Project 3: Reinforcement Learning

Introduction:

In this project you will implement Q-Learning using Malmo. The goal of Q-Learning is to learn a policy, which tells an agent what action to take under what circumstances. It does not require a model of the environment and can handle problems with stochastic transitions and rewards, without requiring adaptations. We have provided you with multiple layout files to test your code as well as a visual representation of the Q-Table.

Files you'll edit:

QLearning.py

Files you can ignore:

qlearning.xml - this it the layout file used to generate the grid for the agent to traverse
// more xml files will be added in the future to facilitate testing

Academic Dishonesty:

The University of Virginia's Honor Code: students pledge never to lie, cheat, or steal, and accept that the consequence for breaking this pledge is permanent dismissal from the University.

In general, we expect that you will be using code, examples, and ideas from many different websites and resources for your projects. This is allowed within reason. Wholesale copying of a project is not allowed and will result in an honor violation. In ALL cases, you need to cite all sources at the top of the file where the code or algorithm was used. Failure to properly attribute your sources will result in a 0 for the project at a minimum.

Getting Help:

You are not alone! If you find yourself stuck on something, contact the TAs for help. If you can't make our office hours, let us know and we will schedule more. We want these projects to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask.

Q-Learning:

In this project you will implement 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(reward, state, action, nextState) methods. A stub of a Q-learner is specified in *QLearning.py*.

In this assignment, the purpose of the agent is to learn how to navigate a given landscape and get to the redstone ore block to receive a large reward of 100 points. If the agent falls in the water, it will incur a negative reward of -100 points. The agent will also lose a point for each action it takes.

Note: please see the qlearning.xml file for more details on how rewards are assigned.

Question 1: Updating the Q-Table (5 Points)

For this question you will have to complete two methods: *updateQTable()* and *updateQTableFromTerminatingState().* Both of these methods are responsible for updating the Q-Table after an agent takes a particular action and receives some rewards.

In your updateQTable() method you will receive a sample transition (s, a, r, s'), where **s** is the previous state, **a** is the previous action, **r** is the reward and **s'** is the current state. The sample you received suggests that

$$Q(s,a) \approx r + \gamma \max_{a'} Q(s',a')$$

However we want to keep a running average of all our previously observed samples so we use

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)\left[r + \gamma \max_{a'} Q(s',a')\right]$$

You must implement this in your updateQTable() method, updating the `self.q_table[prev_state][prev_a]`. Please be sure to use the `self.alpha` and `self.gamma` constants we provided (where alpha is the learning rate and gamma is the reward discount factor).

Question 2: Updating the Q-Table from terminating state (5 Points)

In this question you must implement the *updateQTableFromTerminatingState().* This part is much simpler as it does not require you to average over your previous observations. The reward received in any particular terminating state should always be the same (either -100 or +100) and there are no future expected rewards to account for. Thus in the terminating state your ***Q(s, a)*** should simply be the reward you recieve.

Question 3: Choosing the correct action (5 Points)

In this question you will need to decide which action the agent should take based on the values in your Q-Table. To do this you will need to edit the *act()* method.

In the case where there are multiple maximum choices with equal values, you should select the best action randomly.

In order to ensure that the agent explores the problem space and does not simply follow the first path to the reward it finds, you will need to implement the epsilon-greedy action selection. This implies that the agent chooses random actions an epsilon fraction of the time, and follows its current best Q-values otherwise. Note that choosing a random action may result in choosing the best action - that is, you should not choose a random sub-optimal action, but rather any random legal action. Please be sure to use the `self.epsilon` constant we provided to calculate when your agent should carry out random movements.

Grading:

We will run your code on a number of different terrains to ensure that your code can learn the optimal policy of each given map. You will be awarded 5 points for a correct implementation of each method, for a total of 15 points. However, keep in mind that your agent must actually do something, so if you successfully complete the updateQTable() and *updateQTableFromTerminatingState()* but add no code to the act() method, your Q-Table will never get updated, your agent will stand in place and your implementation will receive a score of 0.