Inteligência Artificial
T3 - Relatório

Professor: Eduardo Bezerra
Aluno: Lucas Lima da Cruz

• Q1 - Value Iteration

Implementando o agente de iteração em *ValueIterationAgent* no arquivo valueIterationAgents.py e executando o seguinte comando:

```
python gridworld.py -a value -i 100 -k 10
```

Obtemos a seguinte tela de valores, em que o valor do estado inicial V(start) é igual a 0.49.

Após 10 rodadas de execução, obtemos uma média empírica dos valores das recompensas muito próxima do valor do estado inicial, como mostra a saída abaixo:

```
AVERAGE RETURNS FROM START STATE: 0.44936289000000007
```

Executando o agente para 5 iterações de valor com o comando abaixo:

```
python gridworld.py -a value -i 5
```

A saída produzida é a seguinte tela, idêntica à mostrada pelo enunciado:

- **Q2 - Travessia de Ponte**

O valor de desconto padrão de 0.9 indica que o agente terá a tendência de priorizar recompensas a longo prazo, o que é o desejado para que o agente atravesse a ponte e busque a recompensa mais distante. Assim, mantendo o valor padrão do desconto e reduzindo o ruído default para 0 - para que o agente não vá para o lado errado - é possível fazer com que o agente atravesse a ponte e chegue até a recompensa mais distante.

Executando `python gridworld.py -a value -i 100 -g BridgeGrid --discount 0.9 --noise 0.0` , obtemos a seguinte saída:

● Q3 - Políticas

Utilizando os seguintes valores para as variáveis de desconto, ruído e recompensa de viver, foi possível chegar à políticas ótimas em cada item:

- **3(a):** Preferir a saída mais próxima (+1), arriscando cair no precipício (-10)
  answerDiscount = 0.1
  answerNoise = 0
  answerLivingReward = 0

- **3(b):** Preferir a saída mais próxima (+1), evitando o precipício (-10)
  answerDiscount = 0.3
  answerNoise = 0.3
  answerLivingReward = 0

- **3(c):** Preferir a saída mais distante (+10), arriscando cair no precipício (-10)
  answerDiscount = 0.9
  answerNoise = 0
  answerLivingReward = 0

- **3(d):** Preferir a saída mais distante (+10), evitando o precipício (-10)
  answerDiscount = 0.9
  answerNoise = 0.2
  answerLivingReward = 0

- **3(c):** Evitar ambas as saídas (também evitando o precipício)
  answerDiscount = 0.5
  answerNoise = 0.5
  answerLivingReward = 0

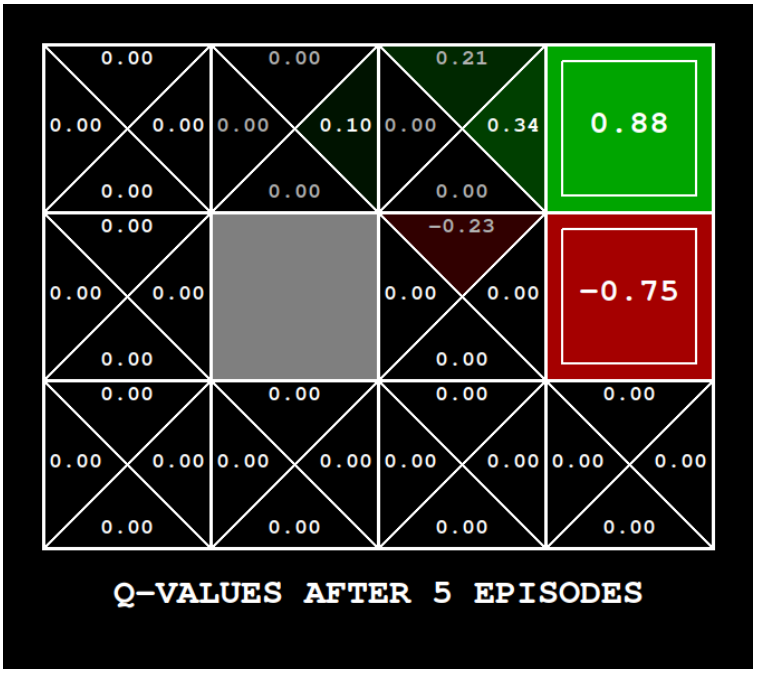
● Q6 - Q-Learning

As funções *update*, *computeValueFromQValues*, *getQValue*
e *computeActionFromQValues* foram implementadas na classe *QLearningAgent* do arquivo qlearningAgents.py.

