Artificial Intelligence: Lab04 report

1. Value Iteration

Principle

Here, the class ValueIterationAgent implements the value iteration algorithm for a given Markov decision process (MDP) to compute the optimal values of each state, and then uses these values to compute the optimal policy. The value iteration algorithm involves iteratively updating the estimated value of each state based on the values of its successor states and the rewards obtained by taking different actions from the current state.
The iteration value is based on Bellman's equation :

$$V^*(s) = max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^*(s')]$$

Code screenshots

At the beginning we tried this code. Indeed, since we want to compute the Q-values we can directly use the associated method. However, we were not able to obtain exactly the same results as those wanted.

```python
def runValueIteration(self):
    """
      Run the value iteration algorithm. Note that in standard
      value iteration, V_k+1(...) depends on V_k(...)'s.
    """
    "*** YOUR CODE HERE ***"
    # The higher the number of iterations is, the more accurate the result will be
    for iter in range(self.iterations):
        for state in self.mdp.getStates():
            tempMax = -float('inf')
            for action in self.mdp.getPossibleActions(state):
                # Retrieves the best Q-value for all the possible actions
                tempMax = max(tempMax, self.computeQValueFromValues(state, action))
            # The best Q-value is the initial one so the result is in reality zero
            if tempMax == -float('inf'):
                self.values[state] = 0.0
            # Updates V_iter(state) with the new best Q-value
            else:
                self.values[state] = tempMax
```

Then, we changed the method used and made a copy of the $V_k(s)$ and directly computed the Q-values in the runValueIteration method. With this, we were able to obtain correct results.

```python
def runValueIteration(self):
    """
      Run the value iteration algorithm. Note that in standard
      value iteration, V_k+1(...) depends on V_k(...)'s.
    """
    "*** YOUR CODE HERE ***"
    # The higher the number of iterations is, the more accurate the result will be
    for iter in range(self.iterations):
        values = self.values.copy
        for state in self.mdp.getStates():
            tempMax = -float('inf')
            for action in self.mdp.getPossibleActions(state):
                # Retrieves the best Q-value for all the possible actions
                tempMax = max(tempMax, sum([prob*(self.mdp.getReward(state, action, nextState) + self.discount * self.values[nextState])
                          for nextState, prob in self.mdp.getTransitionStatesAndProbs(state, action)]))
            # The best Q-value is the initial one so the result is in reality zero
            if tempMax == -float('inf'):
                self.values[state] = 0.0
            # Updates V_iter(state) with the new best Q-value
            else:
                self.values[state] = tempMax
```

This method runs the value iteration algorithm for the specified number of iterations. It iterates over each state and for each state, iterates over all possible actions and calculates the expected value of taking that action, based on the values of the successor states and the rewards obtained by taking that action. It then updates the estimated value of the state with the maximum expected value over all actions. This is repeated for each state for a specified number of iterations until convergence.

```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 ***"
    # Returns the value Q*(s, a) = Sum(T(s,a,s') * [R(s,a,s') + gamma*V(s')])
    return sum([prob*(self.mdp.getReward(state, action, nextState) + self.discount * self.values[nextState])
                for nextState, prob in self.mdp.getTransitionStatesAndProbs(state, action)])
```

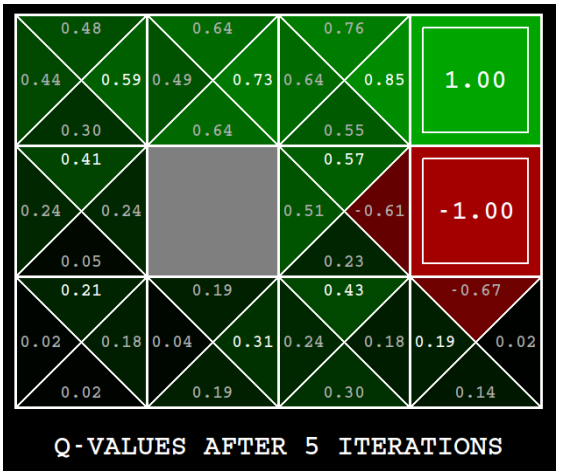
This method computes the Q-value of a given state-action pair, which is the expected value of taking that action and then following the optimal policy afterwards. It does this by iterating over the possible successor states and computing the weighted sum of the rewards obtained and the estimated values of the successor states.

```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 ***"
    # Checks if the current state is a terminal state
    if self.mdp.isTerminal(state):
        return None
    Qmax, bestAction = -float('inf'), None
    for action in self.mdp.getPossibleActions(state):
        # Computes the Q-value for all possible actions
        Q = self.computeQValueFromValues(state, action)
        # If the new Q-value is better than the stored one, we update it and keep the associated action
        if Qmax < Q:
            Qmax = Q
            bestAction = action
    return bestAction
```

This method computes the best action to take in a given state according to the currently estimated values of the states. It does this by iterating over all possible actions and computing the Q-value of taking each action, and then selecting the action with the highest Q-value.

