Eric Smrkovsky
CSci 166
10/18/2020
Reinforcement Learning Project

I will provide a picture of each function that I wrote. Each function has a description within the code comments.

Photos 1-5 are from the valueiterationagents.py file.

Photos 6-8 are from analysis.py

The rest are from qlearningagents.py

The very last picture is the autograder results.

Enjoy!

mdp.py | featureExtractors.py | valueIterationAgents.py

```python
def __init__(self, mdp, discount = 0.9, iterations = 100):
    """
      Your value iteration agent should take an mdp on
      construction, run the indicated number of iterations
      and then act according to the resulting policy.

      Some useful mdp methods you will use:
        Note: these are described in mdp.py, defined in gridworld.py
        Note: these pull needed values from the mdp object
          mdp.getStates()
                    Return a list of all states in the MDP.
          mdp.getPossibleActions(state)
                    Returns list of actions from state
          mdp.getTransitionStatesAndProbs(state, action) t(s,a,s')
                    Returns list of (nextState, prob) pairs
                        Note: if terminal state returns []
          mdp.getReward(state, action, nextState)
                    Returns reward from t(s,a,s')
          mdp.isTerminal(state)
                    Returns true if the current state is a terminal state.
    """
    #compared to algorithim in textbook Fig 17.4
    self.mdp = mdp              # input mdp
    self.discount = discount    # input gamma
    self.iterations = iterations # number of iterations defined by user
    self.values = util.Counter() # --U-- A Counter is a dict with default 0
    self.runValueIteration()    # function VALUE-ITERATION call to run
```

Eric Smrkovsky
CSci 166
10/18/2020
Reinforcement Learning Project

mdp.py featureExtractors.py valueIterationAgents.py

```
def runValueIteration(self):
    # Write value iteration code here
    "*** YOUR CODE HERE ***"
    """ This value iteration function checks that the state is not a
    terminal state and then runs a loop for each iteration of the algorithim
    calculating the maximum value of the possible actions and filling a vector
    with the q values. It calls getAction and getQValue which I define later.
    These two functions do all of the calculations necessary to determin them
    best action to take for a optimum policy. """

    # runs this loop (-i ...) specified in input
    for i in range(self.iterations): # for each iteration

        # get a new dict for this iteration U <- U'
        newStateValues = self.values.copy()

        s = self.mdp.getStates()
        # for each state in s do
        for state in s:

            terminal = self.mdp.isTerminal(state)
            # if the current state is NOT terminal then compute action/Qval
            if not terminal:
                action = self.getAction(state) # get max action
                newStateValues[state] = self.getQValue(state, action)

        # re populate the original dict with the new values
        self.values = newStateValues

        # completed value iteration
```

Eric Smrkovsky
CSci 166
10/18/2020
Reinforcement Learning Project

```python
    #these two functions just call the functions I defined.
def getAction(self, state):
    # "Returns the policy at the state (no exploration)."
    return self.computeActionFromValues(state)

def getQValue(self, state, action):
    return self.computeQValueFromValues(state, action)
```

```python
def computeQValueFromValues(self, state, action):
    """
      Compute the Q-value of action in state from the
      value function stored in self.values.
    """
    "*** YOUR CODE HERE ***"
    """
        This function uses the Bellman equation to calculate the utility
        of a q-state. First we get the state (called nextState) that is
        reachable by taking an action from the given state along with
        it's probability of occuring. Next we find the reward of going to
        that next state. Next, we calculate the utility using the Bellman
        equation to find the utility (currentStateValue) of k+1 for the
        input state.
    """

    currentStateValue = 0  # value starts at zero

    for nextState, p in self.mdp.getTransitionStatesAndProbs(state, action):
        r = self.mdp.getReward(state, action, nextState) # get reward
        y = self.discount   # get discount (gamma)
        nextU = self.getValue(nextState) # get the max utility of next state
        currentStateValue += p * (r + y * nextU) # calculate U of k+1 given s
    return currentStateValue
```

