Diego Sáez Mulero U137519
Amanda Pintado Lineros U137702

Project 3- Reinforcement Learning

Introduction

The purpose of this project is to implement the iteration value and Q-learning. You will first test your agents on Gridworld (of the class), then apply them to a simulated robot controller (Crawler) and Pacman.

Value Iteration

__init__: We update the values of all the states of the grid for each iteration. We use a vector to store the Q-value in each state.

```
self.mdp = mdp
self.discount = discount #gamma 0.9
self.iterations = iterations #Num de iteraciones
self.values = util.Counter() # A Counter is a dict with default 0

# Write value iteration code here
"*** YOUR CODE HERE ***"


for i in range(0,iterations):
    v = util.Counter()#vector de estados

    for state in self.mdp.getStates():
        action = self.getAction(state)#direccion en ese estado

        if action is not None:
            v[state] = self.getQValue(state,action)#Guardamos valor Q en ese estado del vector
    self.values = v #Guarda los valores de todos los estados en cada iteracion
```

getQValue returns the Q value of the action, this is performed by the function **computeQValueFromValues**:

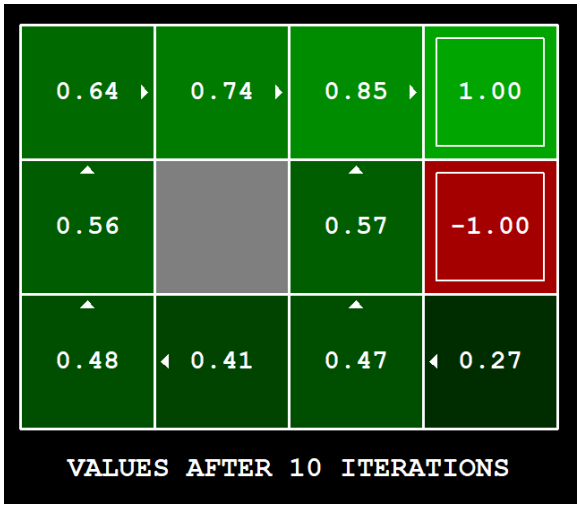
```
def computeQValueFromValues(self, state, action):
    """
      Compute the Q-value of action in state from the
      value function stored in self.values.
    """
    "*** YOUR CODE HERE ***"

    sumat = 0
    for (nState,prob) in self.mdp.getTransitionStatesAndProbs(state, action):
        sumat = sumat + prob * self.getValue(nState)#Sumatorio de valores (Prob*V(next state))



    if action == 'north':
        nState = (state[0],state[1]+1)
    elif action == 'south':
        nState = (state[0],state[1]-1)
    elif action == 'east':
        nState = (state[0]+1,state[1])
    else:
        nState = (state[0]-1,state[1])

    return ( self.mdp.getReward(state,action,nState) + self.discount * sumat )
   # util.raiseNotDefined()
```

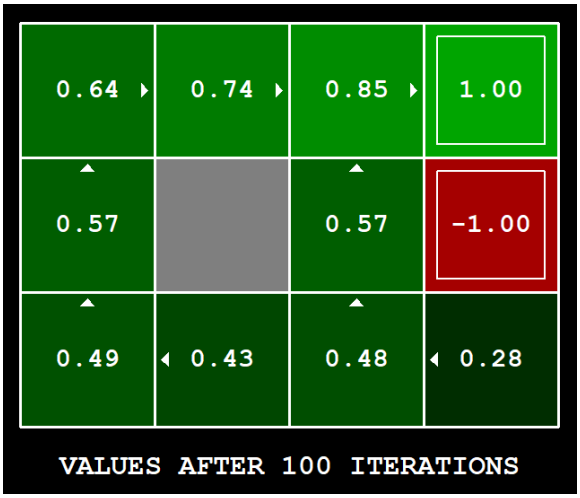
computeActionFromValues: It returns the best option of all the possible actions and this is called by **getAction** in each state and for all the iterations.

```
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 ***"

    #Devuelve la politica de cada estado (N,S,W,E)

    if self.mdp.isTerminal(state):
        return None
    maximAction = self.mdp.getPossibleActions(state)[0] #Primera accion

    q = self.getQValue(state,maximAction)#valor Q
    for action in self.mdp.getPossibleActions(state):#Recorremos todas las posibles acciones
        aux = self.getQValue(state,action)
        if q < aux:
            q = aux
            maximAction = action

    return maximAction #Devuelve la mejor accion
```

Diego Sáez Mulero U137519
Amanda Pintado Lineros U137702

We execute gridworld.py with different number of iterations and we observe that from 10 iterations the Policy converges.





