Artificial Intelligence
Term Project Documentation

Amelie Cameron

CSC 665-01 Professor Yoon

Due 05/20/18

Table of Contents

Project Information:

Reinforcement Learning Professor Yoon Spring 2018 CSC 665-01 Term Project Version 0.1
Amelie Cameron

Introduction:

This is a developer's guide for the CSC 665 Spring 2018 Python Reinforcement Learning Term Project. I implemented the ValueIterationAgents.py, the QLearningAgents.py, and the analysis.py files from the Berkeley project 3.

Scope of Work:

As a starting point for these projects, I was given a zip file for the Berkeley Project 3 to use as an outline. The scope of work for these projects included setting up the ValueIterationAgent class, including computeQValueFromValues and computeActionFromValues. I also wrote the qLearningAgent class including computeValueFromQValues, computeActionFromQValues, getAction, and update. The last was a subclass of PacmanQAgent called ApproximateQAgent which includes getQValue and update. The functions were tested and documentation created.

Background/Given Resources:

Attribution Information: The Pacman AI projects were developed at UC Berkeley.
The core projects and autograders were primarily created by John DeNero (denero@cs.berkeley.edu) and Dan Klein (klein@cs.berkeley.edu).
Student side autograding was added by Brad Miller, Nick Hay, and
Pieter Abbeel (pabbeel@cs.berkeley.edu).

Assumptions:

Development work was done on a MacBook Air 1.8 GHz Intel Core i5
PyCharm 2017.3.3 was used as the development environment.

Project Description:

For the Artificial Intelligence term project I presented the Reinforcement Learning project three from the UC Berkeley Introduction to AI Course. There are eight sections that cover Value Iteration, Bridge Crossing Analysis, Policies, Q-Learning, Epsilon Greedy, Bridge Crossing Revisited, Q-Learning and Pacman, and Approximate Q-Learning. I will go into detail about each section. Similar to the first two projects, there is an autograder that checks for code accuracy and completeness. I am modifying the Python files valueIterationAgents.py for solving known Markov Decision Processes, qLearningAgents.py(for Gridworld, Crawler, and Pacman), and analysis.py which is a file to put all of the project answers.

Value Iteration (6 points): The file valueIterationAgent takes an MDP and runs value iteration for a given amount of iterations(-i). It also computes k-step estimates of the optimal values Vk. There are also two methods I implemented in this file, computeActionFromValues(state) which finds the best action by using self.values, and computeQValueFromValues(state, action) which finds the Q-Value of the given parameters by using self.values.

Bridge Crossing Analysis (1 point): Within BridgeGrid, the map has a low-reward start state and a high-reward goal state. The agent begins at the low reward side of the "bridge" with negative values on either side. By changing one of the discount and noise parameters, the agent attempts to cross the bridge to the high-reward terminal state.

Policies (5 points): The DiscountGrid has two terminal states: a close exit with a payoff of one, and a far exit with a payoff of ten. The agent can chose to go close to the "cliff" where there is a risk of a negative payoff or a longer path that follows the top side of the board with no risk. By choosing settings of the discount, noise, and living reward parameters, the MDP will produce five different optimal policy types: close exit near the cliff, close exit avoiding the cliff, far exit near the cliff, far exit avoiding the cliff, and avoid both exits and the cliff (never terminates).

Q-Learning (5 points): The Q-Learning Agent learns by trial-and-error through the method update(state, action, nextState, reward). There are four methods to implement for this question: getQValue, computeActionFromQValues, computeValueFromQValues, and update. Unseen actions have a Q-value of zero, otherwise calling getQValue will return an accurate value for the action. It is also important to note that the agent only learns about previous states rather than about future states.

Epsilon Greedy (3 points): Within getAction, I implemented the epsilon-greedy action selection. It picks random actions "an epsilon fraction of the time" from any legal action. While in the learning phase, the average returns were lower than the value iteration agent Q-values. I also ran the crawler robot to test how the step delay and other parameters affect the crawler.

Bridge Crossing Revisited (1 point): For this problem, I trained a Q-learner on the BridgeGrid fifty times to see whether an optimal policy was found. It had a default learning rate and no noise. Then I repeated the experiment with an epsilon value of zero. I tried to find a learning rate and epsilon value which caused the Q-learner to find the optimal policy after fifty iterations greater than ninety nine percent of the time. However, I had to return Not Possible because I wasn't able to find an optimal policy within fifty iterations.