Eric Smrkovsky
CSci 166
10/18/2020
Reinforcement Learning Project

```python
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 ***"
    """
        This function checks if the state is terminal and if it is
        then returns None as in the instructions. Next, it calculates the
        values of each action and stores them in a temporary dict for
        comparison. Next, the function argMax() defined in util.py is used
        to find the maximum value to be returned which is eventually
        multiplied with gamma in the Bellman equation.
    """
    # if there aren't any legal actions, return None (break ties)
    if self.mdp.isTerminal(state):
      return None

    # create a temp dict for comparing actions
    tempCache = util.Counter()

    # get the optimum policy value of each state/action/Qval
    for action in self.mdp.getPossibleActions(state):
        tempCache[action] = self.getQValue(state, action)
    # use argmax to find best policy
    # (returns key with highest value defined in util.py line 334)
    policy = tempCache.argMax()
    return policy # return the best action from the given state
```

Eric Smrkovsky
CSci 166
10/18/2020
Reinforcement Learning Project

mdp.py | featureExtractors.py | analysis.py | valueIterationAgents

```python
######################
# ANALYSIS QUESTIONS #
######################

# Set the given parameters to obtain the specified policies through
# value iteration.

"""     Question Two

It seems that any value less than .016 for the noise paramerter causes the
probability of taking the action east starting from state (1,1) and endings
in state (6,1) to become positive. The original value (0.02) for noise causes
a negative probability in state (1,1) of going east. As I tested noise values,
I decreased the noise parameter by 0.005 until I started to see changes.
Slowly I watched the far right states turn green, but they all had to become
green for a reward of ten.
"""
def question2():
    answerDiscount = 0.9
    answerNoise = 0.016 # 0.2
    return answerDiscount, answerNoise
```

Eric Smrkovsky
CSci 166
10/18/2020
Reinforcement Learning Project

```python
""" Question 3
I found these mainly by trial and error. You can watch the affects of each try
on the different q values for taking an action from a state and how it changes.
"""
def question3a():
    # You can use a very small value of gamma to cause the bellman equation to
    # add the value of the next states reward. Since we get to the rewards
    # faster by going right, this method works for our wanted policy type.
    answerDiscount = .1 # or even .000000001
    answerNoise = 0
    answerLivingReward = 0
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'

def question3b():
    answerDiscount = .2
    answerNoise = .2
    answerLivingReward = .5
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'

def question3c():
    answerDiscount = .9
    answerNoise = .02
    answerLivingReward = 0
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'

def question3d():
    answerDiscount = .9
    answerNoise = .5
    answerLivingReward = 0
    return answerDiscount, answerNoise, answerLivingReward
```

Eric Smrkovsky
CSci 166
10/18/2020
Reinforcement Learning Project

```python
def question3d():
    answerDiscount = .9
    answerNoise = .5
    answerLivingReward = 0
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'

def question3e():
    answerDiscount = .9
    answerNoise = .02
    answerLivingReward = 1
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'

def question8(): #after testing values, I determined that it is not possible.
    answerEpsilon = None
    answerLearningRate = None
    return 'NOT POSSIBLE'
```

Eric Smrkovsky
CSci 166
10/18/2020
Reinforcement Learning Project

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

        "*** YOUR CODE HERE ***"
        # Initialize the q values by assigning a new dict to it
        # A Counter is a dict with default 0
        self.Q_value = 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 ***"
        # This function returns 0 if there is no possible actions, this means we
        # haven't seen a state yet. If there is possible actions we return the
        # Q node value.

        if not self.getLegalActions(state):
            return 0.0
        else:
            return self.Q_value[(state, action)]
```

Eric Smrkovsky
CSci 166
10/18/2020
Reinforcement Learning Project

```python
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 ***"

    legalActions = self.getLegalActions(state)

    #if no legal actions return 0.0
    if not legalActions:
        return 0.0

    #otherwise we assign value to a really small negative number as a default
    else:
        maxValue = float('-inf')

    #next we update value to the highest value from all states within reach
    for action in legalActions: # for each action
        Qval = self.getQValue(state, action) #get Q value of that action
        maxValue = max(maxValue, Qval) #take max of previous value and Qval
    return maxValue #finally return the max value
```

Eric Smrkovsky
CSci 166
10/18/2020
Reinforcement Learning Project