Bridge Crossing Analysis

By setting the noise to 0, the agent can cross the bridge without falling off the cliff of (-100).

```
answerDiscount = 0.9

answerNoise = 0
```

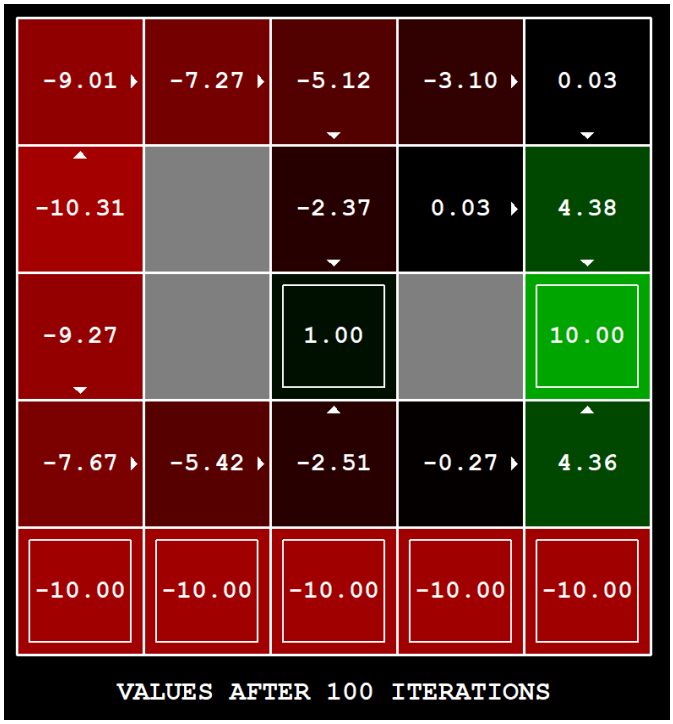
Diego Sáez Mulero U137519
Amanda Pintado Lineros U137702

Policies

A. Prefer the close exit (+1), risking the cliff (-10)

We have chosen these values since having such a large discount we need the noise to be low to encourage the dangerous side and we also need a negative reward large enough for the algorithm to choose the shortest path.

```
python gridworld.py -a value -i 100 -g DiscountGrid --discount 0.8 --noise 0.1 --livingReward -3.0
```



Diego Sáez Mulero U137519
Amanda Pintado Lineros U137702

B. Prefer the close exit (+1), but avoiding the cliff (-10)

Due to the low noise and negative reward, we needed a smaller number of discounts (for more iterations to converge) for the pacman to conclude that the long way to the reward is the best.

```
python gridworld.py -a value -i 100 -g DiscountGrid --discount 0.5 --noise 0.1 --livingReward -1.0
```

C. Prefer the distant exit (+10), risking the cliff (-10)

As we need him to go through the most dangerous place but also with the highest score (+10), we have to encourage through the reward and the noise that this action makes, therefore, the noise to a minimum and the reward positive enough to favor the movement.

```
python gridworld.py -a value -i 100 -g DiscountGrid --discount 0.8 --noise 0.0 --livingReward 1.0
```

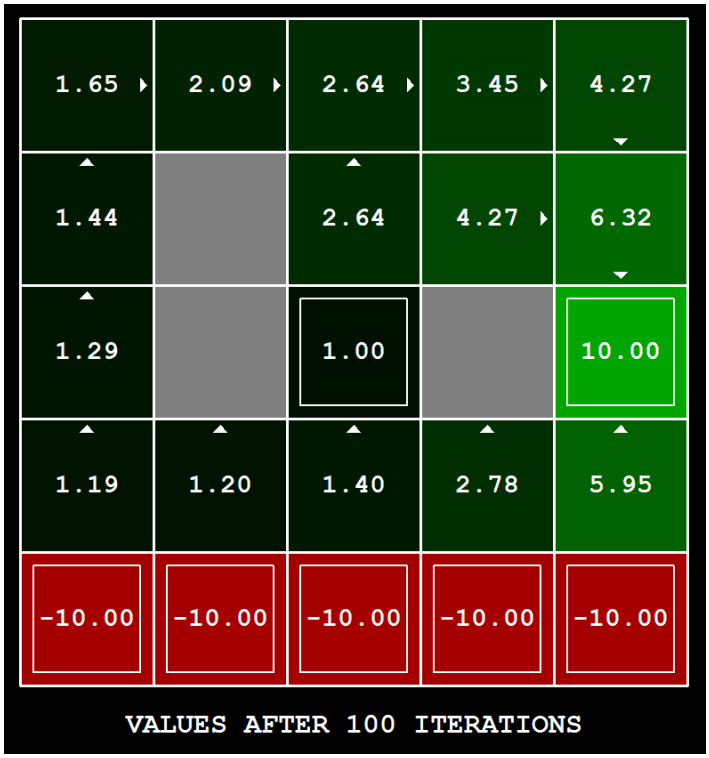
Diego Sáez Mulero U137519
Amanda Pintado Lineros U137702