Tests

Configuration 1: python3 gridworld.py -a value -i 5




```
RUNNING 1 EPISODES

BEGINNING EPISODE: 1

Started in state: (0, 0)
Took action: north
Ended in state: (0, 1)
Got reward: 0.0

Started in state: (0, 1)
Took action: north
Ended in state: (0, 2)
Got reward: 0.0

Started in state: (0, 2)
Took action: east
Ended in state: (0, 2)
Got reward: 0.0

Started in state: (0, 2)
Took action: east
Ended in state: (1, 2)
Got reward: 0.0

Started in state: (1, 2)
Took action: east
Ended in state: (2, 2)
Got reward: 0.0

Started in state: (2, 2)
Took action: east
Ended in state: (3, 2)
Got reward: 0.0

Started in state: (3, 2)
Took action: exit
Ended in state: TERMINAL_STATE
Got reward: 1

EPISODE 1 COMPLETE: RETURN WAS 0.5314410000000002


AVERAGE RETURNS FROM START STATE: 0.5314410000000002
```

As expected, we obtain the same results as those present in the statement.

Configuration 2: python3 gridworld.py -a value -i 100 -k 10

```
EPISODE 1 COMPLETE: RETURN WAS 0.47829690000000014
EPISODE 2 COMPLETE: RETURN WAS 0.43046721000000016
EPISODE 3 COMPLETE: RETURN WAS 0.5314410000000002
EPISODE 4 COMPLETE: RETURN WAS 0.43046721000000016
EPISODE 5 COMPLETE: RETURN WAS 0.38742048900000015
EPISODE 6 COMPLETE: RETURN WAS 0.47829690000000014
EPISODE 7 COMPLETE: RETURN WAS 0.47829690000000014
EPISODE 8 COMPLETE: RETURN WAS 0.5904900000000002
EPISODE 9 COMPLETE: RETURN WAS 0.5904900000000002
EPISODE 10 COMPLETE: RETURN WAS 0.47829690000000014

AVERAGE RETURNS FROM START STATE: 0.4873963509000001
```

The conclusion that can be drawn from this part is that the value of the start state (0,0) and the empirical resulting average reward (printed after the 10 rounds of execution finish) are quite close. This suggests that the value iteration algorithm is effective in computing a policy that leads to a good overall reward in this grid world scenario.

2. Q-Learning

Q-Learning Agent

Principle

This class is a reinforcement learning agent using the Q-Learning algorithm. The goal of this algorithm is to learn an optimal policy for choosing actions in a given environment based on the current state. The agent uses Q values to estimate the quality of actions in each state. The functions getQValue, computeValueFromQValues and computeActionFromQValues are used to calculate these Q values.
The update function updates the Q values using Bellman's equation :

$$V^*(s) = max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^*(s')]$$

The value iteration agent does not actually learn from experience. On the contrary, he thinks about his CDM model to arrive at a complete policy before even interacting with a real environment. When interacting with the environment, it simply follows the precomputed policy.

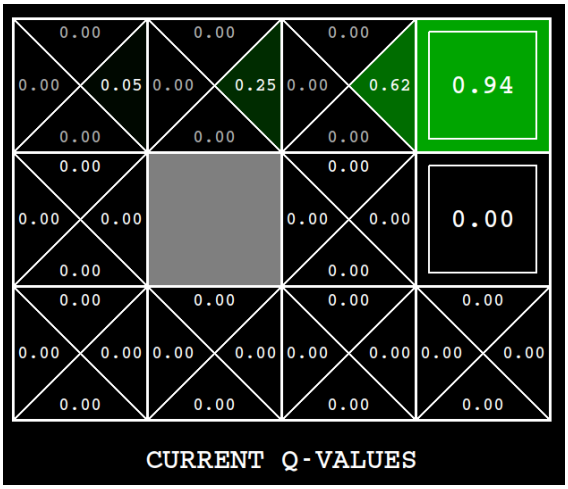
Code screenshots

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

    "*** YOUR CODE HERE ***"
    self.values = util.Counter() # Same as in ValueIteration

