.