Q-Learning and Pacman (1 point): There are two modes for Pacman in this problem: training mode and testing mode. The first is spent learning accurate values of positions and actions. The second is spent implementing the learned policy. The PacmanQAgent has default learning parameters(epsilon=0.05, alpha=0.2, gamma=0.8). A total of 2010 games were played, the first 2000 being training games with no output. After training is complete, the Pacman won ninety percent of test games.

Approximate Q-Learning (3 points): In qLearningAgents.py, I implemented a subclass of PacmanQAgent called ApproximateQAgent that learns weights for features of states. Feature functions are provided in featureExtractors.py. I will implement the weight vector as a dictionary that maps features to weight values and will update the weight vectors the same way I updated Q-values. The approximate Q-Learning agent should win almost all of the games, even with only fifty training games provided.

Solution Design:

The overall design and structure of this program was predetermined by the UC Berkeley project creators. However, I designed the solutions for the following classees: valueIterationAgents, qLearningAgent, ApproximateQAgent. I also filled in answers for the discount factors and noise percentages in the analysis.py file. The architecture diagrams of the program can be found in the Documentation file within the zip for the project.

Architecture Diagrams:

The architecture diagrams were too large to fit within this document. They are both included in the zip file for the project. There is a diagram of the packages and one of the classes. I created them using Pyreverse. The files are called DiagramClasses.png and DiagramPackages.png. They are located in a file called Documentation within the source code zip file along with the "CSC665 Term Project Video."

Solution Implementation:

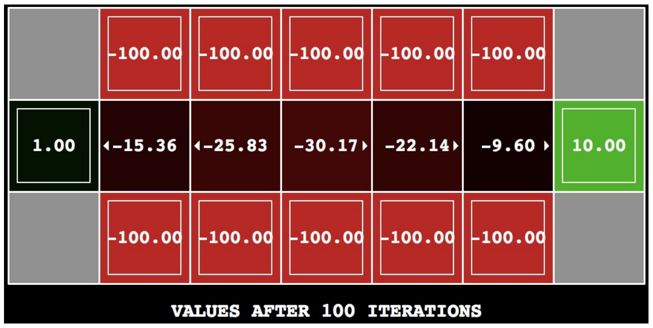
For each section of the project, I will got into detail about how I wrote my code and what I struggled with the most. The code also has line-by-line comments to clearly explain my solutions. Listed below are the solution implementations for every part of the project.

Value Iteration:

The value iteration agent takes an MDP on construction. I created a counter to keep track of the iterations. Then I looped through the number of iterations(self.iterations) using a while loop. Within the loop, I saved the values from self.values into the dictionary cur_values. I got the states from self.mdp.getStates and looped through them. For each state, I checked to make sure it wasn't the terminal state. Then I got the possible actions for each state and created another dictionary to keep track of the values of each action. I looped through the actions and called computeQValueFromValues(state, action). The function uses getTransitionStatesAndProbs(state, action) to get the transitions. Then I used the Bellman equation to calculate the Q values. I looped through the transitions and summed up the product of the value of the next state, the discount factor, and the probability. Then I added the reward of the next state to the summed q value. That q value was then added to the list of action scores. Then the highest q value was added to the dictionary of values. I incremented the iteration counter and recopied the values list to the cur_values list to maintain the changes after each iteration.

Bridge Crossing Analysis:

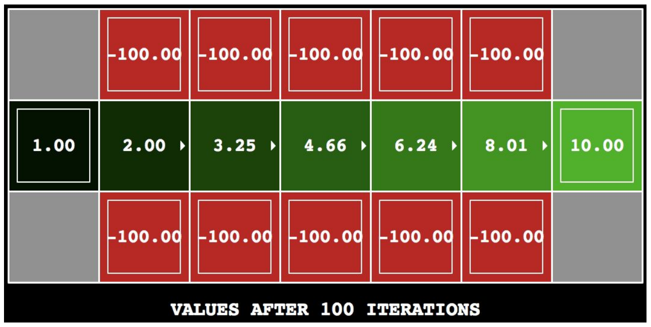
The agent started near the low-reward state. It has a default discount of 0.9 and the default noise of 0.2, the optimal policy does not cross the bridge. I changed one of the discount and noise parameters so that the optimal policy causes the agent to attempt to cross the bridge. I used trial and error to figure out which parameter would effect to agent and cause it to cross the bridge. Changing the discount rate in any way caused the bridge to be negative, so the agent did not cross the bridge. Thus, I decided to change the noise parameter so that the agent is more likely to take a given action across the bridge. The noise percentage has to be very small unless the values on the bridge will remain negative. I found that if the agent moves in the correct direction 99 percent of the time that the bridge values will be positive and the agent will cross the bridge.