D. Prefer the distant exit (+10), avoiding the cliff (-10)

The noise has to be large enough to prevent the path of the abyss and a positive reward that encourages movement to reach the exit of (+10).

```
python gridworld.py -a value -i 100 -g DiscountGrid --discount 0.8 --noise 0.5 --livingReward 0.2
```

Diego Sáez Mulero U137519
Amanda Pintado Lineros U137702



E. Avoid both exits and the cliff (so an episode should never terminate)

We have to force the pacman to stay in a loop, we have to tell him that staying walking is very good (much more than any final reward), and besides, he never finds his way to the final reward. For this we need a very small discount so that it does not exit the loop, since I do not care about the values of the following states.

```
python gridworld.py -a value -i 100 -g DiscountGrid --discount 0.01 --noise 0.9
--livingReward 20.0
```

Diego Sáez Mulero U137519
Amanda Pintado Lineros U137702

Q-Learning

computeValueFromQValues: We go through all the legal actions for each state and return the maximum value, in the case that it is a terminal state we return 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 ***"

    actionV = float('-inf')
    for action in self.getLegalActions(state):#(N,S,E,W)
        expectedQVal = self.getQValue(state, action)
        if actionV < expectedQVal:
            actionV = expectedQVal
    if actionV == float('-inf'):
        return 0
    return actionV #valor maximo de las acciones
```

computeActionFromQValues: In this case we are interested in returning the best policy of the next state

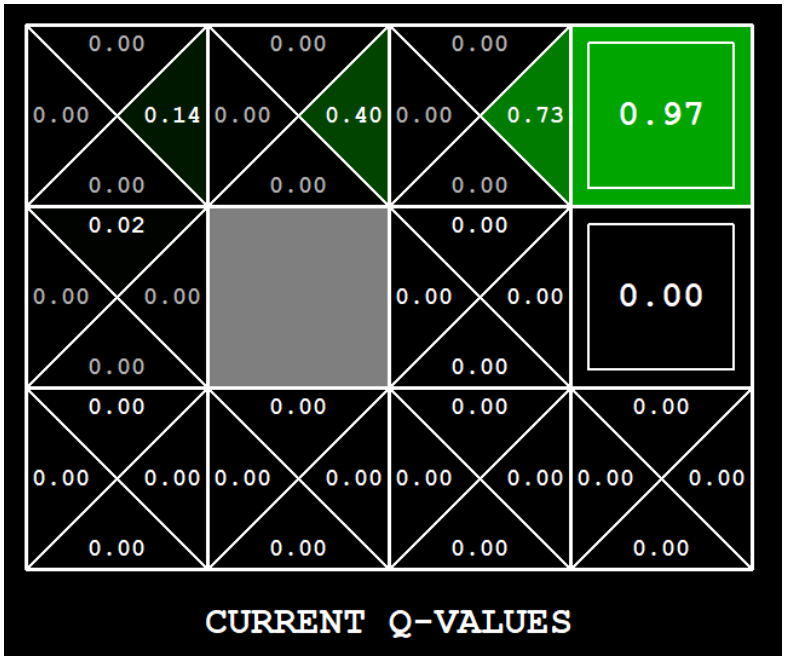
```
def computeActionFromQValues(self, state):
    """
      Compute the best action to take in a state.  Note that if there
      are no legal actions, which is the case at the terminal state,
      you should return None.
    """
    "*** YOUR CODE ***"

    legalActions = self.getLegalActions(state)
    if not legalActions:#Estado terminal None
        return None

    maxAction = legalActions[0]
    maxValue = self.qvalue[(state,maxAction)]


    for action in legalActions:
        aux = self.qvalue[(state, action)]
        if aux > maxValue:
            maxAction = action
            maxValue = aux

    return maxAction#la mejor accion
```

update: With this method we recover the old QValue and update its value.

```
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 ***"

    q = self.getQValue(state, action)

