TP2 - Introdução à Inteligência Artificial - Q-Learning - Documentação

Nome: Rodrigo Ferreira Araújo — Matrícula: 2020006990

Julho 2023

1 Algoritmos e Estruturas de Dados

Nesta seção vamos tratar acerca da implementação dos métodos e lógicas principais e as e estruturas de dados empregadas para a implementação do *Q-Learning* no arquivo fonte (único) TP2.py.

1.1 Constantes e Variáveis Globais

1.1.1 Constantes

Casas do Grid:

1. **GROUND:** ***int***: Um inteiro igual a **0** de modo que um valor 0 na posição [i, j] no Grid significa que [i, j] é uma "casa" válida para o agente se mover, e está vazia no momento.
2. **AGENT:** ***int***: Um inteiro igual a **10** de modo que um valor 10 na posição [i, j] no Grid significa que o agente está em [i, j].
3. **OBSTACLE:** ***int***: Um inteiro igual a **-1** de modo que um valor -1 na posição [i, j] no Grid significa que há um obstáculo em [i, j]. Nesse sentido, o agente não poderá se mover para este [i, j].
4. **GOAL:** ***int***: Um inteiro igual a **7** de modo que um valor 7 na posição [i, j] no Grid significa que [i, j] é um estado terminal de recompensa +1 caso o agente o atinja ("ouro").
5. **DEFEAT:** ***int***: Um inteiro igual a **4** de modo que um valor 4 na posição [i, j] no Grid significa que [i, j] é um estado terminal de recompensa -1 caso o agente o atinja ("buraco").

Ações:

1. **UP:** ***char***: Um caracter 'c' (cima) que simboliza a ação de tentar se mover uma casa para cima.
2. **DOWN:** ***char***: Um caracter 'b' (baixo) que simboliza a ação de tentar se mover uma casa para baixo.

3. **LEFT: *char***: Um caracter 'e' (esquerda) que simboliza a ação de tentar se mover uma casa à esquerda.

4. **RIGHT: *char***: Um caracter 'd' (direita) que simboliza a ação de tentar se mover uma casa à direita.

5. **NONE: *char***: Um caracter 'n' (nada) que simboliza uma situação não aplicável, ou seja, ações a serem feitas nos obstáculos e nas casas terminais, uma vez que o intuito é de que o agente reinicie a exploração após atingir um estado terminal.

1.1.2 Variáveis Globais

1. **Grid: int$_{(\mathbf{N},\mathbf{N})}$**: Matriz N x N de entrada. Representa as casas e o estado corrente do Grid.

2. **nIters: *int***: Inteiro que representa o número de iterações que o *Q-Learning* deve executar (parâmetro **i** de entrada).

3. **learningRate: *float***: Taxa de aprendizado do *Q-Learning* (parâmetro **a** de entrada).

4. **discountFactor: *float***: Fator de desconto do *Q-Learning* (parâmetro **g** de entrada).

5. **stdReward: *float***: Recompensa padrão dos estados não terminais (e que não são obstáculos) do Grid (parâmetro **r** de entrada).

6. **epsilon: *float***: Fator [0, 1) opcional do algoritmo que reflete a chance do agente executar uma ação aleatória (parâmetro **e** de entrada).

7. **data_list: list(int$_{(\mathbf{N},\mathbf{N})}$)**: Array de estados diferentes do Grid em sequência, de modo que cada item desta lista corresponderá a um frame do GIF a ser gerado.

1.2 *class* QSlot

Esta classe encapsula dados e operação úteis à cada casa do grid Q do algoritmo Q-Learning, isto é, diferente do **Grid**, cada casa s do grid Q contém os valores de $Q(s, a)$ para cada ação a possível neste s, bem como o seu valor de recompensa r. A operação em questão seria a própria função de atualização dos valores de $Q(s, a)$ de acordo com o *Q-Learning*.

1.2.1 Atributos:

1. **id: *int***: Casa correspondente deste estado no Grid de entrada: **AGENT**, **OBSTACLE**, **GROUND**, **GOAL** ou **DEFEAT**.

2. **av: *dict(char $\mapsto$ float)***: Dicionário que mapeia **a**ction $\mapsto$ **v**alue, ou seja, mapeia as ações possíveis deste estado s (**UP**, **DOWN**, **LEFT** ou **RIGHT**) para o seu respectivo valor $Q(s, a)$.

3. **reward: *float***: Valor de recompensa para este estado. Caso seja **GROUND**, é **stdReward**, caso seja **GOAL** é +1, caso seja **DEFEAT** é -1 e caso seja **OBSTACLE**, é 0.

1.2.2 Métodos:

1. **updateQValue(action: *char*, sNextBestActionQValue: *float*, reward: *float*) $\mapsto$ void**:
   Método que executa a atualização dos valores *Q(s,***action***)* de acordo com o algoritmo de *Q-Learning*. **sNextBestActionQValue** é o $max_{a' \in A}(Q(s'a'))$, ou seja, o maior valor de Q para o próximo estado *s'* em que o agente caiu, e **reward** é a recompensa de *s*. Portanto, o método realiza a atualização:

   *Q(s,****action****)* += **a** * (**reward** + **g** * **sNextBestActionQValue** - *Q(s,****action****)*).

   Lembrando que **a** = **learningRate** e **g** = **discountFactor**.

1.3 class QLearn

Esta classe implementa a lógica principal do *Q-Learning*. Possui atributos globais ao problema em si, como o grid de valores Q, e métodos auxiliares para a iteração principal do algoritmo, que consta essencialmente em realizar uma ação e atualizar os valores de Q para a ação realizada.

