Algoritmo Q-Learning no ambiente Agent0

Beatriz Silva (2019108992@uac.pt), Pedro Sousa (2019101451@uac.pt), Salif Faustino (20172005@uac.pt)

Resumo

Este trabalho foi realizado no âmbito da disciplina de Inteligência Artificial e teve como objetivo a exploração e interação entre um agente e um ambiente. O software consiste num ambiente onde o agente se consegue movimentar. Com base nas experiências efetuadas com o algoritmo escolhido, foi possível interpretar e perceber os diversos cenários a que o agente foi submetido.

Introdução

Na cadeira de Inteligência Artificial, do 3º Ano da Licenciatura em Informática, foi-nos proposto a interpretação e implementação de um algoritmo no Agente0, sendo assim, decidimos utilizar algoritmo **Q-Learning.** A versão utilizada neste projeto foi o Agent0_Minotauro_Reinforcement, que permite explorar a interação entre um agente e um ambiente. Este software consiste num ambiente com um tabuleiro retangular de casas quadradas onde o agente se pode movimentar. De modo a movimentar-se, o agente pode deslocar-se em frente ou mudar de direção e dependendo da sua capacidade, pode também inspecionar o tabuleiro e as suas casas. O agente pode utilizar diversos algoritmos para chegar ao seu objetivo e é possível observar as casas com diversas cores. A interação entre o agente e o ambiente é comandada através de um cliente e acontece no servidor. A versão final do programa pode ser encontrada no github, através do link seguinte:
https://github.com/SalifNTC/projeto-2-g10
Link do vídeo:
https://youtu.be/sTyNQGC9T6M

Descrição do Algoritmo

O algoritmo **Q-Learning** é um algoritmo de aprendizagem por reforço, com uma política que procura encontrar a melhor ação a ser executada no estado atual. O **Q-Learning** é um algoritmo off-policy, podendo assim atualizar as funções de valor estimado usando ações que ainda não foram experimentadas. O **Q-Learning** aprende a política ideal, mesmo quando as ações são selecionadas de acordo com uma política mais exploratória ou até aleatória.
O **"Q"** em que **Q-Learning** representa o quão útil uma determinada ação é para ganhar alguma recompensa.
Quando o algoritmo é utilizado é criada uma tabela designada por **Q-table** que obedece o seguinte formato: [estado, ação]. Os valores são inicializados a zero. A atualização desses valores-q sempre que ocorre um episódio. Esta tabela torna-se uma referência para o agente selecionar a melhor ação com base no valor q.

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$$

Q learning algorithm

For each s, a initialize the table entry $\hat{Q}(s, a)$ to zero.
Observe the current state s
Do forever:

- Select an action a and execute it
- Receive immediate reward r
- Observe the new state s'
- Update the table entry for $\hat{Q}(s, a)$ as follows:

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

- $s \leftarrow s'$

Figura 1- Algoritmo Q-Learning

Implementação do Q-Learning

A tabela ***(Figura 2) Q – table*** é iniciada com valores a zero. Foi criada uma lista auxiliar adaptada ao mapa utilizado com as localizações dos obstáculos, posição inicial, e objetivo. Esta lista é alterada quando modificamos o mapa. Isto deveu-se à falta de tempo e experiência para atualizar esta lista com as localizações dos obstáculos, sendo assim feita de maneira rudimentar no código.

```
def initializeTable(self):
    lista_aux = [(0,0),(0,1),(0,1),(0,2),(0,3),(0,4),(1,0),(1,4),(2,0),(2,4),(3,0),(3,4),(4,4),
    (4,0),(5,0),(5,4),(6,0),(6,4),(7,0),(7,1),(7,2),(7,3),(7,4)]
    for x in range(self.maxCoord[0]):
        for y in range(self.maxCoord[1]):
            if (x,y) not in lista_aux:
                self.qLearningTable[(x,y)] = [0,0,0,0]
```

