Dự án đã được thực hiện đúng theo hướng dẫn đã đưa ra và đã có chú thích và giải thích rõ ràng cho tất cả các phần trong code.

## Lặp giá trị (Value Iteration):

Để thực hiện phần này, cần hoàn thiện hàm computeActionFromValues và computeQValueFromValues trong file valueIterationAgents.py và trong lớp ValueIterationAgent để sử dụng hàm thứ nhất nhằm tìm ra hành động tốt nhất dựa trên hàm giá trị được lưu trong self.values, và hàm thứ hai với đầu vào là state và action để tính toán giá trị Q cho action tại state dựa trên hàm giá trị được lưu trong self.values. Ngoài ra, cần hoàn thiện hàm runValueIteration được gọi ở đầu lớp này để tính toán giá trị cho self.values.

Mục tiêu chính là thực hiện phương trình sau:

[Phương trình lặp giá trị]

Giải thích code:

#### Hàm computeQValueFromValues:

Hàm này tính đúng theo công thức, tính tổng tích xác suất xảy ra mỗi trạng thái phụ khi thực hiện hành động, phần thưởng thu được, giá trị hiện tại của trạng thái phụ và giá trị gamma để tính ra giá trị Q, tức phần được tính tổng trong công thức trên. Để lưu trữ giá trị của mỗi trạng thái phụ mà cần tính tổng, sử dụng một bộ đếm được triển khai sẵn trong chương trình có cấu trúc dictionary.

def computeQValueFromValues(self, state, action):

finalStates = self.mdp.getTransitionStatesAndProbs(state, action)

        weightedAverage = 0

        for finalState in finalStates:

            nextState = finalState[0]

            probability = finalState[1]

            weightedAverage += (probability \* (self.mdp.getReward(state, action, nextState)

+ (self.discount \* self.values[nextState])))

        return weightedAverage

#### Hàm computeActionFromValues:

Hàm này sử dụng giá trị Q đã tính toán để tìm giá trị max tại k+1 chính xác như công thức.

Để lưu trữ giá trị của mỗi hành động cần tìm giá trị lớn nhất (argMax), sử dụng một bộ đếm được triển khai sẵn trong chương trình có cấu trúc dictionary.

def computeActionFromValues(self, state):

if self.mdp.isTerminal(state):

            return None

        allActionsForState = self.mdp.getPossibleActions(state)

        finalAction = ""

        maxSum = float("-inf")

        for action in allActionsForState:

            weightedAverage = self.computeQValueFromValues(state, action)

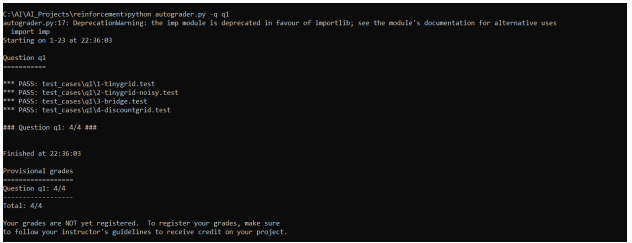
            if (maxSum == 0.0 and action == "") or weightedAverage >= maxSum:

                finalAction = action

                maxSum = weightedAverage

        return finalAction

* Kết quả autograder:



## Các chính sách (Policies)

Chúng ta biết rằng độ nhiễu (noise) càng thấp và hệ số chiết khấu (discount factor) càng cao thì càng tốt; bởi vì tăng độ nhiễu sẽ dẫn đến tăng các hành động không mong muốn và hệ số chiết khấu càng cao thì khi độ dài đường đi để đạt mục tiêu tăng lên, giá trị thu được sẽ bị giảm ít hơn nhưng ở đây do cần thay đổi đích nên chúng ta cũng xem xét điều đó với ghi nhớ rằng phần thưởng càng âm thì số lượt di chuyển để đạt các trạng thái kết thúc càng ít và trước tiên sẽ đi đến phần thưởng nhỏ hơn. Chúng ta đã sử dụng auto grader để đánh giá các phỏng đoán và đề xuất của mình cho các giá trị này.

