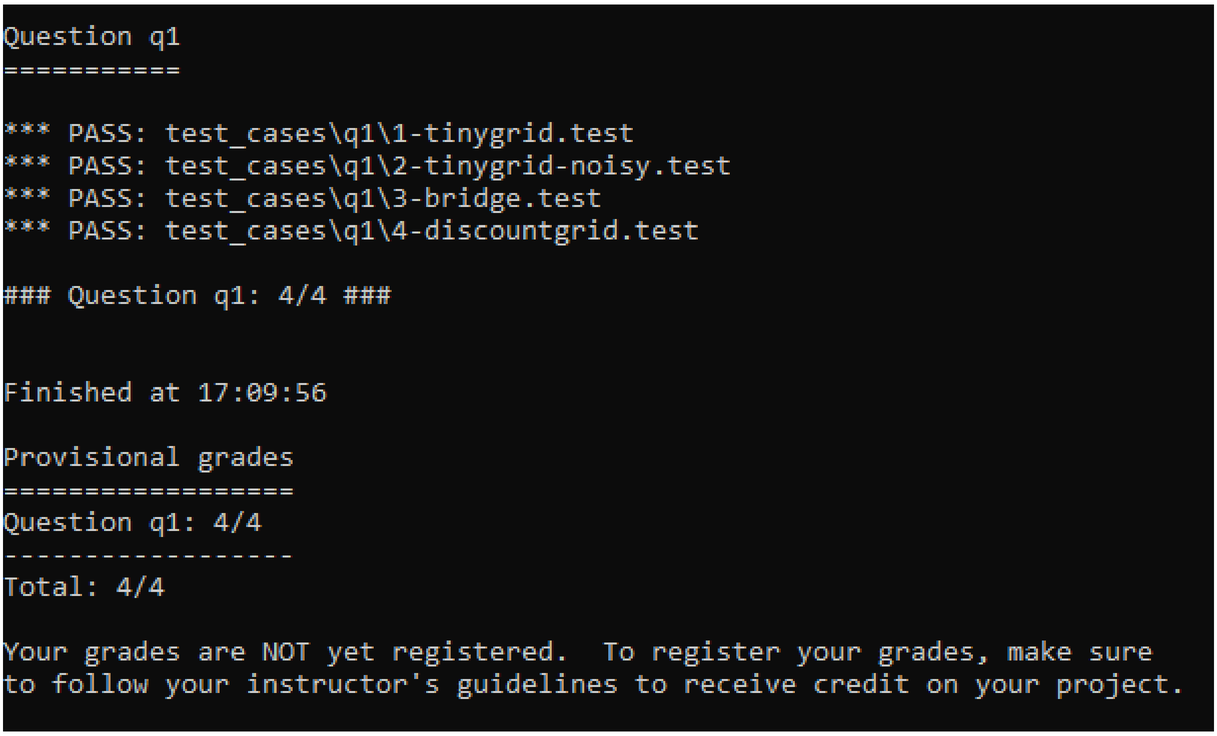
CSCI 166
Reinforcement Learning Assignment

John Eagan

October 2021

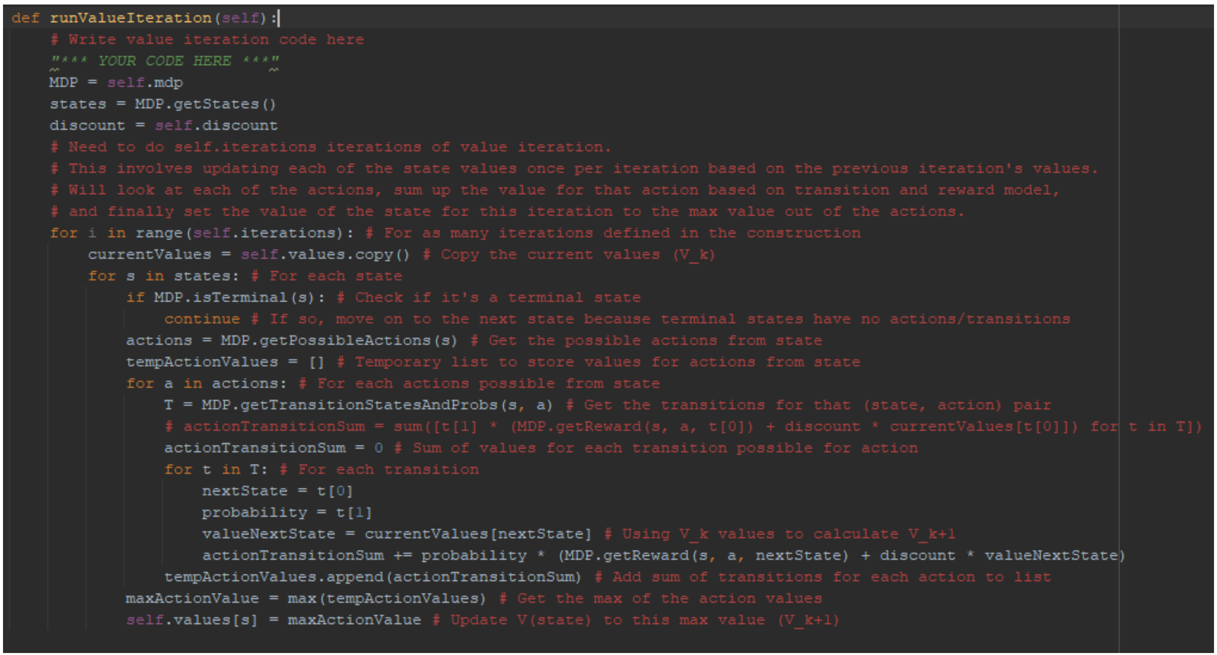
Question 1: Value Iteration

Autograder Results

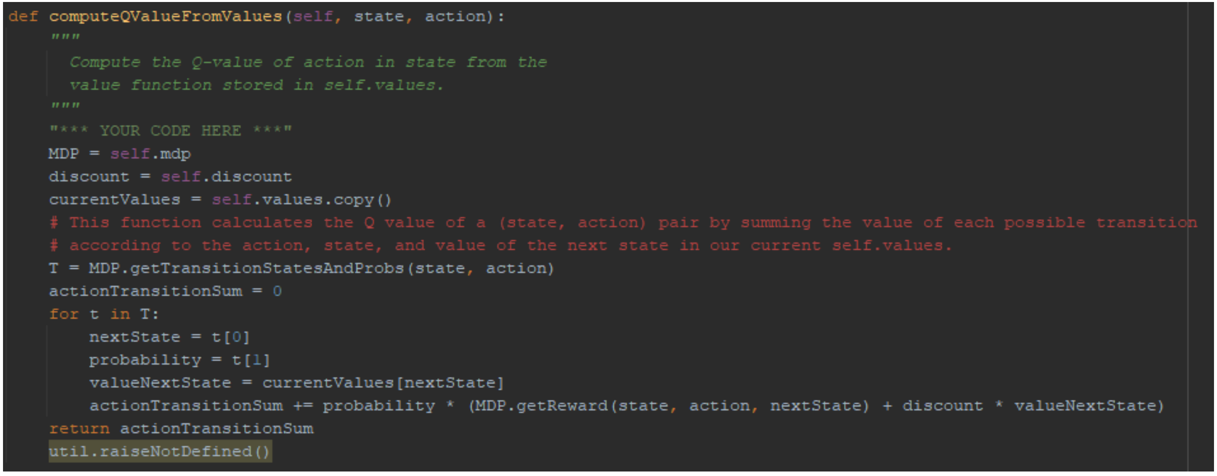


```
Question q1
===========

*** PASS: test_cases\q1\1-tinygrid.test
*** PASS: test_cases\q1\2-tinygrid-noisy.test
*** PASS: test_cases\q1\3-bridge.test
*** PASS: test_cases\q1\4-discountgrid.test

### Question q1: 4/4 ###


Finished at 17:09:56

Provisional grades
==================
Question q1: 4/4
------------------
Total: 4/4

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

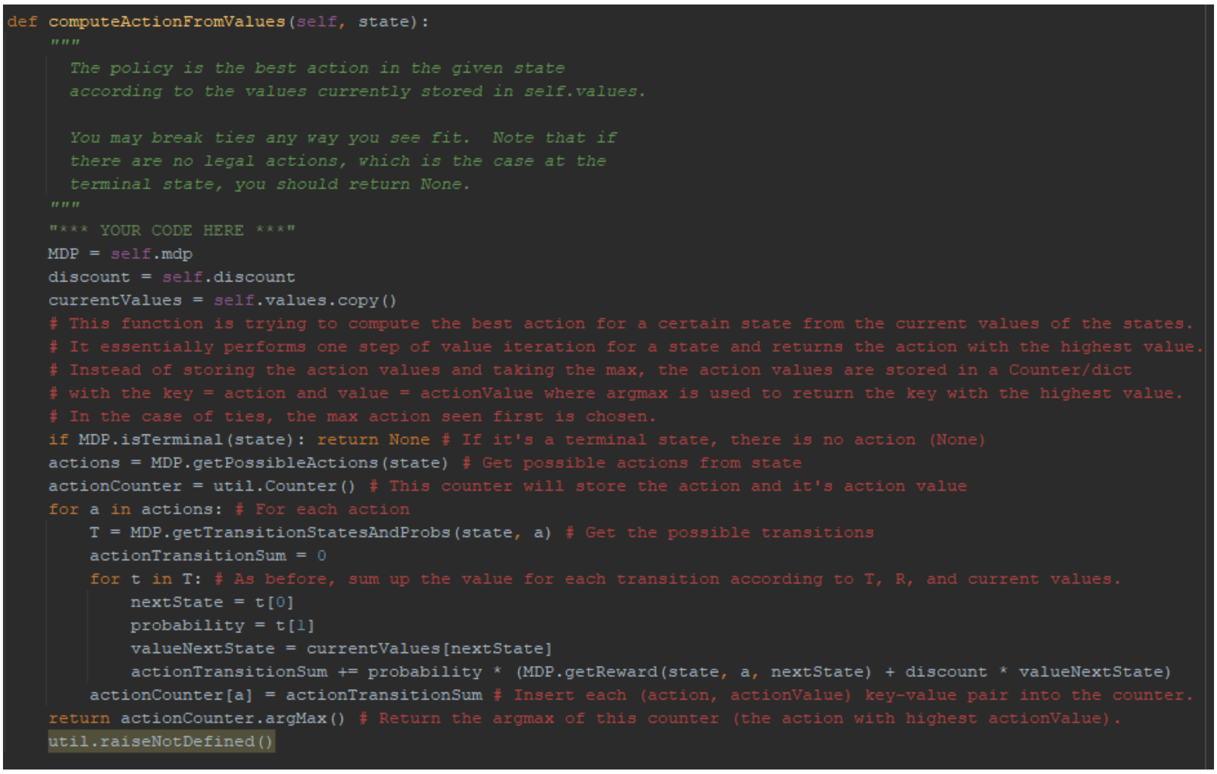
Code Snippet