	-100.00	-100.00	-100.00	-100.00	-100.00	
1.00	◂-15.36	◂-25.83	-30.17▸	-22.14▸	-9.60 ▸	10.00
	-100.00	-100.00	-100.00	-100.00	-100.00	

VALUES AFTER 100 ITERATIONS

Discount: 0.8, Noise: 0.2



	-100.00	-100.00	-100.00	-100.00	-100.00	
1.00	2.00 ▸	3.25 ▸	4.66 ▸	6.24 ▸	8.01 ▸	10.00
	-100.00	-100.00	-100.00	-100.00	-100.00	

VALUES AFTER 100 ITERATIONS

Discount: 0.9, Noise 0.01

Policies:

I had a difficult time with this part of the assignment. Since I was unable to find an effective method for testing these values, I used the autograder to determine whether or not the values were correct. There is room for variation because I noticed I was able to fine-tune the values for discount and noise and the tests would still pass. The optimal policy types I had to produce are as follows:

1. Prefer the close exit (+1), risking the cliff (-10)
   Discount 0.3, Noise 0.0
   I chose a low discount value so it would prioritize immediate rewards rather than distant future rewards and no noise so that the agent would always move in the intended direction to avoid falling off the cliff when the agent passes by it.
2. Prefer the close exit (+1), but avoiding the cliff (-10)
   Discount 0.2, Noise 0.1
   I chose a low discount value so it would prioritize immediate rewards rather than distant future rewards and low noise so that the agent would always move in the intended direction to avoid falling off the cliff in case it can not avoid it.

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

   Discount 0.7, Noise 0.1

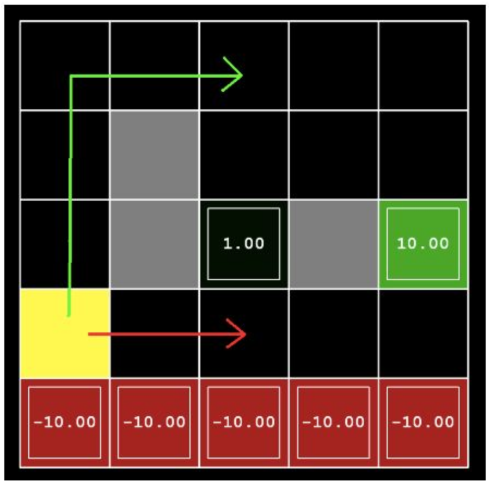
   I chose a relatively high discount value so it would prioritize large future rewards rather than smaller immediate rewards and low noise so that the agent would always move in the intended direction to avoid falling off the cliff when the agent passes by it.
4. Prefer the distant exit (+10), avoiding the cliff (-10)

   Discount 0.8, Noise 0.2

   I chose a relatively high discount value so it would prioritize large future rewards rather than smaller immediate rewards and low noise so that the agent would always move in the intended direction to avoid falling off the cliff in case the agent cannot avoid it.
5. Avoid both exits and the cliff (so an episode should never terminate)

   Discount 0.4, Noise 0.4

   I chose average discount and noise values because I did not want the agent to prioritize either exit or path so that it avoid both exits and the cliff. Thus, the episode would never terminate.