* Giải thích code:
* Ở phần đầu, kết quả gần hơn được ưu tiên và mức chấp nhận rủi ro cao nên phần thưởng cần âm nhiều.
* Ở phần hai, kết quả gần hơn vẫn quan trọng nên phần thưởng vẫn âm nhưng điểm quan trọng là cần tránh vách đá nên cần tăng độ nhiễu và hệ số chiết khấu.
* Ở phần ba, cần tăng hệ số chiết khấu và phần thưởng âm hơn.
* Ở phần bốn, cần phần thưởng ít âm hơn và ở phần năm cần phần thưởng dương. Trong tất cả các phần, chúng tôi đã xem xét các điểm nêu trong hướng dẫn và sử dụng auto grader để đảm bảo tính đúng đắn của các số đưa ra.

def question2a():   
 answerDiscount = 0.02   
 answerNoise = 0   
 answerLivingReward = -4   
 return answerDiscount, answerNoise, answerLivingReward

def question2b():   
 answerDiscount = 0.5   
 answerNoise = 0.3   
 answerLivingReward = -0.5   
 return answerDiscount, answerNoise, answerLivingReward

def question2c():   
 answerDiscount = 0.9   
 answerNoise = 0.2   
 answerLivingReward = -1.5   
 return answerDiscount, answerNoise, answerLivingReward

def question2d():   
 answerDiscount = 0.8   
 answerNoise = 0.2   
 answerLivingReward = -0.5   
 return answerDiscount, answerNoise, answerLivingReward

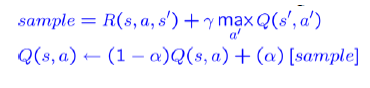
def question2e():   
 answerDiscount = 0.8   
 answerNoise = 0.01   
 answerLivingReward = 2   
 return answerDiscount, answerNoise, answerLivingReward

* Kết quả autograder:



## Học Q (Q-learning):

Ở phần này, cần thay đổi 4 hàm getQValue, computeValueFromQValues, update và computeActionFromQValues trong lớp QLearningAgent ở file qlearningAgents.py để triển khai công thức sau:



* Giải thích code:

#### Hàm update:

Hàm trước tiên kiểm tra xem trạng thái này đã được xét trước đó chưa và nếu chưa thì cần một bộ đếm kiểu dictionary cho nó. Sau đó làm chính xác theo công thức, sử dụng gamma và phần thưởng để tính mẫu trước. Sau đó thực thi dòng thứ hai của công thức và cập nhật giá trị Q cho trạng thái này với hành động đã thực hiện ở đầu vào hàm.

def update(self, state, action, nextState, reward):   
 difference = (reward + (self.discount \* self.getValue(nextState))) - self.getQValue(state, action)

      features = self.featExtractor.getFeatures(state, action)

        for feature in features:

            self.weights[feature] = self.weights[feature] + (self.alpha \* features[feature] \* difference)

#### Hàm computeActionFromQValues:

Trong hàm này, chúng ta tìm hành động tốt nhất và để chọn nó, sử dụng hàm random. Như code và các comment đã chỉ ra, với trạng thái đầu vào hàm, chúng ta tìm hành động tốt nhất bằng cách lặp qua các hành động có thể thực hiện trong trạng thái đó để kiểm tra xem có giá trị Q lớn nhất không, nếu có thì thêm vào danh sách để lưu các hành động tốt nhất và cuối cùng nếu danh sách này không rỗng thì chọn ngẫu nhiên một hành động.

def computeActionFromQValues(self, state):  
 actions = self.getLegalActions(state)

        if len(actions) == 0:

            return None

        maxValue = float("-inf")

        stateAction = ""

        for action in actions:

            if maxValue <= self.getQValue(state, action) or maxValue == float("-inf"):

                maxValue = self.getQValue(state, action)

                stateAction = action

        return stateAction

#### Hàm computeValueFromQValues:

Trong hàm này, chúng ta tìm hành động có giá trị Q lớn nhất cho trạng thái hiện tại đang xét và nếu trạng thái này không có hành động nào thì trả về 0.

