CSEN 166L

Lab 6 Report (Reinforcement Learning)

Will Nyman, Calvin Uy, Adian Alvarado

**Task A: Implment *update, computeValuesFromQValues, getQValue, computeActionFromQValues***

**Source code:**

def getQValue(self, state, action):

#util.raiseNotDefined()

# (state,action) is the key; (s,a) is the key (s,a): 0.5

if (state,action) in self.qvalues:

test = 0 # just for testing

# return the associated q-value

return self.qvalues[(state,action)]

else:

test = 0 # just for testing

# add this (s,a) pair to dictionary and set its q value to 0

self.qvalues[(state,action)] = 0

# return its value which is 0

return self.qvalues[(state,action)]

# update the dictionary self.qvalues with this unseen (s,a) and initialize its value as 0.0

# self.qvalues[(s,a)]=0.0

# return this q-value

def computeValueFromQValues(self, state):

# returns V(state) by iterating over legal actions (a') and getting the greatest Q(state,a')

# create a list of all legal actions from state

legal\_actions = self.getLegalActions(state)

# if no legal actions (terminal state) return 0.0

if not (legal\_actions):

return 0.0

# initialize Q value with first legal action as 'max'

max\_dummy = self.getQValue(state, legal\_actions[0])

# for all legal actions after the first

# if its Q value is greater than the current max then make it the max

# (dont have to worry about choosing from multiple because it just wants the value)

for i in legal\_actions[1:]:

if self.getQValue(state,i) > max\_dummy:

max\_dummy = self.getQValue(state,i)

# return the max Q value (aka V(state))

return max\_dummy

#util.raiseNotDefined()

def computeActionFromQValues(self, state):

# returns best action by iterating over all legal actions and populating a list of actions that all

# have the same Q value from state and choosing a random one

#sets current max action and list of possible action candidates to empties

max\_act = None

candidate\_actions = []

# create a list of all legal actions from state

legal\_actions = self.getLegalActions(state)

# if no legal actions (terminal state) return None

if not (legal\_actions):

return None

# set first possible action to max action and add action to candiate actions

candidate\_actions.append(legal\_actions[0])

max\_dummy = self.getQValue(state,legal\_actions[0])

#for every actions starting after the first

for i in legal\_actions[1:]:

# if value for i has same as max then add it to candidate actions

if self.getQValue(state,i) == max\_dummy:

candidate\_actions.append(i)

# if value for i is greater than value for candidate actions

elif self.getQValue(state,i) > max\_dummy:

# set new max value to current actions value and clear the list of candidates

# and add new max action to candidate actions

max\_dummy = self.getQValue(state,i)

candidate\_actions.clear()

candidate\_actions.append(i)

# choose a random action from candidate actions and return

max\_act = random.choice(candidate\_actions)

return max\_act

# YL: if there are multiple best actions, then choose a random one

util.raiseNotDefined()

def update(self, state, action, nextState, reward):

# get max Q(s',a')

# it references the gamma\*max(Q(s',a')) from slides/pdf

# create a list of legal actions from next state (to use in sample variable)

next\_legal\_actions = self.getLegalActions(nextState)

# initial max\_next\_q:

# 0 if no legal actions (terminal)

# Q value of (nextState,next\_legal\_actions[0]) if there are legal actions

if not (next\_legal\_actions):

max\_next\_q = 0

else:

max\_next\_q = self.getQValue(nextState,next\_legal\_actions[0])

# find max Q value for all possible actions from nextState

for next\_action in next\_legal\_actions[1:]:

if self.getQValue(nextState, next\_action) > max\_next\_q:

max\_next\_q = self.getQValue(nextState, next\_action)

# create sample variable

sample = reward + self.discount \* (max\_next\_q)

# update Q value

self.qvalues[(state,action)] = ((1-self.alpha) \* self.getQValue(state,action)) + (self.alpha\*(sample))

#OR

#self.qvalues[(state,action)] = self.getQValue(state,action) + self.alpha\*(sample - self.getQValue(state,action))