Q-Learning:

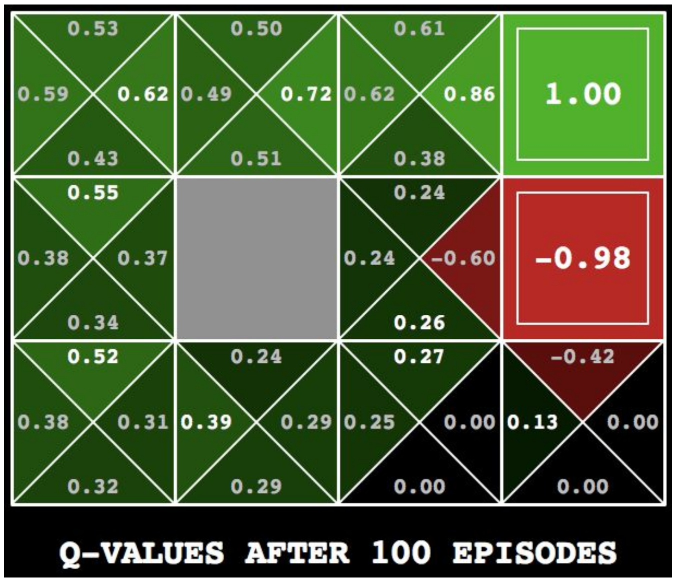
The Q Learning agent learns by trial and error from interactions with the environment using the update(state, action, nextState, reward) method. I used instance variables like epsilon, the exploration probability, alpha, the learning rate, and the discount rate. I wrote the functions computeValueFromQValues, computeActionFromQValues, getQValue, getAction, and update. In the computeValueFromQValues function, I initialized an action list and got the legal actions given the current state. If there were no legal actions, I'd return zero. If there were legal actions left, then I would loop though the possible actions. I would append the q value from each action and state pair to the list. Then I set the max value in the list to a variable q that was originally initialized to negative infinity. In each iteration, I compared the current q value for the current state and action pair. If it was larger than the list max variable q, then I would reset q to that value.

I used the same implementation in the function computeActionFromQValues. The only difference it that that function returns an action. So I had a variable cur_action to hold the best action. I looped through the actions just like in the previous function. In each iteration, I compared the current q value for the current state and action pair. If it was larger than the list max variable q, then I would reset q to that value and set max_action to the action of the current iteration. Once the best action is found, I would return that action.

The update function is only called on my behalf. The parent class calls update. First, I initialized the learning rate, alpha. Then I got the q value using getQValue(state, action). Then I set a value

Epsilon Greedy:

This method chooses random actions from any legal actions. The final q values were similar to the value iteration agent. However, the average returns were lower due to the random actions and the initial learning phase. The PacmanQAgent is initialized with the parameters epsilon = 0.5, gamma = 0.8, alpha = 0.2, and numTraining = 0. The prewritten function getAction calls QLearningAgent.getAction(self, state) and self.doAction(state, action). Then it returns the action for Pacman to the parent. I was also able to run the crawler robot at this stage of the project. I was able to change the values for step delay, discount, epsilon, and learning rate to see how it changed the speed, efficiency, and movement of the crawler.

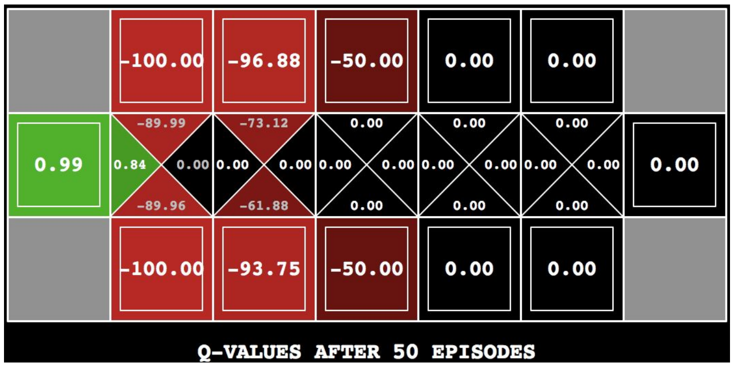


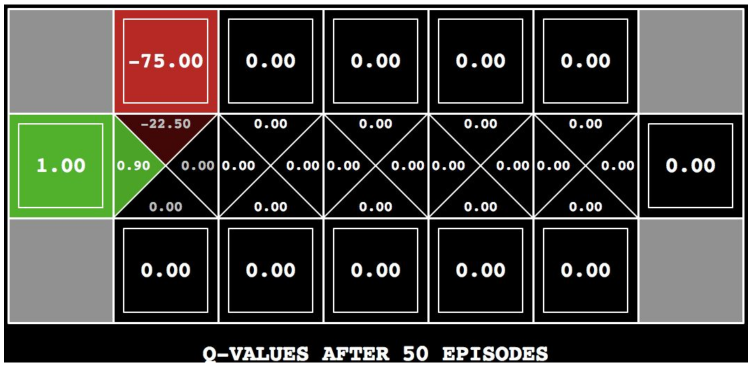
Bridge Crossing Revisited:

For this part of the project, I tried to find an epsilon and a learning rate that would help to learn an optimal policy after fifty iterations. By changing the exploration probability values from one to zero, the policy got worse as seen below. I also tried different epsilon and learning rate values and I was unable to find an optimal policy within fifty iterations. Thus, I had to set both values to None and I returned 'Not Possible.'