def computeValueFromQValues(self, state):   
 actions = self.getLegalActions(state)

        if len(actions) == 0:

            return 0.0

        maxValue = float("-inf")

        for action in actions:

            if maxValue <= self.getQValue(state, action) or maxValue == float("-inf"):

                maxValue = self.getQValue(state, action)

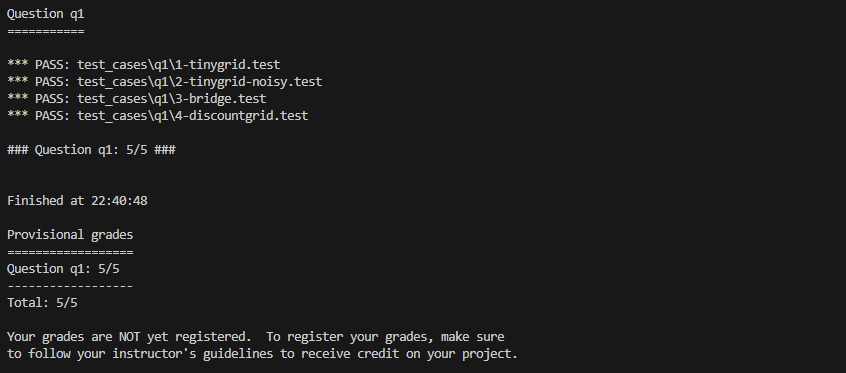
        return maxValue

#### Hàm getQValue:

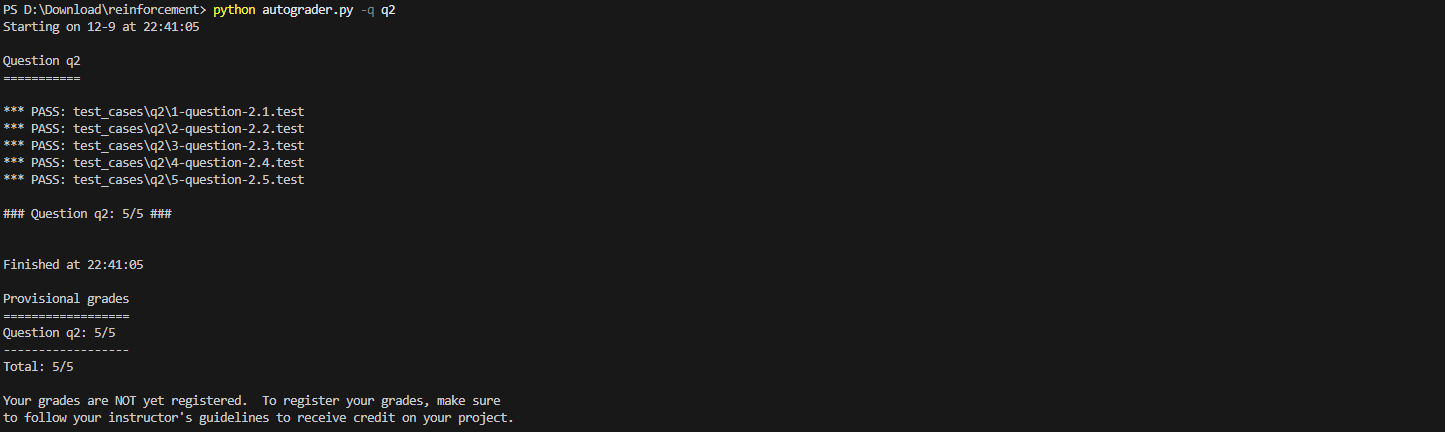
Đây là hàm getter rất đơn giản, với đầu vào là trạng thái và hành động, nếu trạng thái này có trong Qvalues thì trả về giá trị QValue, ngược lại kết quả trả về là 0.

def getQValue(self, state, action):   
 return self.values[(state, action)]

* Kết quả autograder:
* Q1



* Q2



* Q3