    self.qvalue[(state,action)] = q + self.alpha * ( reward + self.discount * self.getValue( nextState ) - q )
```

We manually move north and east in 5 iterations.

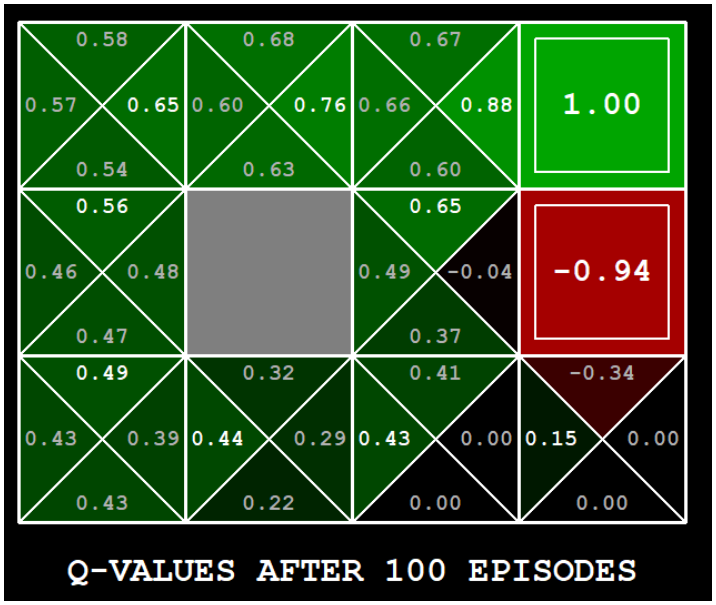
Diego Sáez Mulero U137519
Amanda Pintado Lineros U137702



Epsilon Greedy

We use the method **util.flipCoin(p)**, which returns True with probability p and False with probability 1-p.

```
if util.flipCoin(self.epsilon):#Random Prob. p = true , 1-p= false
    return random.choice(legalActions)

return self.getPolicy(state)
```



Diego Sáez Mulero U137519
Amanda Pintado Lineros U137702

Bridge Crossing Revisited

It's not possible. Pacman knows that going to the initial state implies a good reward, then it will always stay there, unless we apply a probabilistic epsilon factor. The larger the epsilon, the less pacman is likely to remain in the initial state. With such a large random factor, pacman may or may not cross the bridge. Therefore, it is impossible to guarantee that with such a large random factor, pacman could go anywhere without a brain and cross the bridge in just 50 steps with a probability greater than 99%.

```
def question6():
    #answerEpsilon = 0
    #answerLearningRate = None
    #return answerEpsilon, answerLearningRate
    return 'NOT POSSIBLE'
```

Q-Learning and Pacman

We apply a similar code as seen in some parts of Q-Learning, concretely what we seen in QLearningAgent. The parameters is changed and we assign the pacman to a concrete index of the class.

```
157     def __init__(self, epsilon=0.05,gamma=0.8,alpha=0.2, numTraining=0, **args):
158         """
159         These default parameters can be changed from the pacman.py command line.
160         For example, to change the exploration rate, try:
161             python pacman.py -p PacmanQLearningAgent -a epsilon=0.1
162 
163         alpha    - learning rate
164         epsilon  - exploration rate
165         gamma    - discount factor
166         numTraining - number of training episodes, i.e. no learning after these many episodes
167         """
168         args['epsilon'] = epsilon
169         args['gamma'] = gamma
170         args['alpha'] = alpha
171         args['numTraining'] = numTraining
172         self.index = 0  # This is always Pacman
173         QLearningAgent.__init__(self, **args)
```

Here we use the mentioned QLearningAgent method to use this action for the Pacman.

```
175     def getAction(self, state):
176         """
177         Simply calls the getAction method of QLearningAgent and then
178         informs parent of action for Pacman.  Do not change or remove this
179         method.
180         """
181         action = QLearningAgent.getAction(self,state)
182         self.doAction(state,action)
183         return action
```

Approximate Q-Learning

using again QLearningAgent as reference, we overwrite the method getQValue.

```
202     def getQValue(self, state, action):
203         """
204           Should return Q(state,action) = w * featureVector
205           where * is the dotProduct operator
206         """
207         "*** YOUR CODE HERE ***"
208 
209         val = 0
210         for key in self.featExtractor.getFeatures(state, action).keys():
211             val += self.weights[key] * self.featExtractor.getFeatures(state, action)[key]
212         return val
```

To make the updates and when we finish the game, we have to recalculate with past classes the weights, as well as taking note of the number of practices that have been done.

```
214     def update(self, state, action, nextState, reward):
215         """
216            Should update your weights based on transition
217         """
218         "*** YOUR CODE HERE ***"
219 
220         upd = (reward + (self.discount*self.getValue(nextState))) - self.getQValue(state, action)
221         for key in self.featExtractor.getFeatures(state, action).keys():
222             self.weights[key] = self.weights[key] + self.alpha*upd*self.featExtractor.getFeatures(state, action)[key]
223 
224 
225     def final(self, state):
226         "Called at the end of each game."
227         # call the super-class final method
228         PacmanQAgent.final(self, state)
229 
230         # did we finish training?
231         if self.episodesSoFar == self.numTraining:
232             # you might want to print your weights here for debugging
233             "*** YOUR CODE HERE ***"
234 
235             for weight in self.weights.keys():
236                 print "For weight: %s have value: %2.2f" % (str(weight), float(self.weights[weight]))
237             pass
```

Diego Sáez Mulero U137519
Amanda Pintado Lineros U137702