#util.raiseNotDefined()

**Experiment 1: Gridworld after four episodesA screenshot of a computer

Description automatically generated**

**Experiment 2: Autograder resultsA screenshot of a computer

Description automatically generated**

**Task B: modify getAction**

**Source code:**

def getAction(self, state):

# Pick Action

legalActions = self.getLegalActions(state)

action = None

# if random choice prob. hits

if (util.flipCoin(self.epsilon)):

# choose random legal action and return

random\_action = random.choice(legalActions)

return random\_action

else:

# choose best legal action and return

best\_action = self.computeActionFromQValues(state)

return best\_action

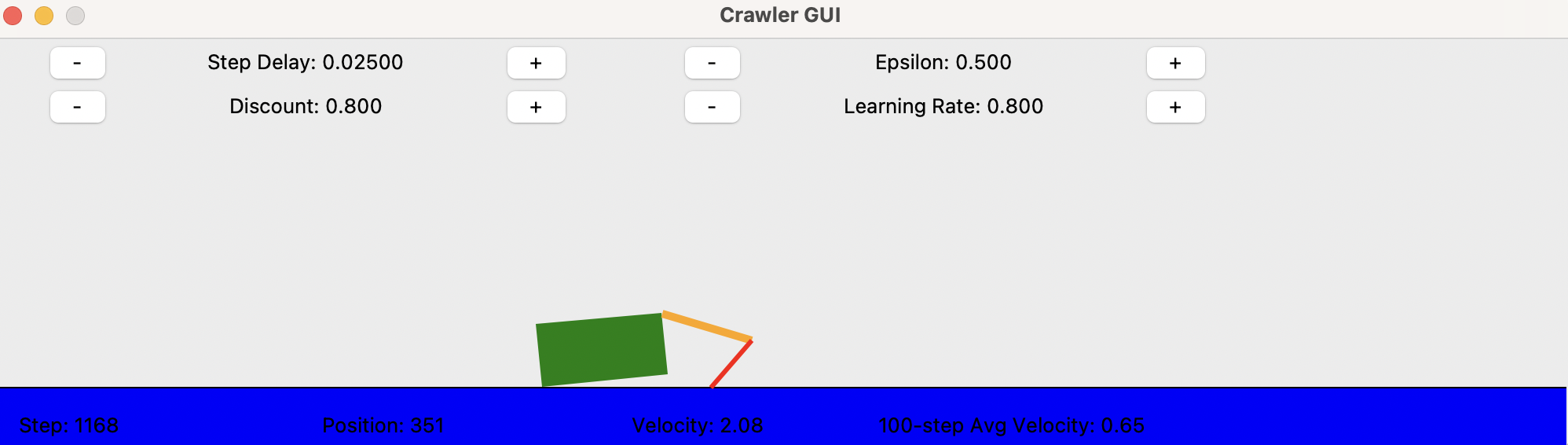
util.raiseNotDefined()