```
def runValueIteration(self):
    # Write value iteration code here
    "*** YOUR CODE HERE ***"
    MDP = self.mdp
    states = MDP.getStates()
    discount = self.discount
    # Need to do self.iterations iterations of value iteration.
    # This involves updating each of the state values once per iteration based on the previous iteration's values.
    # Will look at each of the actions, sum up the value for that action based on transition and reward model,
    # and finally set the value of the state for this iteration to the max value out of the actions.
    for i in range(self.iterations): # For as many iterations defined in the construction
        currentValues = self.values.copy() # Copy the current values (V_k)
        for s in states: # For each state
            if MDP.isTerminal(s): # Check if it's a terminal state
                continue # If so, move on to the next state because terminal states have no actions/transitions
            actions = MDP.getPossibleActions(s) # Get the possible actions from state
            tempActionValues = [] # Temporary list to store values for actions from state
            for a in actions: # For each actions possible from state
                T = MDP.getTransitionStatesAndProbs(s, a) # Get the transitions for that (state, action) pair
                # actionTransitionSum = sum([t[1] * (MDP.getReward(s, a, t[0]) + discount * currentValues[t[0]]) for t in T])
                actionTransitionSum = 0 # Sum of values for each transition possible for action
                for t in T: # For each transition
                    nextState = t[0]
                    probability = t[1]
                    valueNextState = currentValues[nextState] # Using V_k values to calculate V_k+1
                    actionTransitionSum += probability * (MDP.getReward(s, a, nextState) + discount * valueNextState)
                tempActionValues.append(actionTransitionSum) # Add sum of transitions for each action to list
            maxActionValue = max(tempActionValues) # Get the max of the action values
            self.values[s] = maxActionValue # Update V(state) to this max value (V_k+1)
```



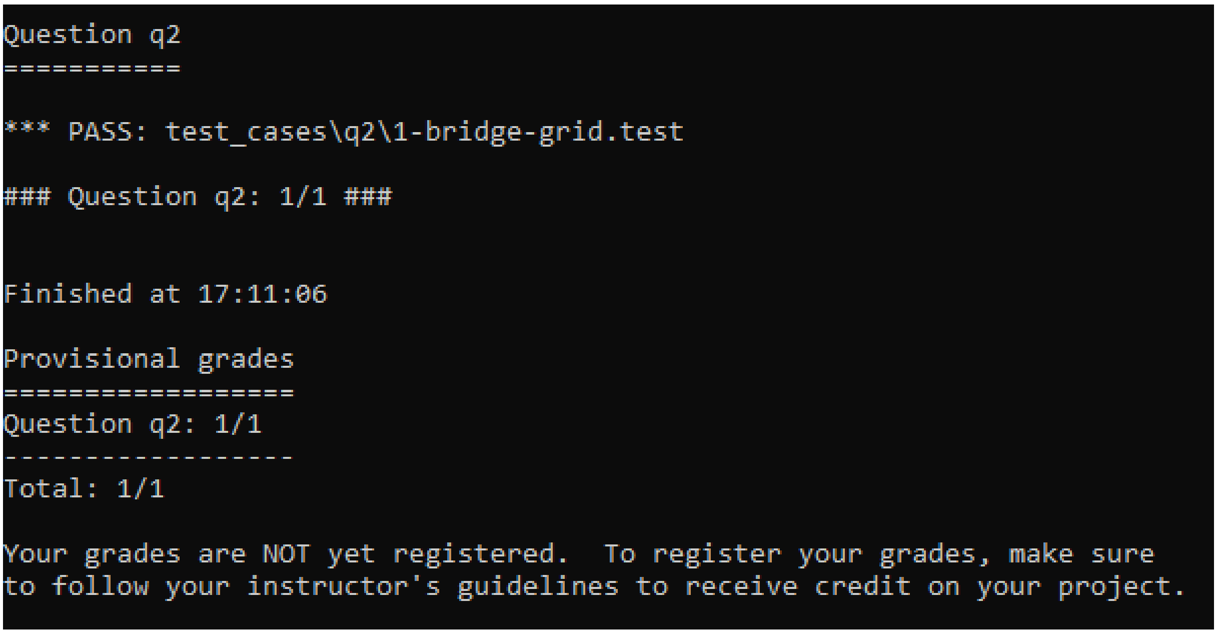
```
def computeQValueFromValues(self, state, action):
    """
      Compute the Q-value of action in state from the
      value function stored in self.values.
    """
    "*** YOUR CODE HERE ***"
    MDP = self.mdp
    discount = self.discount
    currentValues = self.values.copy()
    # This function calculates the Q value of a (state, action) pair by summing the value of each possible transition
    # according to the action, state, and value of the next state in our current self.values.
    T = MDP.getTransitionStatesAndProbs(state, action)
    actionTransitionSum = 0
    for t in T:
        nextState = t[0]
        probability = t[1]
        valueNextState = currentValues[nextState]
        actionTransitionSum += probability * (MDP.getReward(state, action, nextState) + discount * valueNextState)
    return actionTransitionSum
    util.raiseNotDefined()
```



```
def computeActionFromValues(self, state):
    """
      The policy is the best action in the given state
      according to the values currently stored in self.values.

      You may break ties any way you see fit.  Note that if
      there are no legal actions, which is the case at the
      terminal state, you should return None.
    """
    "*** YOUR CODE HERE ***"
    MDP = self.mdp
    discount = self.discount
    currentValues = self.values.copy()
    # This function is trying to compute the best action for a certain state from the current values of the states.
    # It essentially performs one step of value iteration for a state and returns the action with the highest value.
    # Instead of storing the action values and taking the max, the action values are stored in a Counter/dict
    # with the key = action and value = actionValue where argmax is used to return the key with the highest value.
    # In the case of ties, the max action seen first is chosen.
    if MDP.isTerminal(state): return None # If it's a terminal state, there is no action (None)
    actions = MDP.getPossibleActions(state) # Get possible actions from state
    actionCounter = util.Counter() # This counter will store the action and it's action value
    for a in actions: # For each action
        T = MDP.getTransitionStatesAndProbs(state, a) # Get the possible transitions
        actionTransitionSum = 0
        for t in T: # As before, sum up the value for each transition according to T, R, and current values.
            nextState = t[0]
            probability = t[1]
            valueNextState = currentValues[nextState]
            actionTransitionSum += probability * (MDP.getReward(state, a, nextState) + discount * valueNextState)
        actionCounter[a] = actionTransitionSum # Insert each (action, actionValue) key-value pair into the counter.
    return actionCounter.argMax() # Return the argmax of this counter (the action with highest actionValue).
    util.raiseNotDefined()
```

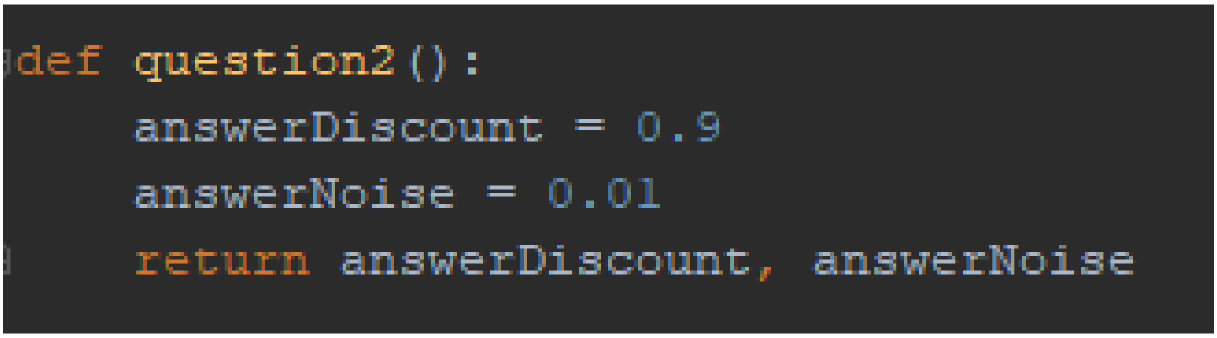
Question 2: Bridge Crossing Analysis

Autograder Results



```
Question q2
===========

*** PASS: test_cases\q2\1-bridge-grid.test

### Question q2: 1/1 ###


Finished at 17:11:06

Provisional grades
==================
Question q2: 1/1
------------------
Total: 1/1

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

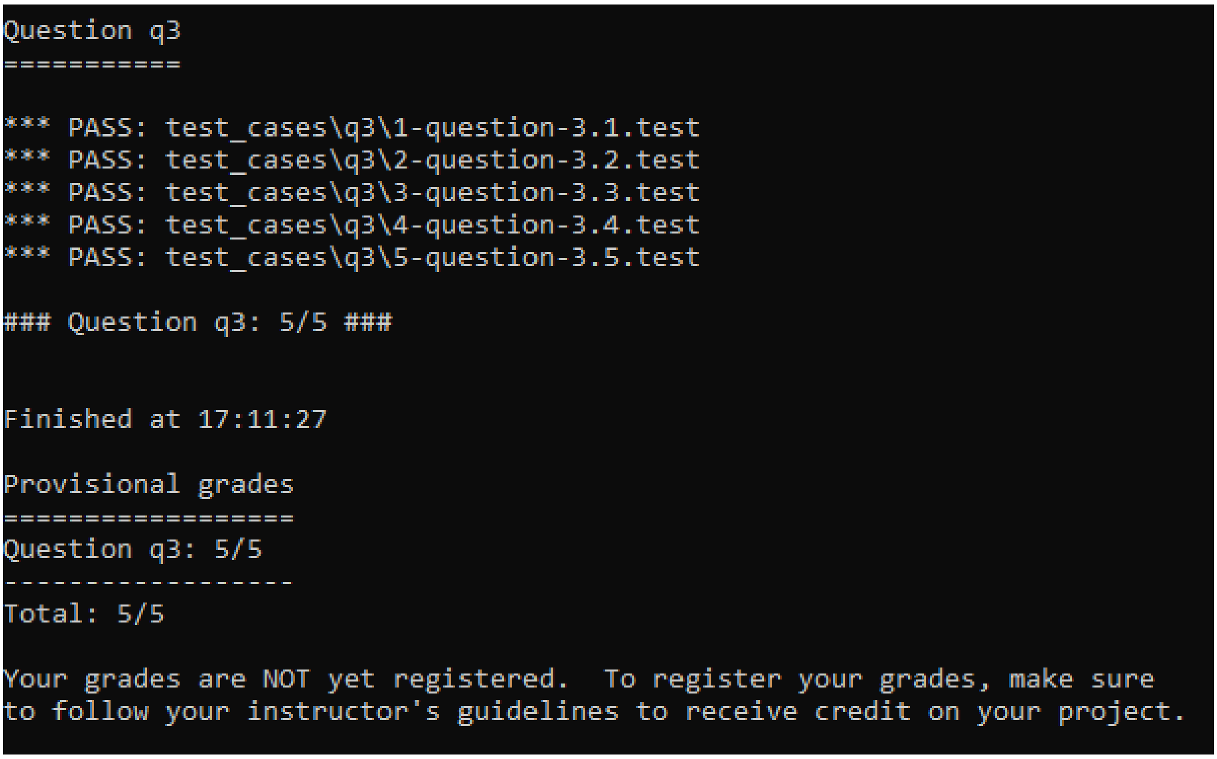
Code Snippet



```
def question2():
    answerDiscount = 0.9
    answerNoise = 0.01
    return answerDiscount, answerNoise
```