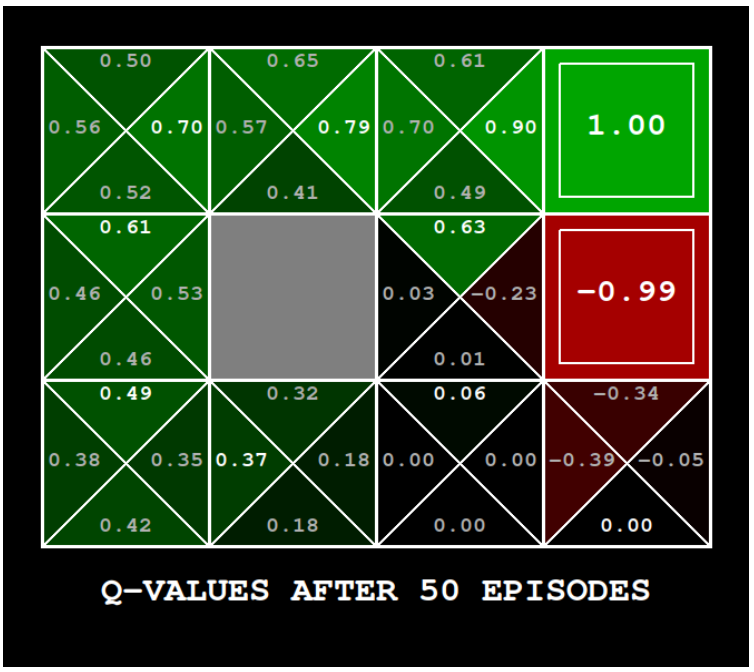
Para *computeActionFromQValues* em específico, utilizamos a função *random.choice()* para resolver empates dos valores de ações de forma aleatória, conforme indicação do enunciado e exemplificado abaixo:

```
def computeActionFromQValues(self, state):
        bestActions = [None]
        legalActions = self.getLegalActions(state)
        maxQValue = float('-inf')

        for action in legalActions:
          if self.getQValue(state, action) > maxQValue:
            maxQValue = self.getQValue(state,action)
            bestActions = [action]
          elif self.getQValue(state, action) == maxQValue:
            bestActions.append(action)

        return random.choice(bestActions)
```

Executando o agente com o comando python gridworld.py -a q -k 5 –m, é possível observar o agente aprendendo manualmente durante 5 episódios.



Aumentando o número de iterações para 50 com o comando python gridworld.py -a q -k 50 –m, observamos o aprendizado da política ótima:



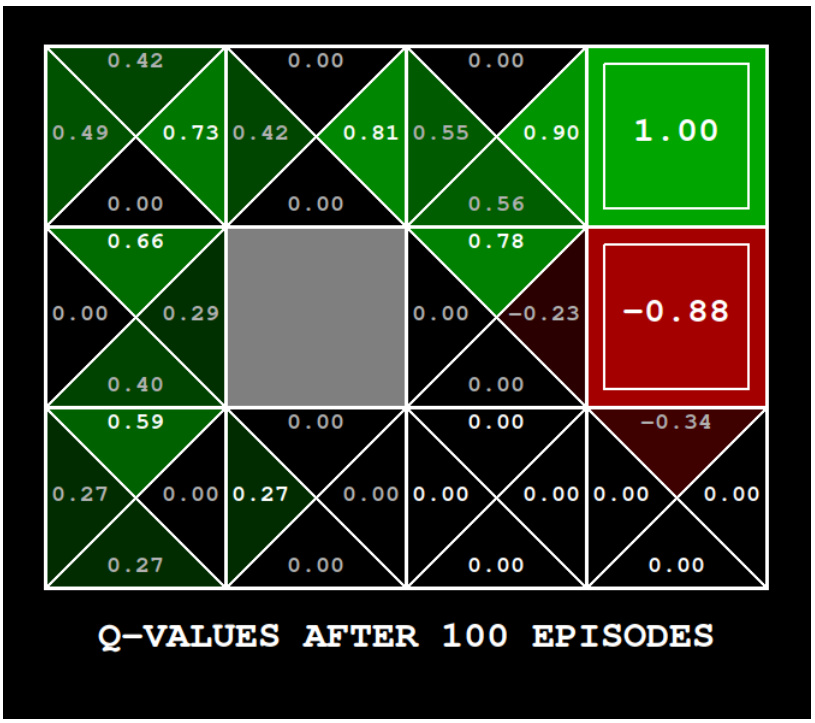
- **Q7 - ε-Greedy**

Utilizando a função *util.flipCoin(ε)* para simular uma variável aleatória binária com probabilidade ε de sucesso, implementamos o algoritmo de seleção de ações ε-Greedy para o agente Q-learning na função *getAction*:

```
def getAction(self, state):
        legalActions = self.getLegalActions(state)
        action = None

        if len(legalActions) == 0:
          return action

        if util.flipCoin(self.epsilon):
          action = random.choice(legalActions)
        else:
          action = self.computeActionFromQValues(state)

        return action
```

Assim, conseguimos definir as condições de *exploration x exploitation* para o agente no ambiente, com base no valor de probabilidade *self.epsilon*.

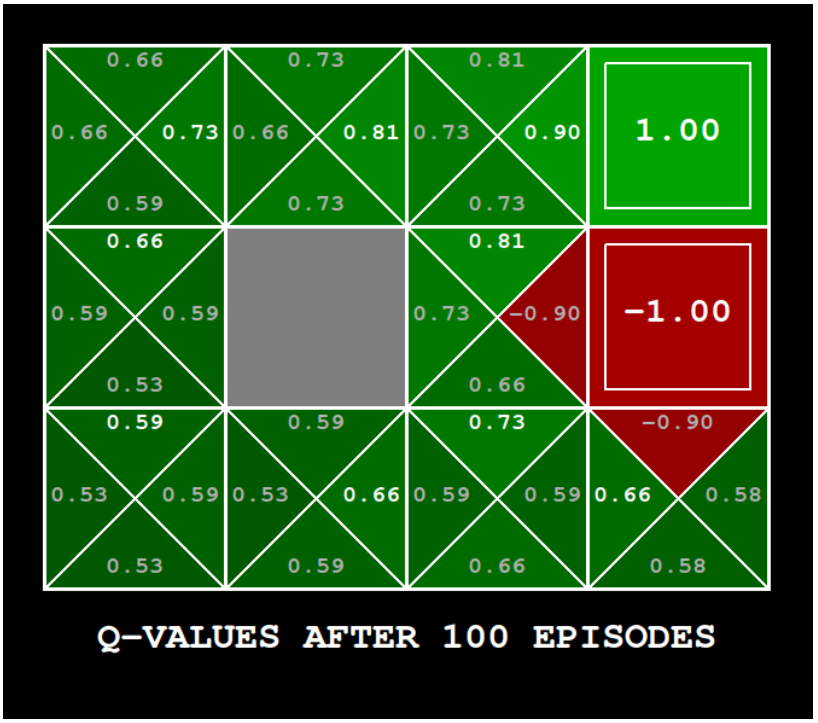
Realizando um experimento executando com o agente com epsilon = 0.1 com o comando `python gridworld.py -a q -k 100 --noise 0.0 -e 0.1`, obtemos:



Com a seguinte retorno para média empírica:

AVERAGE RETURNS FROM START STATE: 0.5220130498396316

Agora realizando um segundo experimento, alterando o valor de epsilon para 0.9, e executando o agente com o comando python gridworld.py -a q -k 100 --noise 0.0 -e 0.9, obtemos os seguintes resultados:



Com o retorno de média empírica das recompensas:

AVERAGE RETURNS FROM START STATE: -0.048746377221072945

A diferença de resultados é notável e esperada. Para o experimento com epsilon = 0.1, o agente explora muito menos o ambiente de forma aleatória, focando em seguir as melhores ações exploradas por ele anteriormente. Já para o segundo experimento, com alto valor de epsilon = 0.9, o agente foca muito mais em explorar o ambiente de forma aleatória, consequentemente descobrindo mais Q-Values do que o primeiro experimento, e tendo um retorno médio de recompensas menor do que o primeiro experimento devido a maior aleatoriedade de suas ações.