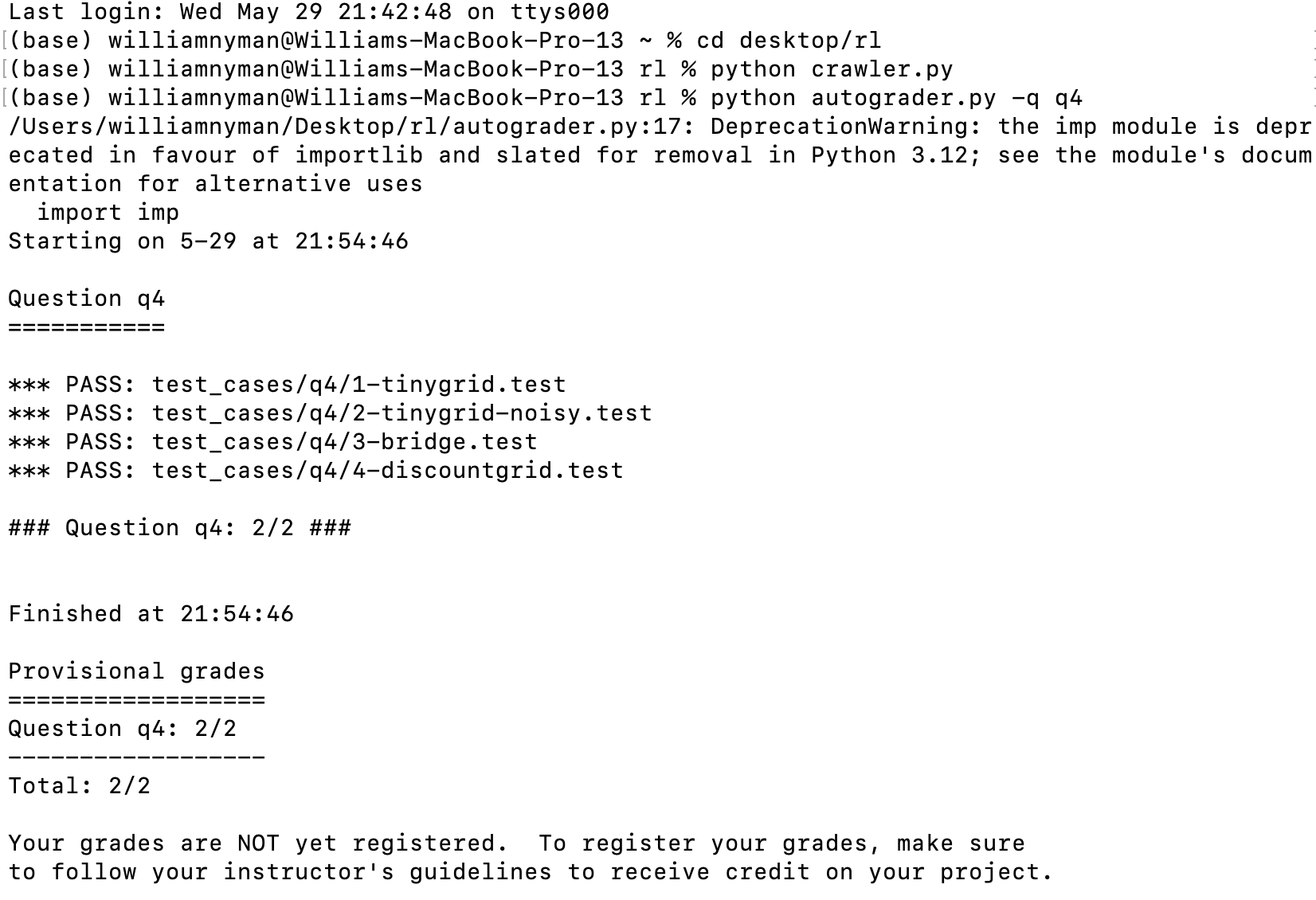
return action

**Experiment 1: Gridworld values after 100 episodesA screenshot of a video game

Description automatically generated**

**Experiment 2: Crawler ~1100 steps in**

****

**Task B – Experiment 3: Autograder results**

**Code explanations:**

**update**

The update function follows the Q learning equation. It references the slide from class/lab assignment. It takes s, a, s’, and r as parameters and updates Q(s,a) within self.qvalues. First it determines the max Q value (*max\_next\_q)* over all possible actions from the next state (*next\_legal\_actions*) to use as part of the *sample* variable. It does so by iterating *next\_action* over *next\_legal\_actions*, if no legal actions are available then it is at a terminal state and *max\_next\_q*  is 0. If there are legal actions, then *max\_next\_q = next\_legal\_actions[0]* initially and *next\_action*  iterates through the rest of the legal actions and finds the max Q-value. Next, *sample* is established as in the sample equation utilizing *reward* and gamma (*discount)* and the value for Q(s,a) is updated within self.qvalues based on the Q-learning formula also utilizing the learning rate (*alpha)* .

**getQValue**