Figura 2 – Inicialização da Q-Table

Atualização ***(Figura 3)*** da tabela utilizado a fórmula reward = reward(local anterior) + 0.9*reward(local). Isto ocorre depois do agente percorrer o caminho aleatoriamente.

```
def updateQLearningTable(self, path):
    self.rewards = self.getRewards()
    for i in range(len(path) -1, -1,-1):
        if self.rewards[path[i][0]][path[i][1]] == 0:
            if path[i][1] == (path[i+1][1] +1) :
                self.qLearningTable[(path[i][0], path[i][1])][0] = (0.9 * self.rewards[path[i+1][0]][path[i+1][1]])
                self.rewards[path[i][0]][path[i][1]] = (0.9 * self.rewards[path[i+1][0]][path[i+1][1]])
            elif path[i][1] == (path[i+1][1] -1) :
                self.qLearningTable[(path[i][0], path[i][1])][1] = (0.9 * self.rewards[path[i+1][0]][path[i+1][1]])
                self.rewards[path[i][0]][path[i][1]] = (0.9 * self.rewards[path[i+1][0]][path[i+1][1]])
            elif path[i][0] == (path[i+1][0] -1) :
                self.qLearningTable[(path[i][0], path[i][1])][2] = (0.9 * self.rewards[path[i+1][0]][path[i+1][1]])
                self.rewards[path[i][0]][path[i][1]] = (0.9 * self.rewards[path[i+1][0]][path[i+1][1]])
            elif path[i][0] == (path[i+1][0] +1) :
                self.qLearningTable[(path[i][0], path[i][1])][3] = (0.9 * self.rewards[path[i+1][0]][path[i+1][1]])
                self.rewards[path[i][0]][path[i][1]] = (0.9 * self.rewards[path[i+1][0]][path[i+1][1]])
        elif self.rewards[path[i][0]][path[i][1]] > 0 and self.rewards[path[i][0]][path[i][1]] != 100 :
            if (0.9 * self.rewards[path[i+1][0]][path[i+1][1]]) > self.rewards[path[i][0]][path[i][1]]:
                if path[i][1] == (path[i+1][1] +1) :
                    self.qLearningTable[(path[i][0], path[i][1])][0] = (0.9 * self.rewards[path[i+1][0]][path[i+1][1]])
                    self.rewards[path[i][0]][path[i][1]] = (0.9 * self.rewards[path[i+1][0]][path[i+1][1]])
                elif path[i][1] == (path[i+1][1] -1) :
                    self.qLearningTable[(path[i][0], path[i][1])][1] = (0.9 * self.rewards[path[i+1][0]][path[i+1][1]])
                    self.rewards[path[i][0]][path[i][1]] = (0.9 * self.rewards[path[i+1][0]][path[i+1][1]])
                elif path[i][0] == (path[i+1][0] -1) :
                    self.qLearningTable[(path[i][0], path[i][1])][2] = (0.9 * self.rewards[path[i+1][0]][path[i+1][1]])
                    self.rewards[path[i][0]][path[i][1]] = (0.9 * self.rewards[path[i+1][0]][path[i+1][1]])
                elif path[i][0] == (path[i+1][0] +1) :
                    self.qLearningTable[(path[i][0], path[i][1])][3] = (0.9 * self.rewards[path[i+1][0]][path[i+1][1]])
                    self.rewards[path[i][0]][path[i][1]] = (0.9 * self.rewards[path[i+1][0]][path[i+1][1]])
```

Figura 3 – Atualização da Q-Table

Função utilizada ***(Figura 4)*** para criar as setas no mapa, baseadas no valor das recompensas na tabela. As setas apontam no sentido do valor máximo das recompensas.

```
def addServerQtableArrows(self):
    lista_aux = [self.getGoalPosition(),self.getSelfPosition()]
    for x, y in self.qLearningTable:
        if (x,y) not in lista_aux:
            aux = self.qLearningTable[(x,y)].index(max(self.qLearningTable[(x,y)]))
            self.markArrow(aux, x, y)
```

Figura 4 – Adição de setas ao mapa

Na ***Figura 5*** o agente inicia a tabela da ***Figura 2,*** a posição do objetivo e o número de interações a executar (Neste caso, 100). Aqui, este executa ações aleatórias até atingir o objetivo, onde faz update da tabela **(*Figura 3*)**. No final é apresentada a tabela final e apresentadas as setas que indicam o caminho a seguir para o objetivo em cada ponto explorado ***(Figura 4)***.

```
if agent.getConnection() != -1:
    estados = ["north", "east", "west", "south"]
    aux = 0
    goalPosition = agent.getGoalPosition()
    obstacles = agent.getObstacles()
    path = []
    rewards = agent.getRewards()
    agent.initializeTable()
    #estado_anterior = None
    interactions = 100
    while aux != interactions:
        estado = random.randint(0,3)
        """
        if previous_estado != estado:
            if estado == 0:
                previous_estado = 3
            elif estado == 1:
                previous_estado = 2
            elif estado == 2:
                previous_estado = 1
            elif estado == 3:
                previous_estado = 0
        """
        agent.c.execute("command", estados[estado])
        path.append(agent.getSelfPosition())
        if agent.getSelfPosition() == goalPosition:
            agent.c.execute("command", "home")
            agent.updateQLearningTable(path)
            path = []
            aux += 1
    q_table = agent.get_q_table()
    print("QTable:\n\n" + str(q_table))
    agent.addServerQtableArrows()
    input("Press to Stop")
```

Figura 5 – Execução e pesquisa realizada pelo agente

Experiências Realizadas ou Exemplos de Aplicação

No conjunto de experiências apresentadas consideramos três estados fundamentais:

- Agente no ambiente;
- Agente após execução do algoritmo com 5 iterações;
- Agente após execução do algoritmo com 50 iterações**.**

Experiência 1, mundo sem obstáculos/target:

Nesta experiência ***(figura 6)***, o agente encontra-se inserido num mundo sem obstáculos/ target, o que faz com que, com a implementação do algoritmo **Q-learning,** o agente percorra o mapa sem ter em conta os mesmos.