Epsilon: 1.0, Learning Rate: 0.5

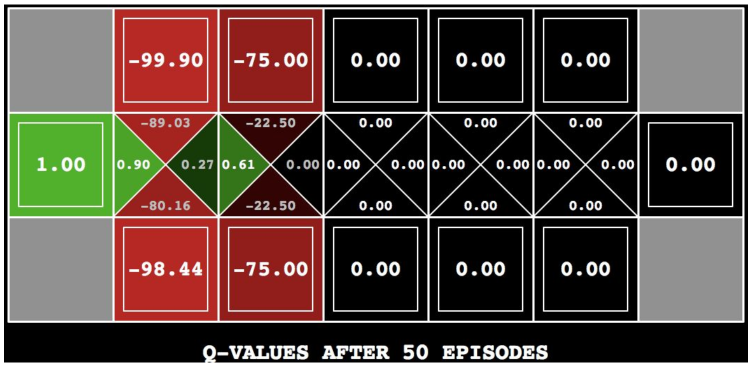
Epsilon: 0.0, Learning Rate: 0.5

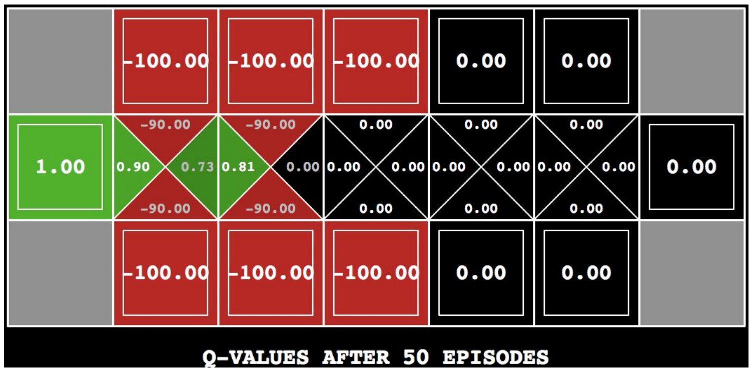




Epsilon: 0.5, Learning Rate: 0.5

Epsilon: 0.5, Learning Rate: 1.0





Q-Learning and Pacman:

For this part of the project, there is a training phase and a testing phase. During testing, epsilon and alpha are set to zero in order to exploit Pacman's learned policy. PacmanQAgent was already defined in QLearningAgent. It's default learning parameters epsilon is 0.05, alpha is 0.2, and gamma is 0.8. The training phase goes through 2000 training episodes. Then the autograder runs an additional 100 test games. My average rewards were 500 and each episode took approximately 2.5 seconds each. I had 100 wins, meaning I won all of the test games after training

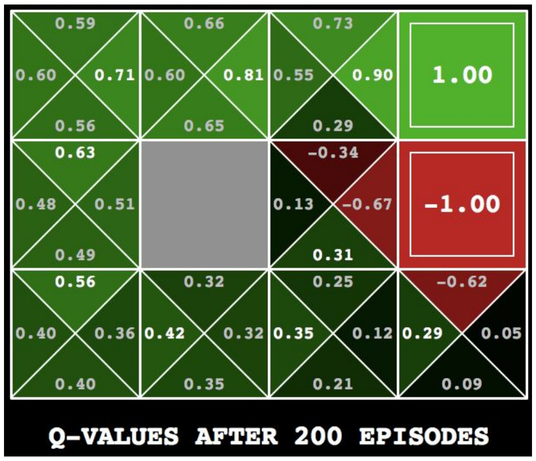
Approximate Q-Learning:

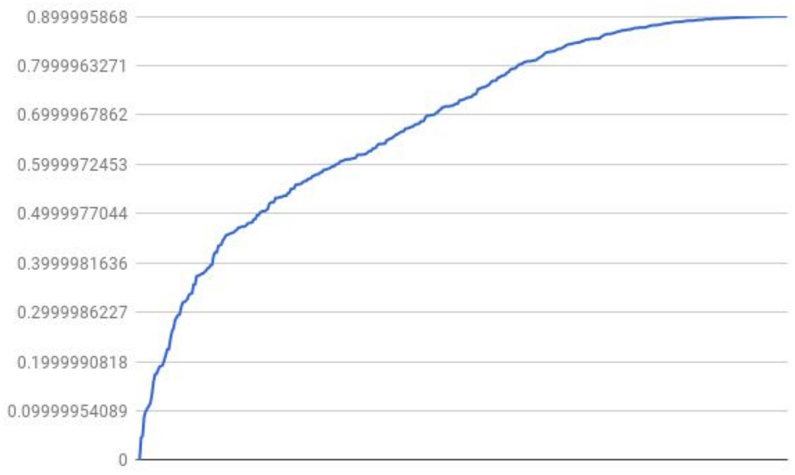
This class utilizes the QLearningAgent functions however it overwrites getQValue and update. The getQValue function uses self.featExtractor.getFeatures(state, action) to get the features and created a featureVector. The Q value is the product of the featureVector and the weights. The update function calls getQValue and saves the value in a variable called q_value. Then I added the reward to the product of the future state's value and the discount and saved it in a variable named val. I also subtracted the current q value from val. I looped through all of the features. For each feature I added the product of val, the feature's value, and the learning rate to the ith position of the weights vector.

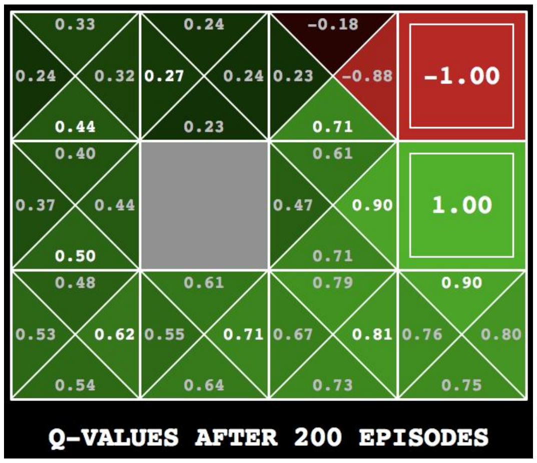
Solution Result/Evaluation:

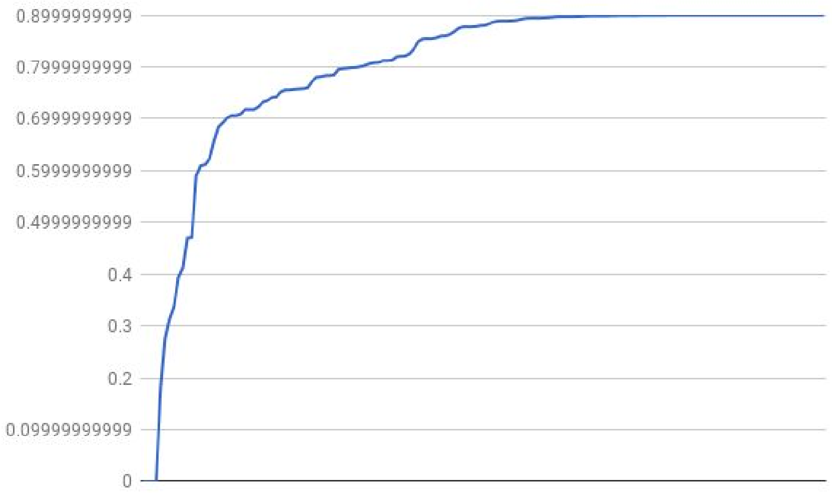
GridWorld Testing:

I changed the layout of getBookGrid as seen below and tested the convergence rates(as seen in the graphs.) I found that the formatted grid increased the convergence rate, as you can see in the corresponding graph(the line is steeper.)









Test_Cases Testing:

I added a test case to the test_cases file. Within q4 tests, I added a fifth test and corresponding solution called amelie.test and amelie.solution. The test runs correctly and can be seen in the video demo for this term project for question 4. I took a simple QLearningTest and modified the reward values to see if it would have an effect on the performance and return values. As expected the return values were lower, in direct response to the reward values being smaller.