● Q8 - Revisitando a Travessia da Ponte

Executando python gridworld.py -a q -k 50 -n 0 -g BridgeGrid -e 1, verificamos que o agente não encontra a política ótima. Assim, não atravessando a ponte.

Com o valor de epsilon = 0, o agente também não atravessa a ponte, uma vez que segue apenas as melhores ações já aprendidas e vai em direção à recompensa mais próxima, que possui um valor menor que a recompensa da política ótima.

Testando com pequenos valores para a taxa de aprendizado, e maiores valores para epsilon, não foi possível fazer com que o agente aprenda a política ótima ao longo das 50 iterações, assim retornando 'NOT POSSIBLE'.

● Q9 - Q-Learning e Pacman

Executando o agente de Pacman Q-learning para 2010 jogos com o comando python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid. Os primeiros 2000 jogos são reservados para o treinamento do agente. Para os últimos 10 jogos, obtemos os seguintes resultados:

```
Training Done (turning off epsilon and alpha)
---------------------------------------------
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 503
```

```
Average Score: 500.6
Scores:       499.0, 499.0, 503.0, 503.0, 503.0, 499.0, 499.0, 499.0, 499.0, 503.0
Win Rate:      10/10 (1.00)
Record:       Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

Já para um experimento num labirinto maior, com o comando `python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l mediumGrid` , obtemos resultados piores:

```
Pacman died! Score: -488
Pacman died! Score: -496
Pacman died! Score: -504
Pacman died! Score: -487
Pacman died! Score: -488
Pacman died! Score: -544
Pacman died! Score: -507
Pacman died! Score: -510
Pacman died! Score: -522
Pacman died! Score: -499
Average Score: -504.5
Scores:       -488.0, -496.0, -504.0, -487.0, -488.0, -544.0, -507.0, -510.0, -522.0, -499.0
Win Rate:      0/10 (0.00)
Record:       Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss
```

• Q10 - Q-Learning Aproximado

Implementando as funções getQValue e update da classe ApproximateQAgent no arquivo qlearningAgents.py:

```
def getQValue(self, state, action):

        features = self.featExtractor.getFeatures(state,action)
        qvalue = 0

        for feature in features.keys():
          qvalue += self.weights[feature] * features[feature]
        return qvalue
```

```
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.alpha*difference*features[feature]
```

Com essa implementação, o agente melhora seu desempenho em relação ao agente da classe QLearningAgent para labirintos maiores. Testando o agente com o comando:

```
python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumGrid
```

Obtemos os seguintes resultados:

```
Beginning 50 episodes of Training
Training Done (turning off epsilon and alpha)
----------------------------------------
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 523
Pacman emerges victorious! Score: 525
Pacman emerges victorious! Score: 527
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 525
Pacman emerges victorious! Score: 523
Pacman emerges victorious! Score: 523
Pacman emerges victorious! Score: 527
Average Score: 526.0
Scores:        529.0, 523.0, 525.0, 527.0, 529.0, 529.0, 525.0, 523.0, 523.0, 527.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

Para o labirinto mediumClassic, com o comando:

```
python3 pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumClassic
```

Os resultados:

```
Beginning 50 episodes of Training
Training Done (turning off epsilon and alpha)
----------------------------------------
Pacman emerges victorious! Score: 1335
Pacman emerges victorious! Score: 1333
Pacman emerges victorious! Score: 1323
Pacman emerges victorious! Score: 1326
Pacman emerges victorious! Score: 1331
Pacman emerges victorious! Score: 1340
Pacman emerges victorious! Score: 1344
Pacman emerges victorious! Score: 1334
Pacman emerges victorious! Score: 1327
Pacman emerges victorious! Score: 1329
Average Score: 1332.2
Scores:        1335.0, 1333.0, 1323.0, 1326.0, 1331.0, 1340.0, 1344.0, 1334.0, 1327.0,
1329.0
Win Rate:      10/10 (1.00)
```

Record:	Win, Win, Win, Win, Win, Win, Win, Win, Win, Win