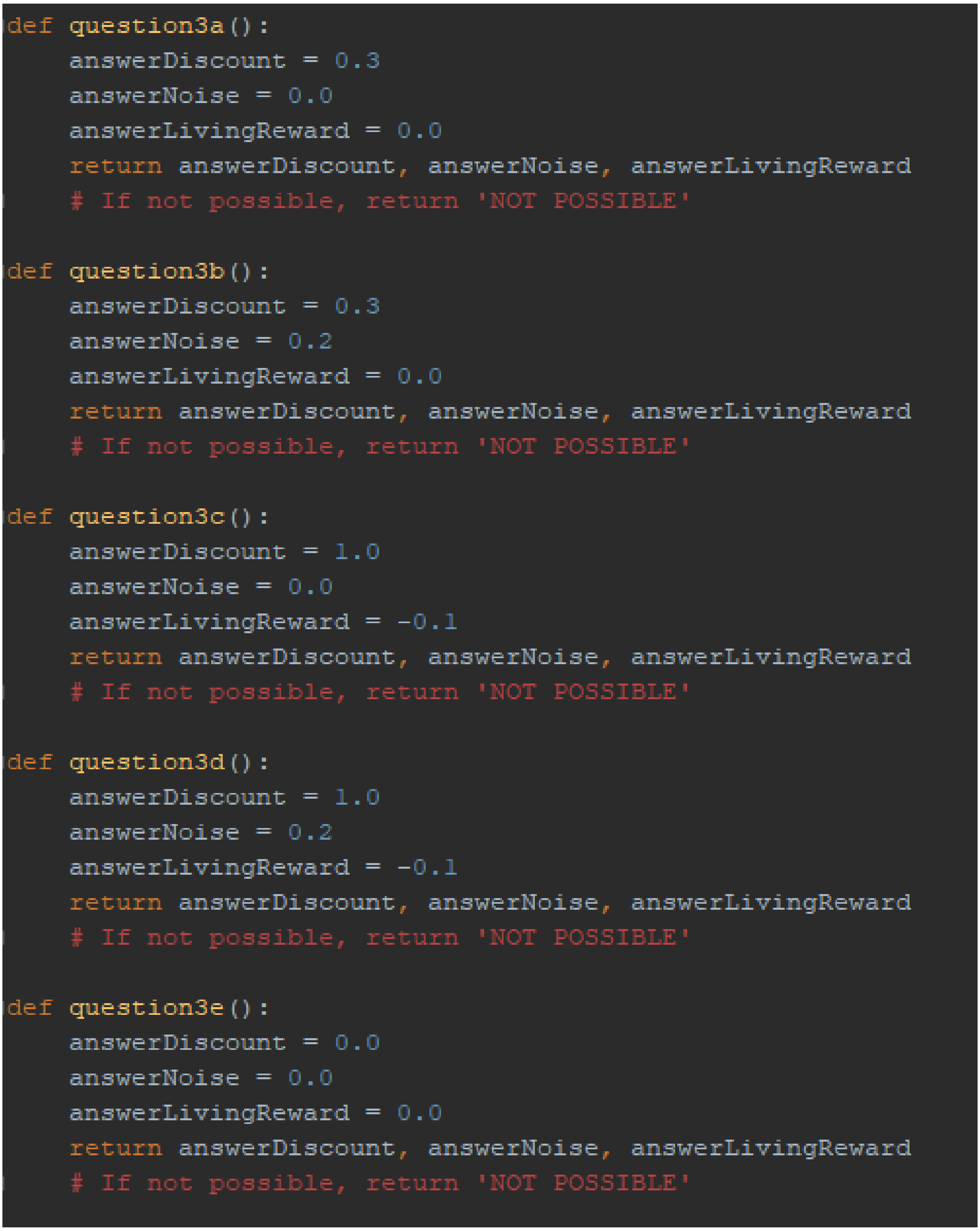
Question 3: Policies

Autograder Results



```
Question q3
===========

*** PASS: test_cases\q3\1-question-3.1.test
*** PASS: test_cases\q3\2-question-3.2.test
*** PASS: test_cases\q3\3-question-3.3.test
*** PASS: test_cases\q3\4-question-3.4.test
*** PASS: test_cases\q3\5-question-3.5.test

### Question q3: 5/5 ###


Finished at 17:11:27

Provisional grades
==================
Question q3: 5/5
------------------
Total: 5/5

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

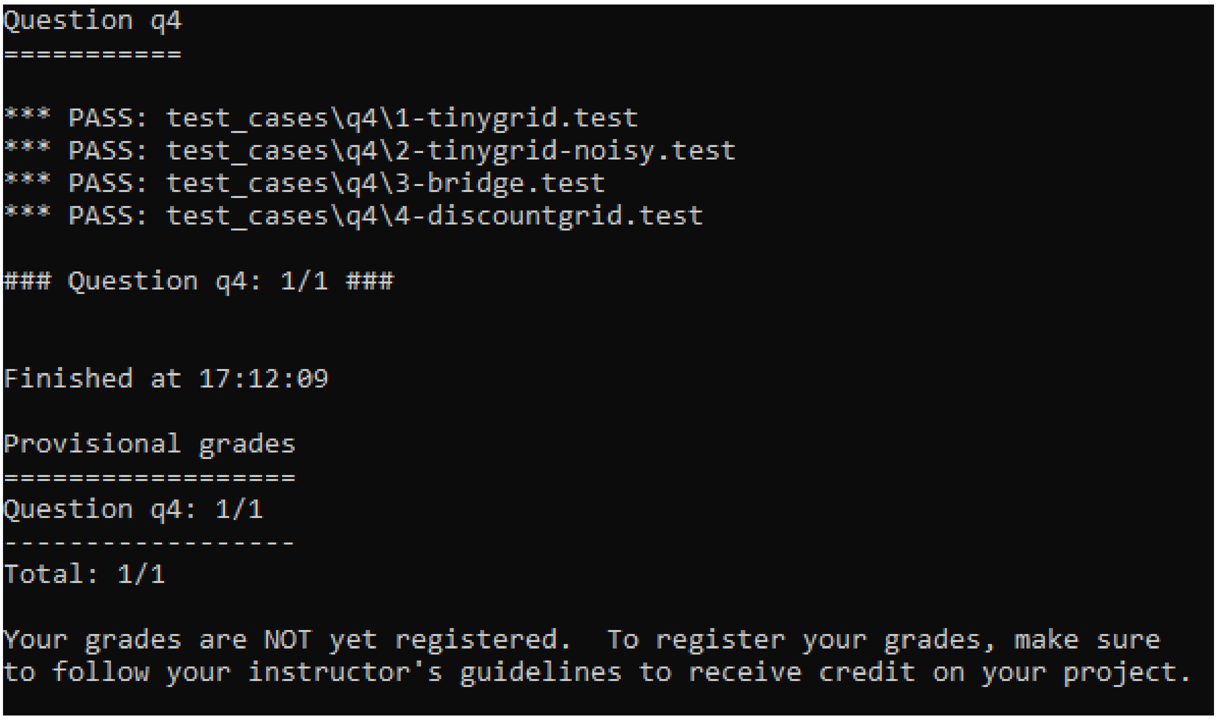
Code Snippet



```
def question3a():
    answerDiscount = 0.3
    answerNoise = 0.0
    answerLivingReward = 0.0
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'

def question3b():
    answerDiscount = 0.3
    answerNoise = 0.2
    answerLivingReward = 0.0
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'

def question3c():
    answerDiscount = 1.0
    answerNoise = 0.0
    answerLivingReward = -0.1
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'

def question3d():
    answerDiscount = 1.0
    answerNoise = 0.2
    answerLivingReward = -0.1
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'

def question3e():
    answerDiscount = 0.0
    answerNoise = 0.0
    answerLivingReward = 0.0
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

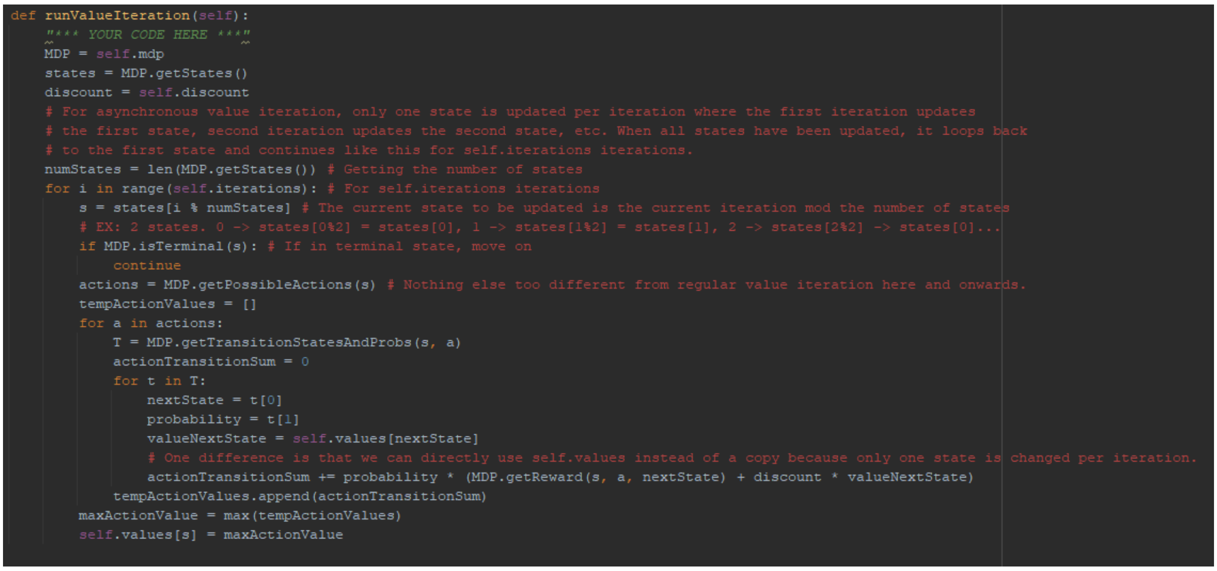
Question 4: Asynchronous Value Iteration

Autograder Results



```
Question q4
===========

*** PASS: test_cases\q4\1-tinygrid.test
*** PASS: test_cases\q4\2-tinygrid-noisy.test
*** PASS: test_cases\q4\3-bridge.test
*** PASS: test_cases\q4\4-discountgrid.test

### Question q4: 1/1 ###


Finished at 17:12:09

Provisional grades
==================
Question q4: 1/1
------------------
Total: 1/1

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

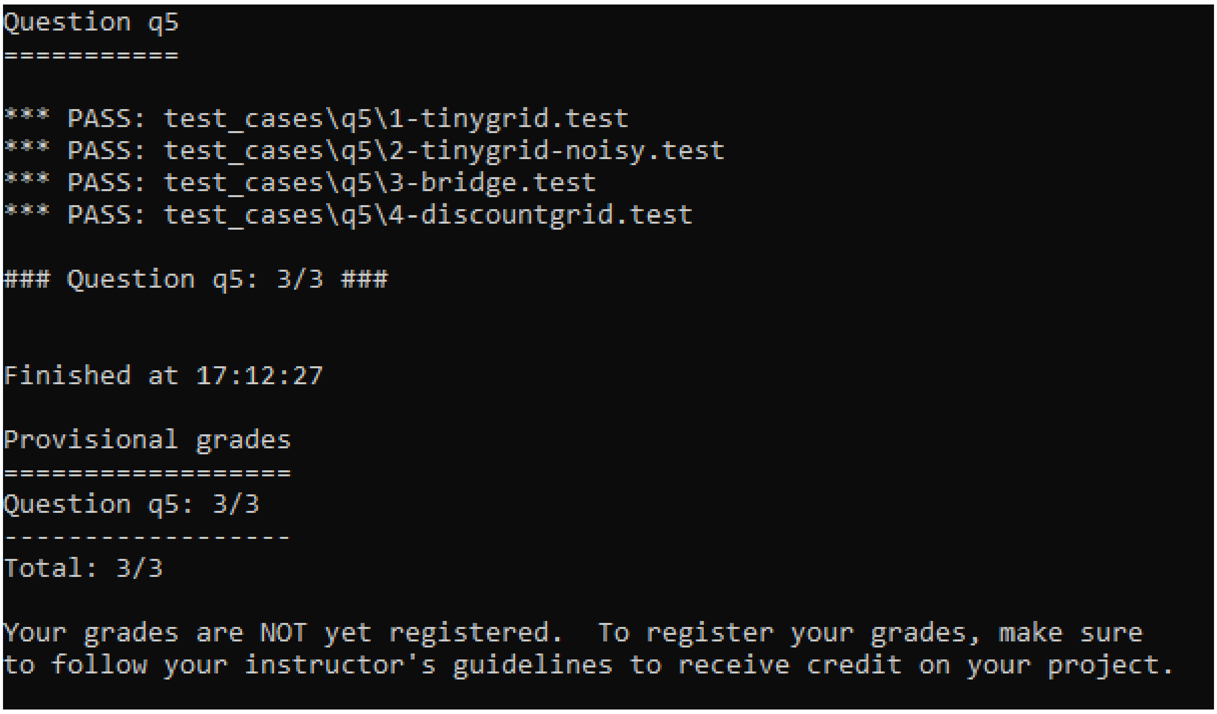
Code Snippet