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
    """
    if (state, action) in self.values.keys():
        return self.values[(state, action)]
    else:
        return 0.0


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 ***"
    Qmax = -float('inf')
    # Retrieves the maximum Q-value among the explored values for all the possible
    # from the current state
    for action in self.getLegalActions(state):
        Q = self.getQValue(state, action)
        if Q > Qmax:
            Qmax = Q
    if Qmax == -float('inf'):
        return 0.0
    return Qmax
```

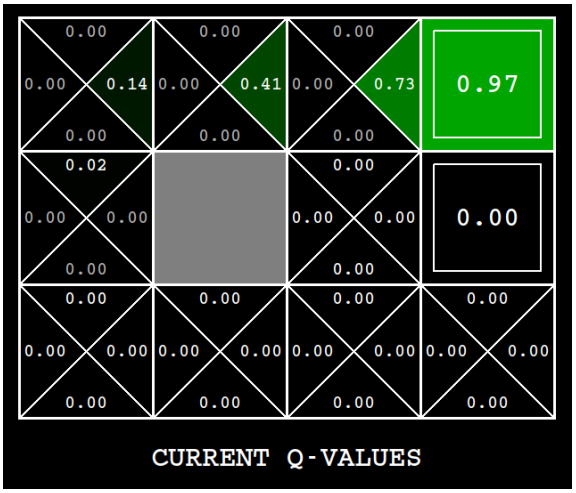
```python
def computeActionFromQValues(self, state):
    """
      Compute the best action to take in a state.  Note that if there
      are no legal actions, which is the case at the terminal state,
      you should return None.
    """
    "*** YOUR CODE HERE ***"
    actions = self.getAction(state)
    Qmax = -float('inf')
    # Retrieves the maximum Q-value among the explored values for all the possible
    # from the current state
    for action in actions:
        Q = self.getQValue(state, action)
        if Q > Qmax:
            Qmax = Q
    if Qmax == -float('inf'):
        return None
    # Randomly pick one action among all the best actions possible (actions which have the max Q-value)
    bestActions = [action for action in actions if self.getQValue(state, action) == Qmax]
    return random.choice(bestActions)
```

```python
def update(self, state, action, nextState, reward: float):
    """
      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 ***"
    # Q(s,a) = (1-alpha)*Q(s,a) + alpha * [R(s,a,s') + gamma * max_a'(Q(s',a'))]
    # At the beginning, Q(s,a) = alpha * [R(s,a,s') + gamma * max_a'(Q(s',a'))] since Q(s,a) is
    # not yet explored
    sample = reward + self.discount*self.computeValueFromQValues(nextState)
    self.values[(state, action)] = (1-self.alpha)*self.getQValue(state, action) + self.alpha * sample
```

Tests



k = 4



k = 5

```
EPISODE 1 COMPLETE: RETURN WAS 0.5904900000000002
EPISODE 2 COMPLETE: RETURN WAS 0.5904900000000002
EPISODE 3 COMPLETE: RETURN WAS 0.5904900000000002
EPISODE 4 COMPLETE: RETURN WAS 0.5904900000000002
EPISODE 5 COMPLETE: RETURN WAS 0.5904900000000002

AVERAGE RETURNS FROM START STATE: 0.5904900000000002
```

We observe that when we deactivate the noises, after 4 iterations, we obtain the same values as the statement.

Epsilon-greedy action

Principle

The goal of this part is that the exploration of state space has to be done first and after getting a good policy, exploitation of that policy has to be done. With the probability epsilon, we choose a random action in a state, otherwise the current optimal policy.

Code screenshots