Com base nas imagens da ***figura 6***, é possível perceber que os "caminhos" indicados após a implementação do algoritmo variam consoante ao número de iterações, ou seja, se as interações realizadas forem poucas, o agente pode não ter dados relativos a determinadas casas. Tal como se pode verificar, na imagem central, com poucas interações o agente não denota o caminho que

deveria fazer na casa no canto superior esquerdo seria diretamente para o objetivo. Isto deve-se, provavelmente, ao agente não ter percorrido essa casa. No entanto, na imagem mais à direita, o agente já identifica propriamente o caminho a executar.

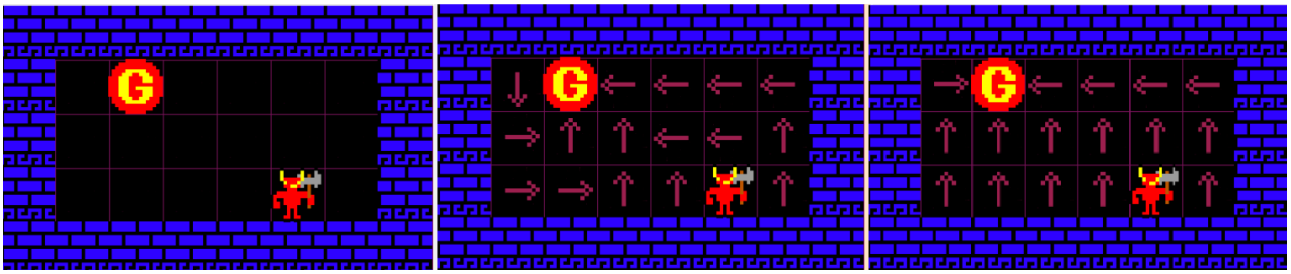

Figura 6 – Mundo sem obstáculos / Target

Experiência 2, mundo com obstáculos/target:

Nesta experiência **(figura 7)**, o agente encontra-se inserido num mundo com obstáculos/target.
A grande particularidade nesta figura foi a utilização do target, onde é possível perceber que com a ocorrência dos episódios o algoritmo não coloca a seta na posição onde encontra-se o target (imagem central, imagem mais a direita).

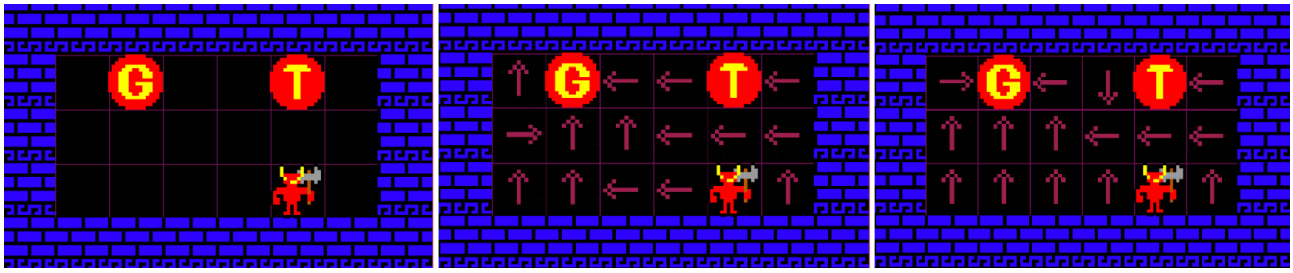

Figura 7 – Mundo com obstáculos / Target

Discussão e Conclusão

Acreditamos ter conseguido alcançar o nosso objetivo na realização do projeto, obtendo uma boa simulação para encontrar o objetivo. O agente atualiza os valores da **Q-Table** ao fim de cada episódio, resultando numa tabela que produz resultados esperados ao fim de muitos episódios.
Para um pequeno número de episódios, o agente é capaz de cometer ligeiros erros nos caminhos mais eficientes, sendo que, por isso, é aconselhável ter um maior número de episódios.
No entanto, devido à política utilizada ser aleatória, é possível haver uma divergência de resultados para mundos maiores ou com a existência de diversos objetivos.

Bibliografia
Slides fornecidos pelo professor José Cascalho.
Livro Machine Learning - Tom Mitchell