```python
def runValueIteration(self):
    "*** YOUR CODE HERE ***"
    MDP = self.mdp
    states = MDP.getStates()
    discount = self.discount
    # For asynchronous value iteration, only one state is updated per iteration where the first iteration updates
    # the first state, second iteration updates the second state, etc. When all states have been updated, it loops back
    # to the first state and continues like this for self.iterations iterations.
    numStates = len(MDP.getStates()) # Getting the number of states
    for i in range(self.iterations): # For self.iterations iterations
        s = states[i % numStates] # The current state to be updated is the current iteration mod the number of states
        # EX: 2 states. 0 -> states[0%2] = states[0], 1 -> states[1%2] = states[1], 2 -> states[2%2] -> states[0]...
        if MDP.isTerminal(s): # If in terminal state, move on
            continue
        actions = MDP.getPossibleActions(s) # Nothing else too different from regular value iteration here and onwards.
        tempActionValues = []
        for a in actions:
            T = MDP.getTransitionStatesAndProbs(s, a)
            actionTransitionSum = 0
            for t in T:
                nextState = t[0]
                probability = t[1]
                valueNextState = self.values[nextState]
                # One difference is that we can directly use self.values instead of a copy because only one state is changed per iteration.
                actionTransitionSum += probability * (MDP.getReward(s, a, nextState) + discount * valueNextState)
            tempActionValues.append(actionTransitionSum)
        maxActionValue = max(tempActionValues)
        self.values[s] = maxActionValue
```

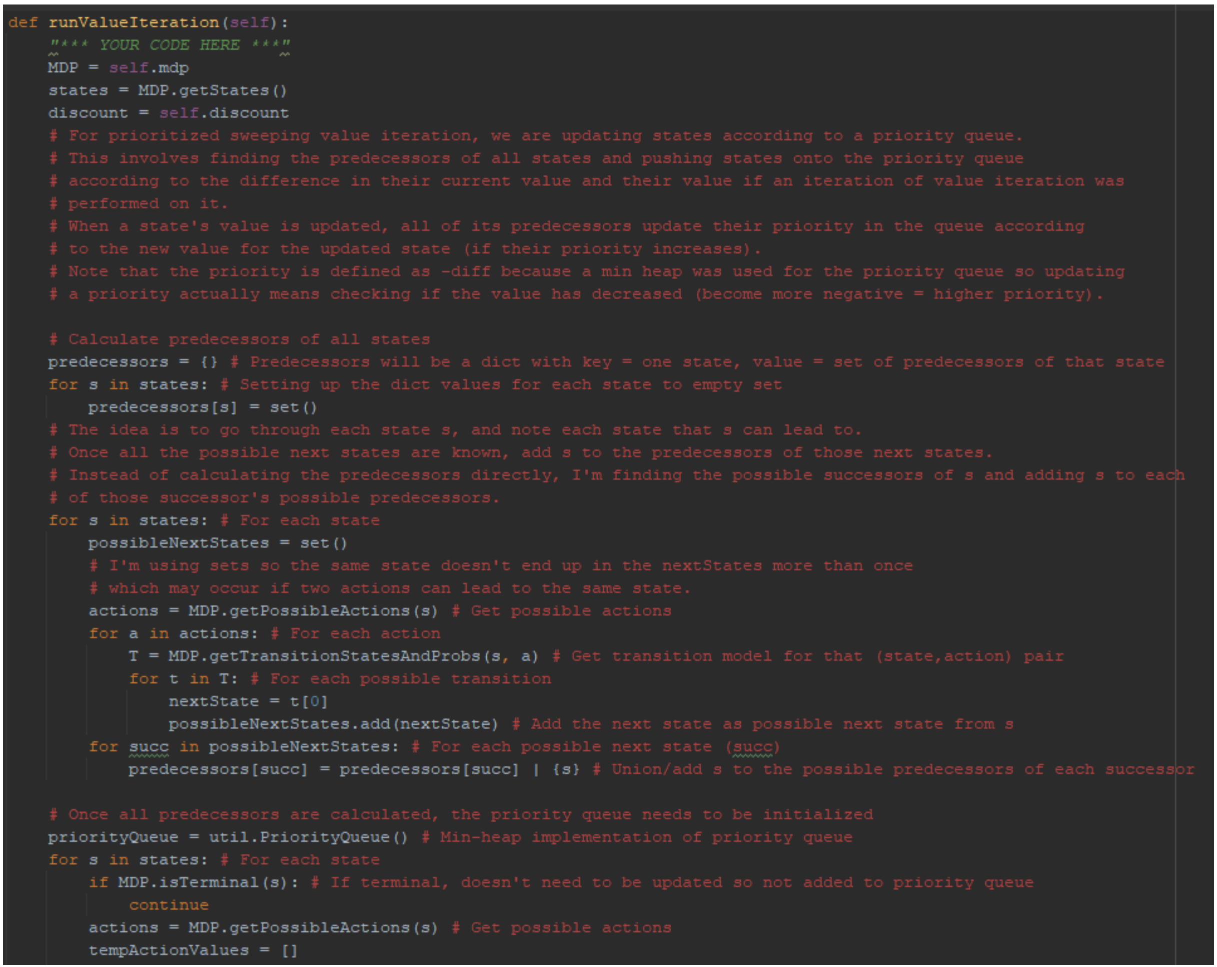
Question 5: Prioritized Sweeping Value Iteration

Autograder Results

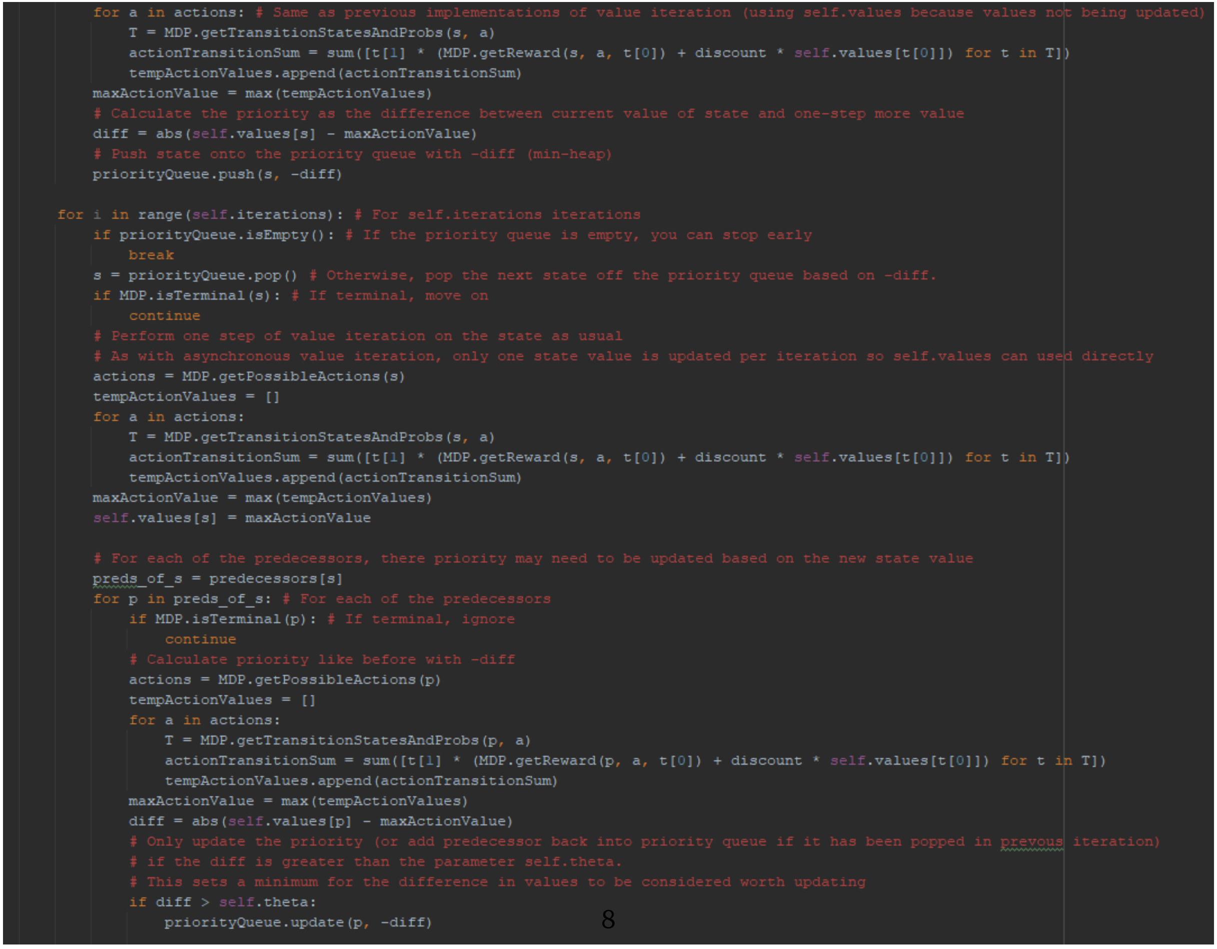


```
Question q5
===========

*** PASS: test_cases\q5\1-tinygrid.test
*** PASS: test_cases\q5\2-tinygrid-noisy.test
*** PASS: test_cases\q5\3-bridge.test
*** PASS: test_cases\q5\4-discountgrid.test

### Question q5: 3/3 ###


Finished at 17:12:27

Provisional grades
==================
Question q5: 3/3
------------------
Total: 3/3