```
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 ***"
    legalActions = self.getLegalActions(state)

    #if no legal actions return None
    if not legalActions:
        return None

    bestAction = [] # no best action yet make blank

    #get the action with the maxValue from the rest of the Qnodes
    QValue = self.computeValueFromQValues(state)

    # for each legal action
    for action in legalActions:
        Qval = self.getQValue(state, action) #get Qvalue of this state/action
        if QValue == Qval: #if they match add to list of best actions
            bestAction.append(action)
    return random.choice(bestAction) # choose a best action at random
```

Eric Smrkovsky
CSci 166
10/18/2020
Reinforcement Learning Project

```python
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)
    """
    epsilon = self.epsilon #define epsilon for this function
    legalActions = self.getLegalActions(state) #list of possible actions

    #if no legal actions return None
    if not legalActions:
        return None
    else:
        # with a small probability we act randomly
        if util.flipCoin(epsilon): # with a probability of epsilon
            return random.choice(legalActions) # we return a random action

        # or we return the best action with probability of 1 - epsilon
        else:
            return self.computeActionFromQValues(state)
```

Eric Smrkovsky
CSci 166
10/18/2020
Reinforcement Learning Project

```python
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 ***"
    """ Update's the Q-Value (Qk+1) I designed this to match slide 57
    """
    gamma = self.discount   #discount
    alpha = self.alpha      #learning rate
    max_qk_sPrime_a = self.computeValueFromQValues(nextState)
    sample = reward + (gamma * max_qk_sPrime_a) #new sample
    qk_sa = self.Q_value[(state, action)]   #old sample (new sample later)

    # Incorporate the new estimate into a running average
    self.Q_value[(state, action)] = qk_sa + alpha * (sample - qk_sa)

    # or this is eqivalent
    # self.Q_value[(state, action)] = ((1 - alpha) * qk_sa) + (alpha * sample)
```

```python
def getQValue(self, state, action):
    """
      Should return Q(state,action) = w * featureVector
      where * is the dotProduct operator
    """
    "*** YOUR CODE HERE ***"
    # Q(s,a) = w1f1(s,a) + w2f2(s,a)+ ... + wnfn(s,a)
    # this function calculater the wnfn(s,a) given the state/action pair
    # get weights and the vector of features and return product

    w = self.getWeights()  # get the weight
    f = self.featExtractor.getFeatures(state, action) # get the feature
    q_s_a = w * f # compute the product
    return q_s_a # return the product
```

Eric Smrkovsky
CSci 166
10/18/2020
Reinforcement Learning Project

```
def update(self, state, action, nextState, reward):
    """
       Should update your weights based on transition
    """
    "*** YOUR CODE HERE ***"
    # This function updates the weights of active features for the qlearner!
    # I wrote this to match the linear Q-functions on slide 68
    # The comments describe what part of the algorithim the code represents
    gamma = self.discount   #discount
    alpha = self.alpha      #learning rate
    # max_qk_sPrime_aPrime == maxQ(s',a')
    max_qk_sPrime_aPrime = self.computeValueFromQValues(nextState)
    stuff = reward + (gamma * max_qk_sPrime_aPrime) # == [r + y(maxQ(s',a'))]
    q_sa = self.getQValue(state, action)   # == Q(s,a)
    difference = stuff - q_sa # == [r + y(maxQ(s',a'))] - Q(s,a)
    features = self.featExtractor.getFeatures(state, action) #get the features

    # for each feature update the weights
    for f in features:
        self.weights[f] += alpha * difference * features[f]
    return
```

Eric Smrkovsky
CSci 166
10/18/2020
Reinforcement Learning Project

```
Finished at 22:13:25

Provisional grades
==================
Question q1: 4/4
Question q2: 1/1
Question q3: 5/5
Question q4: 0/1
Question q5: 0/3
Question q6: 4/4
Question q7: 2/2
Question q8: 1/1
Question q9: 1/1
Question q10: 3/3
------------------
Total: 21/25
```

Note: The only reason why q4 and q5 are at a zero value is because they were not presented to me within the lab questions, so I had no way to answer them.