```
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)
    "*** YOUR CODE HERE ***"
    if util.flipCoin(self.epsilon):
        return random.choice(legalActions)
    return self.computeActionFromQValues(state)
```

The method starts by retrieving the possible legal actions from the given state by calling the "getLegalActions(state)" method.

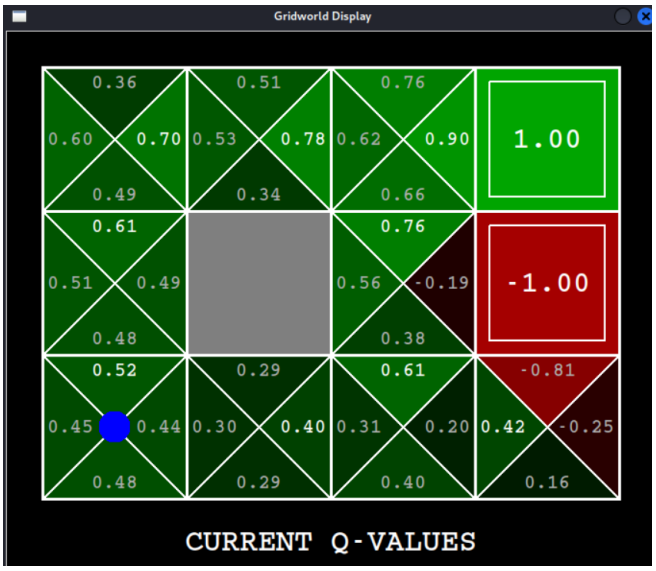
Then the method uses the epsilon-greedy policy to choose the action to perform. The epsilon is a probability of choosing a random action rather than using the estimated Q values for each action. If the result of the util.flipCoin(self.epsilon) method is true, the agent chooses a random action among the possible legal actions. Otherwise, the agent uses the "computeActionFromQValues(state)" method to choose the action that maximizes the Q value for the given state.

Finally, the method returns the chosen action.

It is important to note that the "computeActionFromQValues(state)" method must be implemented separately. It takes the game state as input and returns the action that maximizes the Q value for that state, using the estimated Q values for each possible action from the given state.

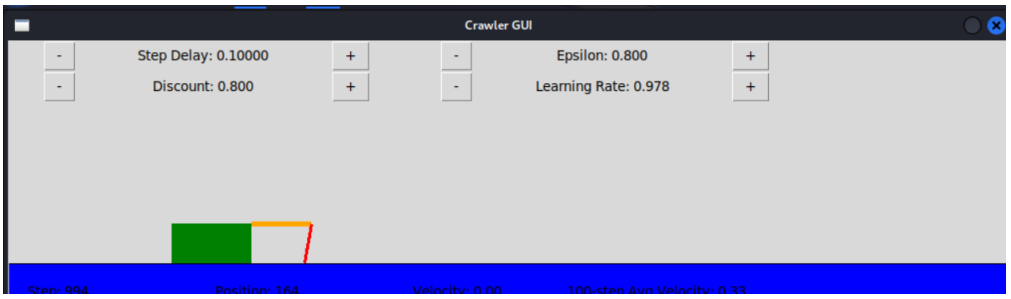
Tests

First with this command : `python3 gridworld.py -a q -k 100`



At first, the program explores a bit of all the possible paths and after a while, it starts to find the right one and ends up taking this one every time.

Then with this one : `python3 crawler.py`



The observation is that after a decent number of iterations trying to walk and having learned, epsilon should be lowered from its original value as it should now exploit the learned one rather than keep trying.

Final test with : `python3 pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid`

```
Average Score: 498.0
Scores:        499.0, 499.0, 495.0, 499.0, 495.0, 503.0, 495.0, 499.0, 501.0, 495.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

In this part, the first reinforcement learning agent is trained for 2000 episodes and then tested on 10 games, my implementation won all 10 games. This method does not scale to larger environments due to the amount of QValues to be stored and the amount of training that will be required to visit most of the state space a decent amount of times.

3. Approximate Q-Learning

Principle

In this part, we implemented an approximate Q-learning agent.
self.weights contain the weights indexed by feature. First the agent is trained and then tested. This solves the major problem of Q-learning in larger state spaces where it's simply not feasible. The idea here is that QValue is represented as a weighted linear sum of feature values for a state, now we need a way to get the feature value of the state and a way of updating weights based on what the agent sees.