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

Code Snippet



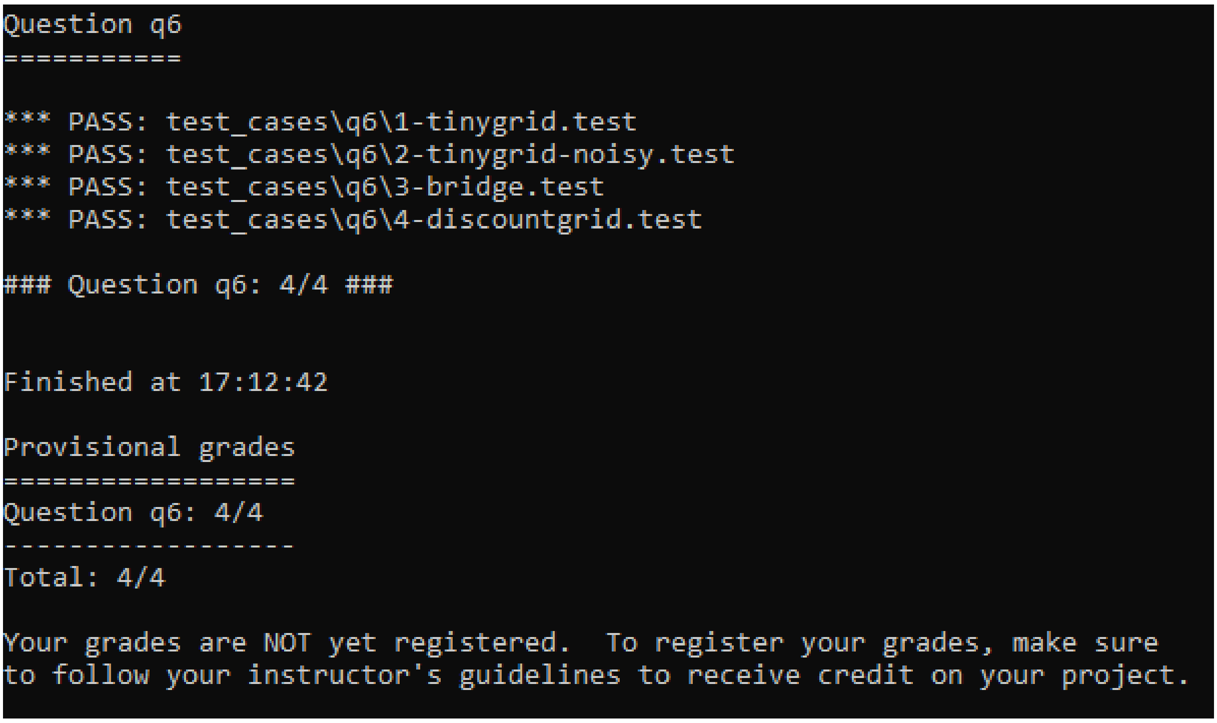
```
def runValueIteration(self):
    "*** YOUR CODE HERE ***"
    MDP = self.mdp
    states = MDP.getStates()
    discount = self.discount
    # For prioritized sweeping value iteration, we are updating states according to a priority queue.
    # This involves finding the predecessors of all states and pushing states onto the priority queue
    # according to the difference in their current value and their value if an iteration of value iteration was
    # performed on it.
    # When a state's value is updated, all of its predecessors update their priority in the queue according
    # to the new value for the updated state (if their priority increases).
    # Note that the priority is defined as -diff because a min heap was used for the priority queue so updating
    # a priority actually means checking if the value has decreased (become more negative = higher priority).

    # Calculate predecessors of all states
    predecessors = {} # Predecessors will be a dict with key = one state, value = set of predecessors of that state
    for s in states: # Setting up the dict values for each state to empty set
        predecessors[s] = set()
    # The idea is to go through each state s, and note each state that s can lead to.
    # Once all the possible next states are known, add s to the predecessors of those next states.
    # Instead of calculating the predecessors directly, I'm finding the possible successors of s and adding s to each
    # of those successor's possible predecessors.
    for s in states: # For each state
        possibleNextStates = set()
        # I'm using sets so the same state doesn't end up in the nextStates more than once
        # which may occur if two actions can lead to the same state.
        actions = MDP.getPossibleActions(s) # Get possible actions
        for a in actions: # For each action
            T = MDP.getTransitionStatesAndProbs(s, a) # Get transition model for that (state,action) pair
            for t in T: # For each possible transition
                nextState = t[0]
                possibleNextStates.add(nextState) # Add the next state as possible next state from s
        for succ in possibleNextStates: # For each possible next state (succ)
            predecessors[succ] = predecessors[succ] | {s} # Union/add s to the possible predecessors of each successor

    # Once all predecessors are calculated, the priority queue needs to be initialized
    priorityQueue = util.PriorityQueue() # Min-heap implementation of priority queue
    for s in states: # For each state
        if MDP.isTerminal(s): # If terminal, doesn't need to be updated so not added to priority queue
            continue
        actions = MDP.getPossibleActions(s) # Get possible actions
        tempActionValues = []
```



```
        for a in actions: # Same as previous implementations of value iteration (using self.values because values not being updated)
            T = MDP.getTransitionStatesAndProbs(s, a)
            actionTransitionSum = sum([t[1] * (MDP.getReward(s, a, t[0]) + discount * self.values[t[0]]) for t in T])
            tempActionValues.append(actionTransitionSum)
        maxActionValue = max(tempActionValues)
        # Calculate the priority as the difference between current value of state and one-step more value
        diff = abs(self.values[s] - maxActionValue)
        # Push state onto the priority queue with -diff (min-heap)
        priorityQueue.push(s, -diff)

    for i in range(self.iterations): # For self.iterations iterations
        if priorityQueue.isEmpty(): # If the priority queue is empty, you can stop early
            break
        s = priorityQueue.pop() # Otherwise, pop the next state off the priority queue based on -diff.
        if MDP.isTerminal(s): # If terminal, move on
            continue
        # Perform one step of value iteration on the state as usual
        # As with asynchronous value iteration, only one state value is updated per iteration so self.values can used directly
        actions = MDP.getPossibleActions(s)
        tempActionValues = []
        for a in actions:
            T = MDP.getTransitionStatesAndProbs(s, a)
            actionTransitionSum = sum([t[1] * (MDP.getReward(s, a, t[0]) + discount * self.values[t[0]]) for t in T])
            tempActionValues.append(actionTransitionSum)
        maxActionValue = max(tempActionValues)
        self.values[s] = maxActionValue

        # For each of the predecessors, there priority may need to be updated based on the new state value
        preds_of_s = predecessors[s]
        for p in preds_of_s: # For each of the predecessors
            if MDP.isTerminal(p): # If terminal, ignore
                continue
            # Calculate priority like before with -diff
            actions = MDP.getPossibleActions(p)
            tempActionValues = []
            for a in actions:
                T = MDP.getTransitionStatesAndProbs(p, a)
                actionTransitionSum = sum([t[1] * (MDP.getReward(p, a, t[0]) + discount * self.values[t[0]]) for t in T])
                tempActionValues.append(actionTransitionSum)
            maxActionValue = max(tempActionValues)
            diff = abs(self.values[p] - maxActionValue)
            # Only update the priority (or add predecessor back into priority queue if it has been popped in prevous iteration)
            # if the diff is greater than the parameter self.theta.
            # This sets a minimum for the difference in values to be considered worth updating
            if diff > self.theta:
                priorityQueue.update(p, -diff)
```

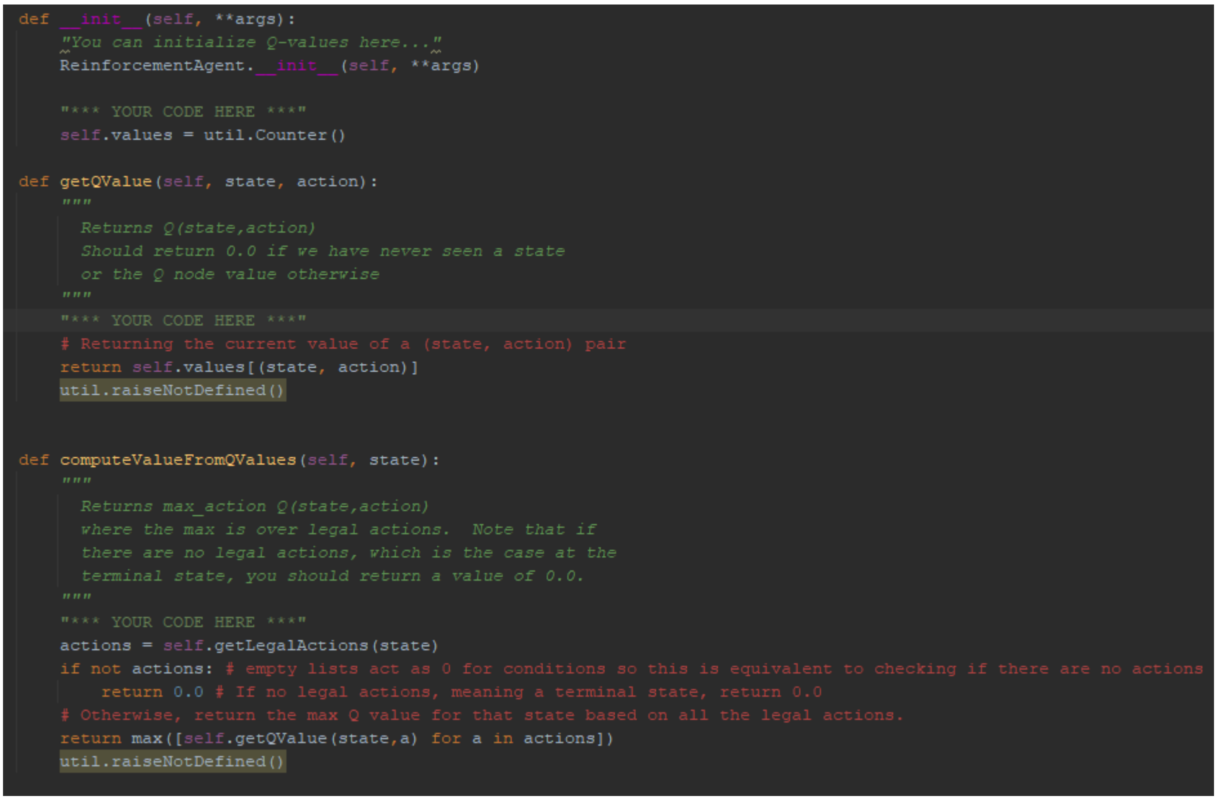
Question 6: Q-Learning

Autograder Results

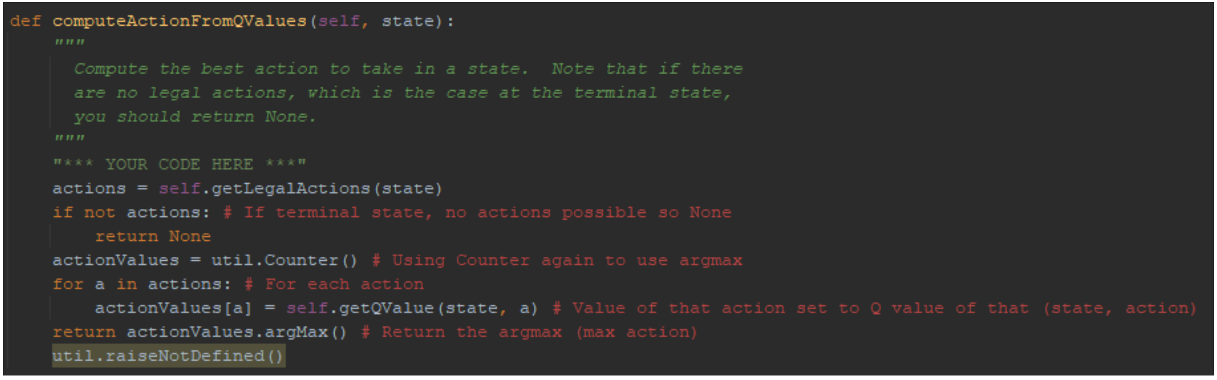


```
Question q6
===========

*** PASS: test_cases\q6\1-tinygrid.test
*** PASS: test_cases\q6\2-tinygrid-noisy.test
*** PASS: test_cases\q6\3-bridge.test
*** PASS: test_cases\q6\4-discountgrid.test

### Question q6: 4/4 ###


Finished at 17:12:42

Provisional grades
==================
Question q6: 4/4
------------------
Total: 4/4

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

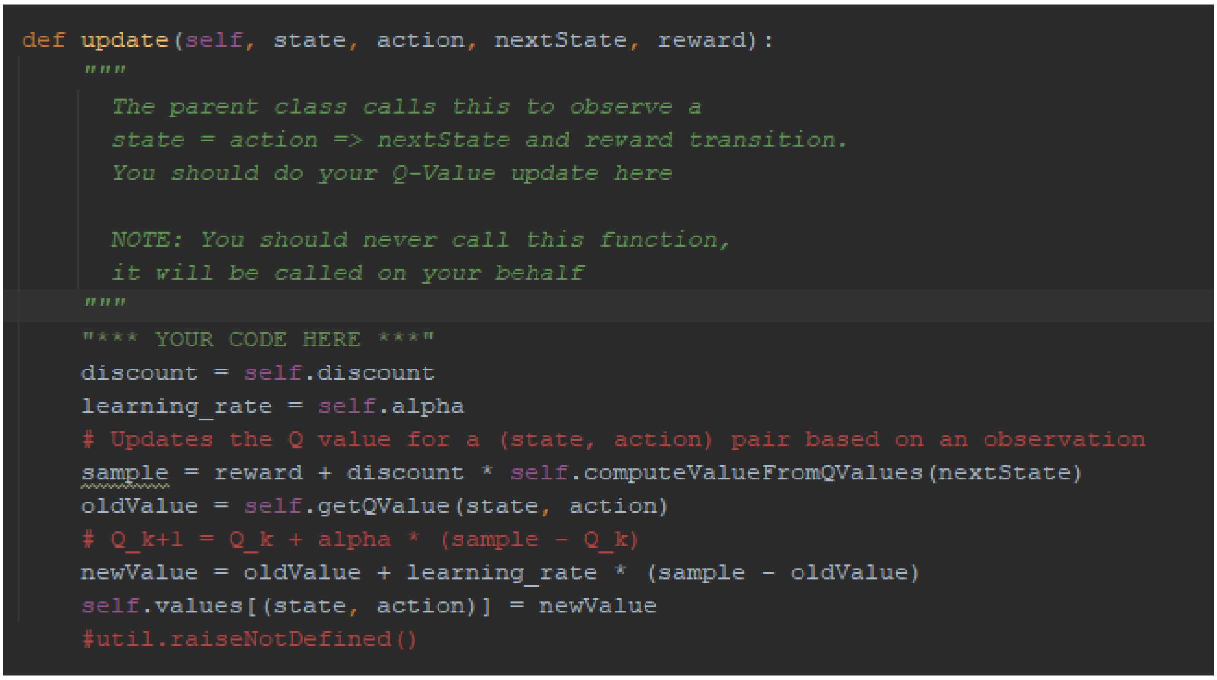
Code Snippet



```
def __init__(self, **args):
    "You can initialize Q-values here..."
    ReinforcementAgent.__init__(self, **args)

    "*** YOUR CODE HERE ***"
    self.values = util.Counter()

def getQValue(self, state, action):
    """
      Returns Q(state,action)
      Should return 0.0 if we have never seen a state
      or the Q node value otherwise
    """
    "*** YOUR CODE HERE ***"
    # Returning the current value of a (state, action) pair
    return self.values[(state, action)]
    util.raiseNotDefined()


def computeValueFromQValues(self, state):
    """
      Returns max_action Q(state,action)
      where the max is over legal actions.  Note that if
      there are no legal actions, which is the case at the
      terminal state, you should return a value of 0.0.
    """
    "*** YOUR CODE HERE ***"
    actions = self.getLegalActions(state)
    if not actions: # empty lists act as 0 for conditions so this is equivalent to checking if there are no actions
        return 0.0 # If no legal actions, meaning a terminal state, return 0.0
    # Otherwise, return the max Q value for that state based on all the legal actions.
    return max([self.getQValue(state,a) for a in actions])
    util.raiseNotDefined()
```



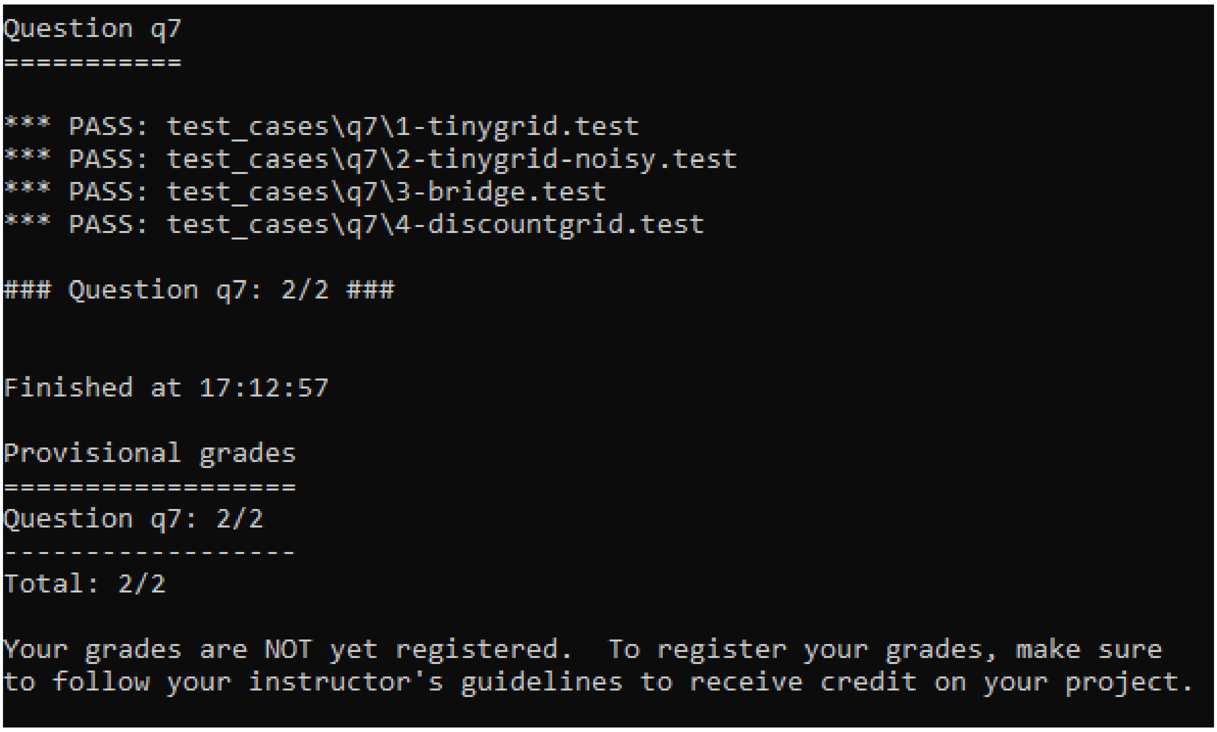
```
def computeActionFromQValues(self, state):
    """
      Compute the best action to take in a state.  Note that if there
      are no legal actions, which is the case at the terminal state,
      you should return None.
    """
    "*** YOUR CODE HERE ***"
    actions = self.getLegalActions(state)
    if not actions: # If terminal state, no actions possible so None
        return None
    actionValues = util.Counter() # Using Counter again to use argmax
    for a in actions: # For each action
        actionValues[a] = self.getQValue(state, a) # Value of that action set to Q value of that (state, action)
    return actionValues.argMax() # Return the argmax (max action)
    util.raiseNotDefined()
```



```
def update(self, state, action, nextState, reward):
    """
      The parent class calls this to observe a
      state = action => nextState and reward transition.
      You should do your Q-Value update here

      NOTE: You should never call this function,
      it will be called on your behalf
    """
    "*** YOUR CODE HERE ***"
    discount = self.discount
    learning_rate = self.alpha
    # Updates the Q value for a (state, action) pair based on an observation
    sample = reward + discount * self.computeValueFromQValues(nextState)
    oldValue = self.getQValue(state, action)
    # Q_k+1 = Q_k + alpha * (sample - Q_k)
    newValue = oldValue + learning_rate * (sample - oldValue)
    self.values[(state, action)] = newValue
    #util.raiseNotDefined()
```

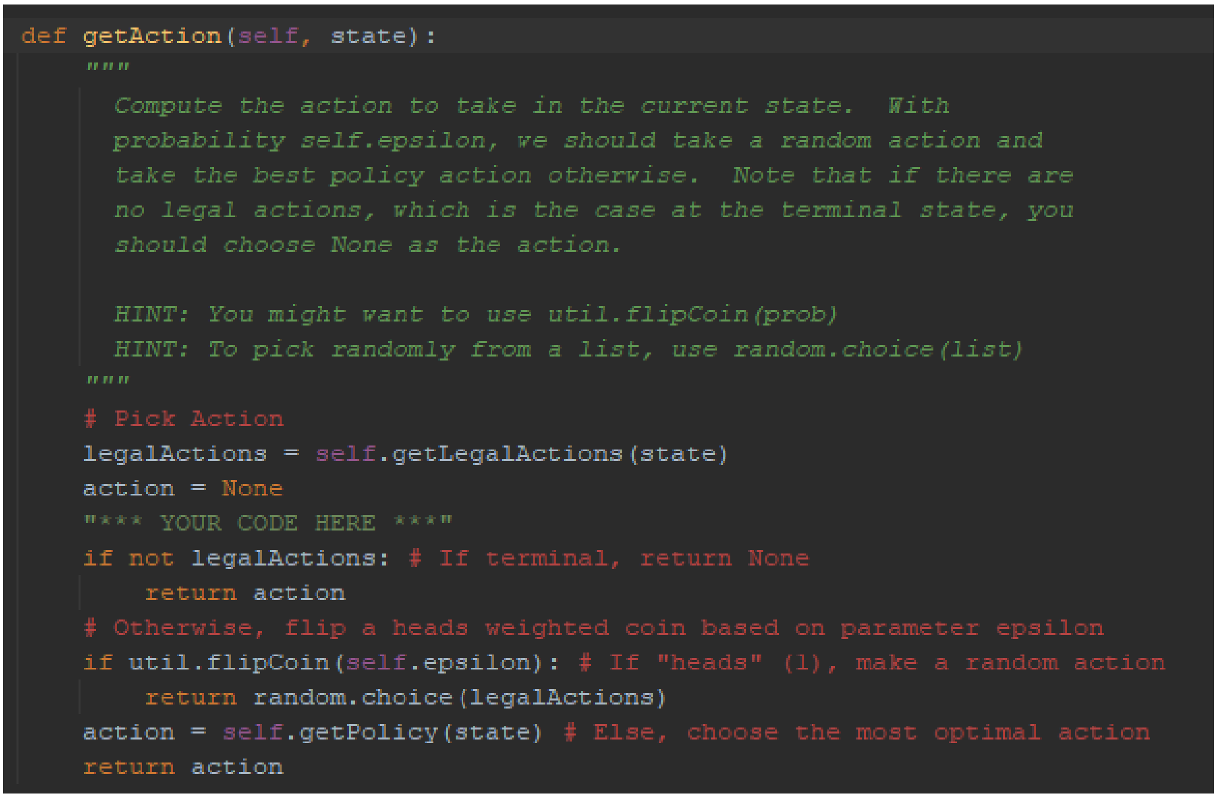
Question 7: Epsilon Greedy

Autograder Results