Code screenshots

```
167 class ApproximateQAgent(PacmanQAgent):
168     """
169        ApproximateQLearningAgent
170        You should only have to overwrite getQValue
171        and update.  All other QLearningAgent functions
172        should work as is.
173     """
174     def __init__(self, extractor='IdentityExtractor', **args):
175         self.featExtractor = util.lookup(extractor, globals())()
176         PacmanQAgent.__init__(self, **args)
177         self.weights = util.Counter()
178 
179     def getWeights(self):
180         return self.weights
181 
182     def getQValue(self, state, action):
183         """
184           Should return Q(state,action) = w * featureVector
185           where * is the dotProduct operator
186         """
187         w = self.getWeights()
188         featureVector = self.featExtractor.getFeatures(state, action)
189         return w * featureVector
```

```
190 
191     def update(self, state, action, nextState, reward: float):
192         """
193            Should update your weights based on transition
194         """
195         featureVector = self.featExtractor.getFeatures(state, action)
196 
197         maxQFromNextState = self.computeValueFromQValues(nextState)
198         actionQValue = self.getQValue(state, action)
199 
200         #Updates the weight vector based on the observed reward and the difference between the estimated Q-value
201         # of the current state-action pair and the estimated Q-value of the next state.
202         for feature in featureVector:
203             self.weights[feature] += self.alpha * (reward + self.discount * maxQFromNextState - actionQValue) * \
204                                      featureVector[feature]
205 
206     def final(self, state):
207         """Called at the end of each game."""
208         # call the super-class final method
209         PacmanQAgent.final(self, state)
210 
211         # did we finish training?
212         if self.episodesSoFar == self.numTraining:
213             # you might want to print your weights here for debugging
214             print ("Final weights vector: ")
215             print (self.weights)
216             pass
```

This ApproximateQAgent class is an implementation of a Pacman agent that uses reinforcement learning to learn how to maximize its reward in a given environment. This agent uses a method called Q-learning (or Q-learning) which is a reinforcement learning method based on the Q-value function, which estimates the value of each state-action based on the expected rewards from this state. -stock.

However, unlike the traditional Q-learning method which uses a Q-value table to store the Q-values of each state-action, this implementation uses an approximate reinforcement learning method. This means that the agent uses a Q-value approximation function that takes an action-state as input and returns an approximate Q-value for that action-state. This approximation function is determined by a set of weights that are adjusted during learning.

The getWeights() method returns the current weights used for estimating the Q value. These weights are initialized to zero in the class's __init__() constructor.

The getQValue() method is used to estimate the Q value of a given action-state by multiplying the current weights by the characteristics of the action-state. These features are extracted using a feature extractor specified during agent creation. This feature extractor transforms an action-state into a feature vector, which is then used for Q-value estimation.

The update() method is used to update the weights of the Pacman agent based on the transition from the current state to the next state using the Q-learning update rule. This method is called on each state-action-to-next-state transition and takes as input the current state, the action performed, the next state, and the reward associated with the transition.

First, the method uses the feature extractor to obtain the feature vector corresponding to the action-state. Next, it calculates the maximum expected Q value for the next state by calling the computeValueFromQValues() method of the PacmanQAgent base class. This method returns the maximum value of the Q-value function for the next state. It also uses the getQValue() method to get the current Q value of the action-state.
Then, the method updates each weight based on the error between the observed reward and the difference between the actual Q value of the action state and the expected Q value of the next state. It updates the weights according to a gradient descent formula that multiplies each feature in the feature vector by a training term alpha and by the difference between the observed reward and the Q value estimate. Specifically, for each characteristic of the value function Q, it calculates:
`self.weights[feature] += self.alpha * (reward + self.discount * maxQFromNextState - actionQValue) * featureVector[feature]`

The final() method is called at the end of each game to determine if the agent has finished learning. If the agent has finished learning, it displays the final weights learned by the agent to allow verification and debugging. This method also calls the final method of the PacmanQAgent base class to perform any other necessary cleanup operations.

Tests

We get this result with this command : python3 pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 100 -l mediumGrid :

```
Average Score: 527.88
Scores:        529.0, 527.0, 529.0, 527.0, 529.0, 527.0, 529.0, 527.0, 527.0, 527.0, 527.0, 529.0, 527.0, 523.0, 527.0,
, 527.0, 529.0, 527.0, 529.0, 529.0, 527.0, 529.0, 527.0, 529.0, 529.0, 529.0, 529.0, 529.0, 527.0, 529.0, 529.0, 529.0
0, 529.0, 527.0
Win Rate:      50/50 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win,
 Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

We observe that as expected, we have very good results, here we have won 50/50 games. Moreover, we notice that we obtain a better score here than with the epsilon-greedy action.