```
Question q7
===========

*** PASS: test_cases\q7\1-tinygrid.test
*** PASS: test_cases\q7\2-tinygrid-noisy.test
*** PASS: test_cases\q7\3-bridge.test
*** PASS: test_cases\q7\4-discountgrid.test

### Question q7: 2/2 ###


Finished at 17:12:57

Provisional grades
==================
Question q7: 2/2
------------------
Total: 2/2

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

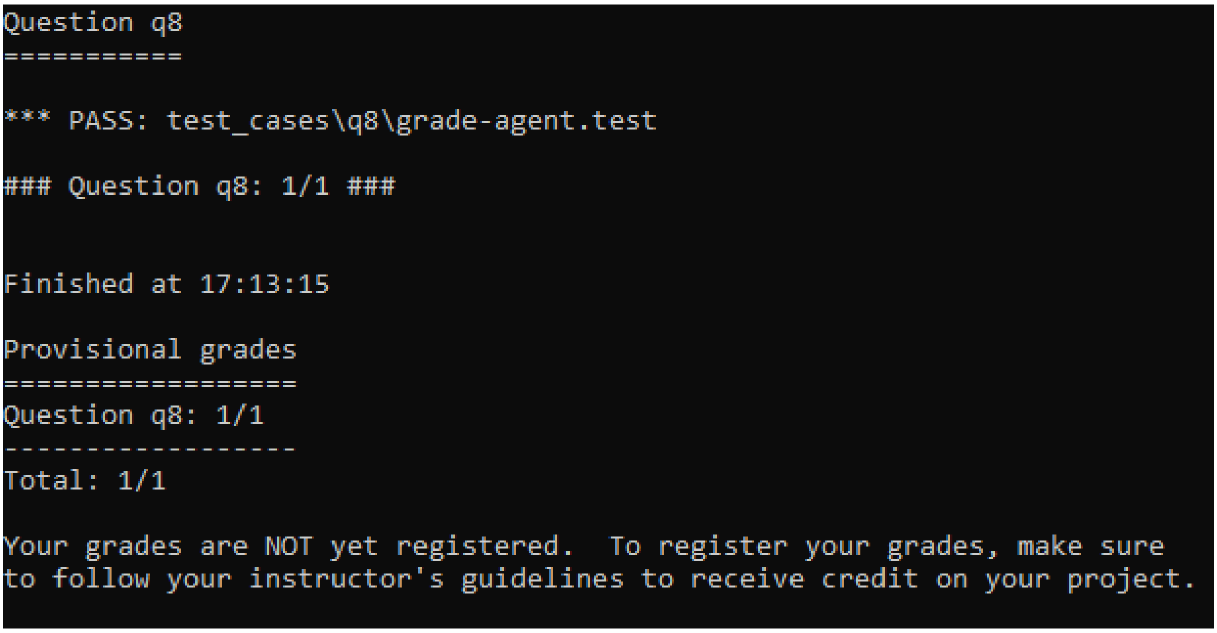
Code Snippet



```
def getAction(self, state):
    """
      Compute the action to take in the current state.  With
      probability self.epsilon, we should take a random action and
      take the best policy action otherwise.  Note that if there are
      no legal actions, which is the case at the terminal state, you
      should choose None as the action.

      HINT: You might want to use util.flipCoin(prob)
      HINT: To pick randomly from a list, use random.choice(list)
    """
    # Pick Action
    legalActions = self.getLegalActions(state)
    action = None
    "*** YOUR CODE HERE ***"
    if not legalActions: # If terminal, return None
        return action
    # Otherwise, flip a heads weighted coin based on parameter epsilon
    if util.flipCoin(self.epsilon): # If "heads" (1), make a random action
        return random.choice(legalActions)
    action = self.getPolicy(state) # Else, choose the most optimal action
    return action
```

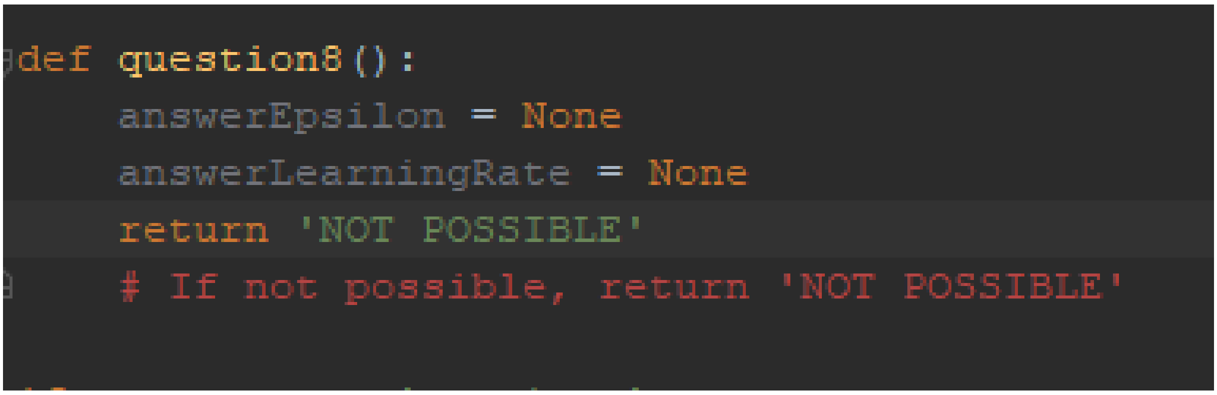
Question 8: Bridge Crossing Revisited

Autograder Results



```
Question q8
===========

*** PASS: test_cases\q8\grade-agent.test

### Question q8: 1/1 ###


Finished at 17:13:15

Provisional grades
==================
Question q8: 1/1
------------------
Total: 1/1

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

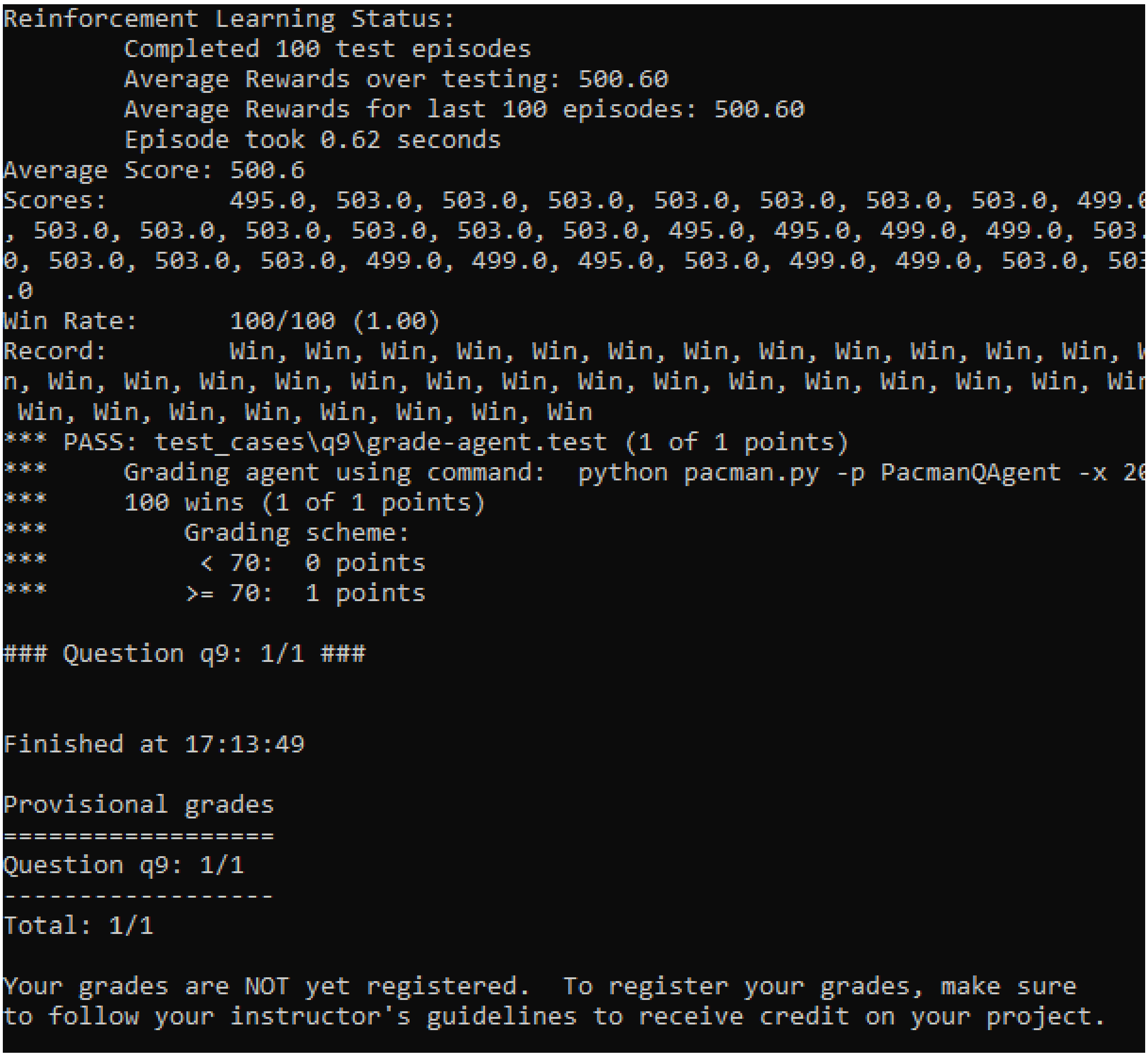
Code Snippet



```
def question8():
    answerEpsilon = None
    answerLearningRate = None
    return 'NOT POSSIBLE'
    # If not possible, return 'NOT POSSIBLE'
```

Question 9: Q-Learning and Pacman

Autograder Results



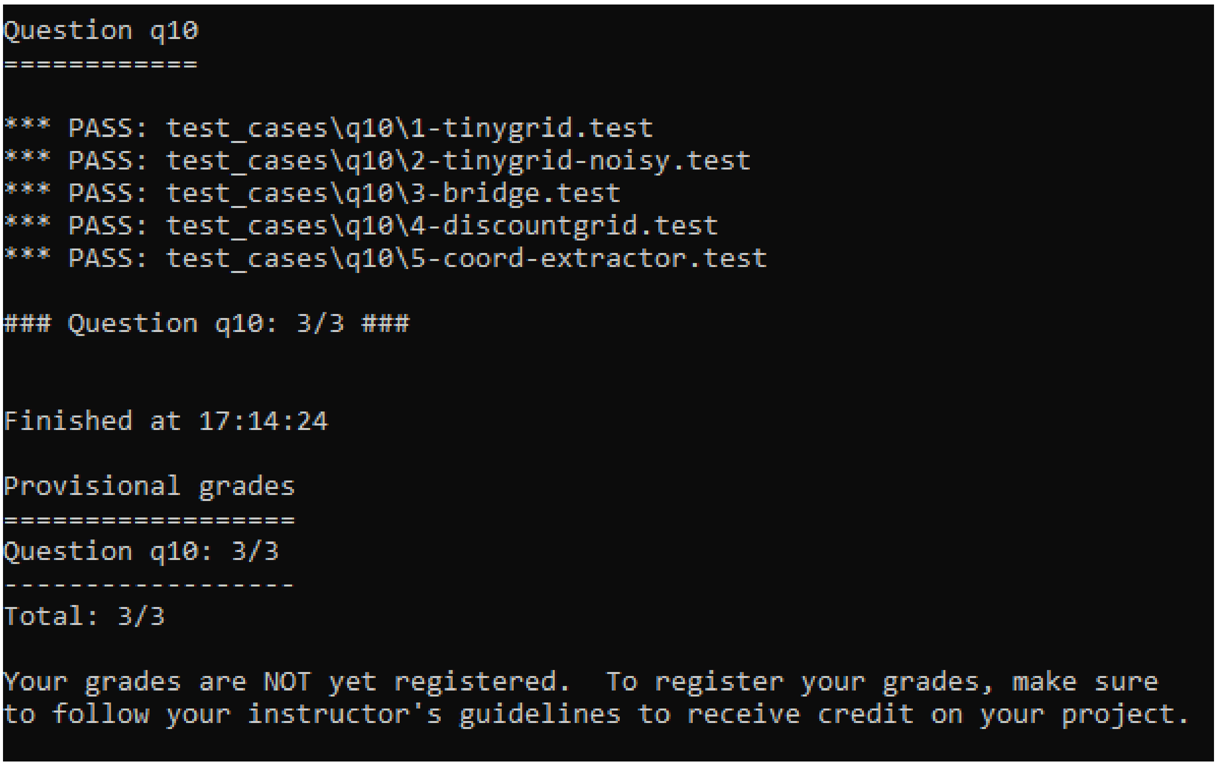
```
Reinforcement Learning Status:
        Completed 100 test episodes
        Average Rewards over testing: 500.60
        Average Rewards for last 100 episodes: 500.60
        Episode took 0.62 seconds
Average Score: 500.6
Scores:        495.0, 503.0, 503.0, 503.0, 503.0, 503.0, 503.0, 503.0, 499.0
, 503.0, 503.0, 503.0, 503.0, 503.0, 503.0, 495.0, 495.0, 499.0, 499.0, 503.
0, 503.0, 503.0, 503.0, 499.0, 499.0, 495.0, 503.0, 499.0, 499.0, 503.0, 503
.0
Win Rate:      100/100 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, W
n, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
 Win, Win, Win, Win, Win, Win, Win, Win
*** PASS: test_cases\q9\grade-agent.test (1 of 1 points)
***     Grading agent using command:  python pacman.py -p PacmanQAgent -x 20
***     100 wins (1 of 1 points)
***         Grading scheme:
***          < 70:  0 points
***         >= 70:  1 points

### Question q9: 1/1 ###


Finished at 17:13:49

Provisional grades
==================
Question q9: 1/1
------------------
Total: 1/1

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

Code Snippet

No coding needed for this question

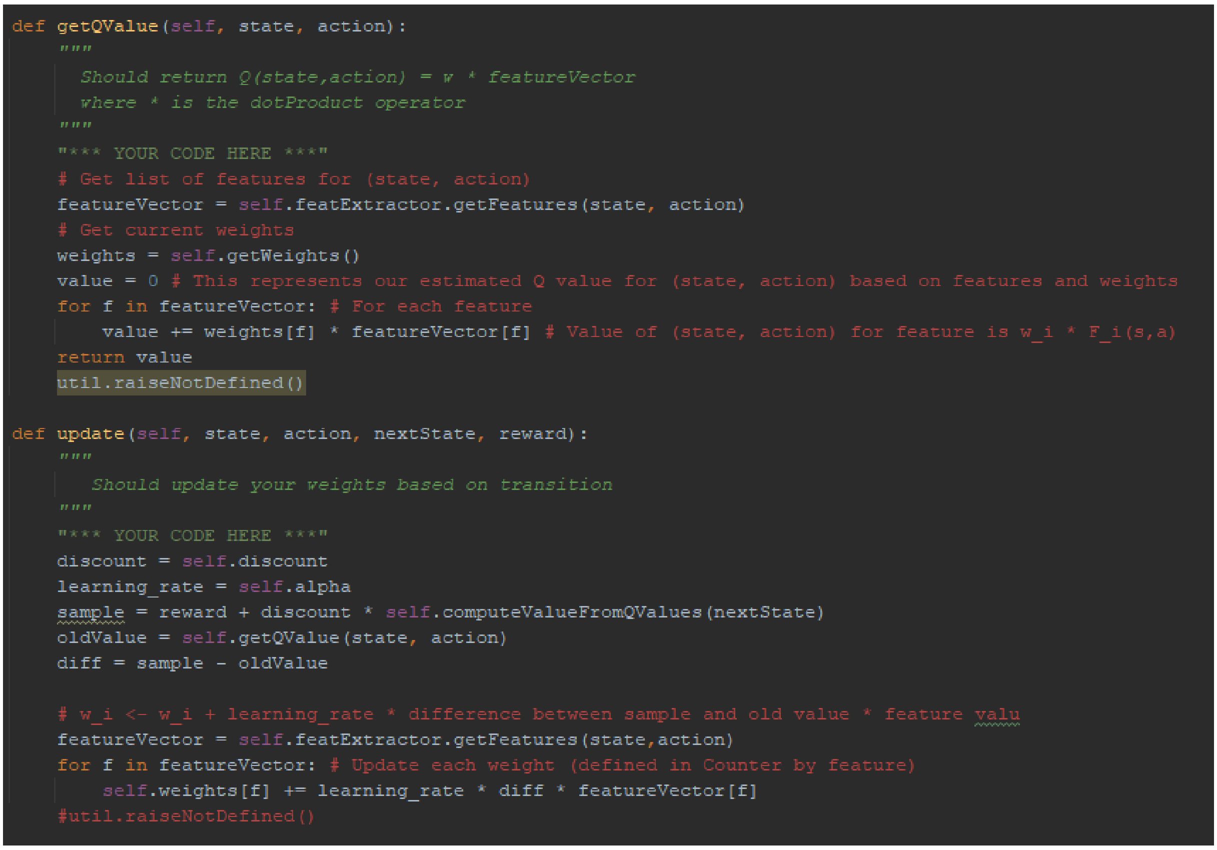
Question 10: Approximate Q-Learning

Autograder Results



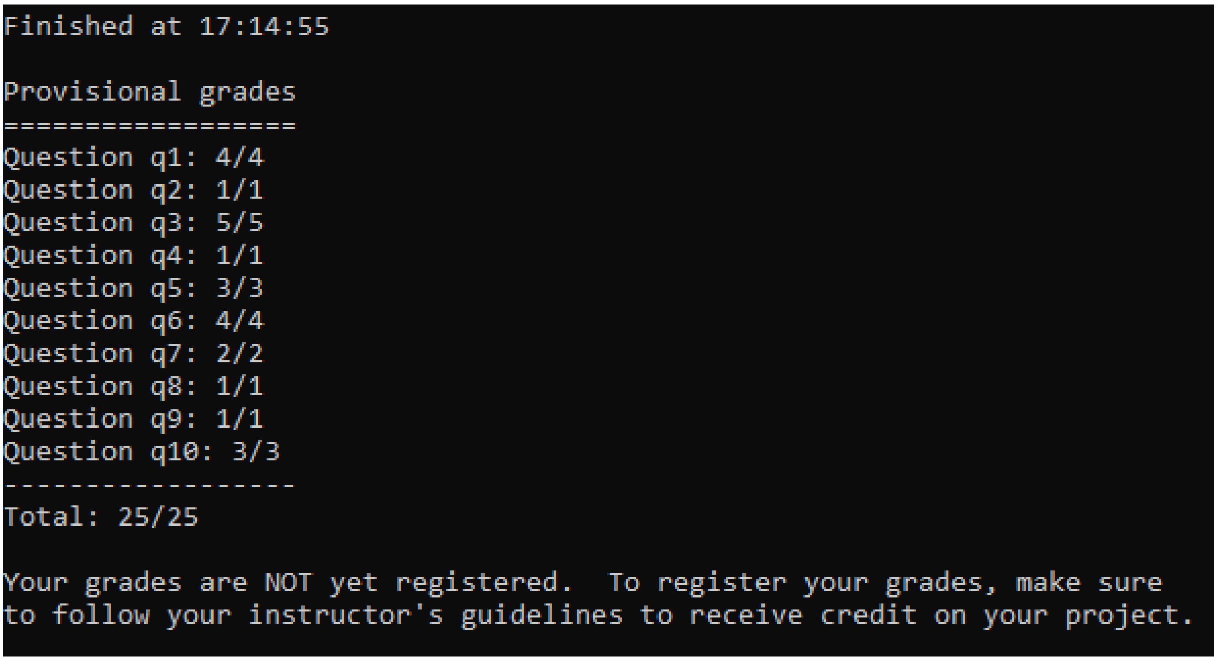
```
Question q10
============

*** PASS: test_cases\q10\1-tinygrid.test
*** PASS: test_cases\q10\2-tinygrid-noisy.test
*** PASS: test_cases\q10\3-bridge.test
*** PASS: test_cases\q10\4-discountgrid.test
*** PASS: test_cases\q10\5-coord-extractor.test

### Question q10: 3/3 ###


Finished at 17:14:24

Provisional grades
==================
Question q10: 3/3
------------------
Total: 3/3

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

Code Snippet



```
def getQValue(self, state, action):
    """
      Should return Q(state,action) = w * featureVector
      where * is the dotProduct operator
    """
    "*** YOUR CODE HERE ***"
    # Get list of features for (state, action)
    featureVector = self.featExtractor.getFeatures(state, action)
    # Get current weights
    weights = self.getWeights()
    value = 0 # This represents our estimated Q value for (state, action) based on features and weights
    for f in featureVector: # For each feature
        value += weights[f] * featureVector[f] # Value of (state, action) for feature is w_i * F_i(s,a)
    return value
    util.raiseNotDefined()

def update(self, state, action, nextState, reward):
    """
       Should update your weights based on transition
    """
    "*** YOUR CODE HERE ***"
    discount = self.discount
    learning_rate = self.alpha
    sample = reward + discount * self.computeValueFromQValues(nextState)
    oldValue = self.getQValue(state, action)
    diff = sample - oldValue

    # w_i <- w_i + learning_rate * difference between sample and old value * feature valu
    featureVector = self.featExtractor.getFeatures(state,action)
    for f in featureVector: # Update each weight (defined in Counter by feature)
        self.weights[f] += learning_rate * diff * featureVector[f]
    #util.raiseNotDefined()
```

All Questions Autograder



```
Finished at 17:14:55

Provisional grades
==================
Question q1: 4/4
Question q2: 1/1
Question q3: 5/5
Question q4: 1/1
Question q5: 3/3
Question q6: 4/4
Question q7: 2/2
Question q8: 1/1
Question q9: 1/1
Question q10: 3/3
------------------
Total: 25/25

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```