1.3.1 Atributos:

1. **QGrid: QSlot$_{(\mathbf{N},\mathbf{N})}$**: Matriz de QSlots, armazena os valores $Q(s, a)$ para cada casa no grid de entrada.

2. **initialSlot: [int, int]**: Lista de dois inteiros que armazenam as coordenadas [linha, coluna] da posição inicial do agente.

3. **s_xy: [int, int]**: Lista de dois inteiros que armazenam as coordenadas [linha, coluna] da posição corrente do agente.

4. **actionsList: *list(char)***: Uma lista literal contendo [**UP**, **DOWN**, **LEFT**, **RIGHT**]. Sua utilidade será explicada em breve.

5. **totalReward: *float***: Contador global de recompensas. É incrementado de **r** a cada estado que o agente passa. Será útil para calcular a recompensa média.

1.3.2 Métodos:

1. **initializeQ() $\mapsto$ void**:
   Método que inicializa os valores de **QGrid** de acordo com o **Grid**. O campo **GOAL** recebe **reward** = +1, o campo **DEFEAT** recebe **reward** -1 e os demais campos **GROUND** recebem **reward** = **stdReward**, com exceção dos campos **OBSTACLE**, que recebem 0. Todos os valores $Q(s, a)$ são inicializados com 0. Além disso, no campo **AGENT**, inicializamos **initialSlot** de acordo, e **s_xy** = **initialSlot**.

2. **slip(action: *char*) $\mapsto$ action: *char***:
   Método que, a partir de uma ação já escolhida pelo agente, aplica uma chance de "escorregar". Com uma chance de 80%, os "atuadores" do agente não escorregam e a ação escolhida é a ação executada. Com uma chance de 10%, os atuadores do agente escorregam para

uma das duas ações adjacentes à escolhida, desse modo, o agente nunca "escorrega para trás". Por exemplo, caso a ação escolhida seja **UP**, com 10% de chance o agente poderá escorregar para **RIGHT** ou **LEFT**.

3. **randomAction() ↦ action: *char***:
   Método que, com um comando *action = actionsList[random.randint(0,3)]*, escolhe uma ação aleatória a partir das ações possíveis. Note que essa escolha aleatória também está sujeita a "escorregar".

4. **bestAction(s: *QSlot*, toSlip: *bool*) ↦ action: *char***:
   A partir de um estado (**QSlot**) **s**, seleciona a ação a que maximiza os valores $Q(\mathbf{s}, a)$. Após a escolha, o agente estará sujeito a escorregar somente se **toSlip** = **True**.

5. **executeAction(action: *char*) ↦ *int***:
   Movimenta o agente no **Grid** e atualiza sua posição corrente **s_xy** de acordo com a ação do parâmetro. Se o estado que o agente irá se movimentar ultrapassa o limite do **Grid** ou é um **OBSTACLE**, o valor -1 é retornado e o agente permanece no mesmo estado, caso contrário, o *label* do estado que o agente se moveu é retornado (**GROUND**, **GOAL** ou **DEFEAT**).

6. **QIter() ↦ *void***:
   A partir dos métodos descritos acima, executa uma iteração do *QLearning*. Primeiro, uma ação é escolhida. Se o fator **epsilon** está presente, uma ação aleatória (**randomAction**) é escolhida com chance de (**epsilon** * 100)%, caso contrário, **bestAction(s, True)** é aplicado. Em seguida, a ação resultante é executada e o novo estado do **Grid** é registrado em **data_list**. A recompensa **r** desse novo estado **s'** é observada e **totalReward** += **r**.

   Após executar a ação, aplicamos a função de atualização dos $Q(s, a)$ de acordo com **updateQValue(action, sNextBestActionQValue, reward)**.

   Caso **s'** seja um **GROUND**, aplicamos a função de atualização normalmente: **action** é a ação feita, **sNextBestActionQValue** é obtido através de **bestAction(s, False)**, recuperando o $max_{a' \in A}(Q(s'a'))$ e **reward** é a recompensa **r** do novo estado.

   Caso **s'** seja um terminal, verificado pelo retorno de **executeAction**, aplicamos a função de atualização, de modo que **reward** = **sNextBestActionQValue** = recompensa do estado terminal (+1 para **GOAL**, -1 para **DEFEAT**). Ademais, o agente atingir um estado terminal significa o fim de um episódio, ou seja, o agente deve voltar para a posição inicial e um *frame* extra do **Grid** indicando esta volta é colocado em **data_list**.

1.4 Programa Principal (main)

O programa principal, executado como

python3 TP2.py <inputFileName>.txt <outputFileName>

, lê e processa a entrada, atribuindo os valores corretos às variáveis globais explicadas. O primeiro estado do **Grid** é colocado em **data_list** e é preenchido com **nIters** iterações da função **QIter**.

Para a geração da imagem final, como proposto, o **Grid** é modificado de modo que cada estado **s** recebe o seu maior valor $Q(s, a)$ após o fim das iterações, juntamente com um *label* **a** indicando esta ação. Estados terminais recebem seu valor de recompensa e obstáculos recebem 0. O *heatmap* resultante é salvo no caminho saidas/outputFileName_acoes.png.

Por fim, para gerar o GIF, os métodos e parâmetros utilizados são os que constam na especificação, com um toque extra: dois *frames* com os valores zerados foram adicionados ao fim do GIF para identificar melhor quando ele foi finalizado. O GIF será salvo no caminho saidas/outputFileName.gif.

2 Análise dos Resultados

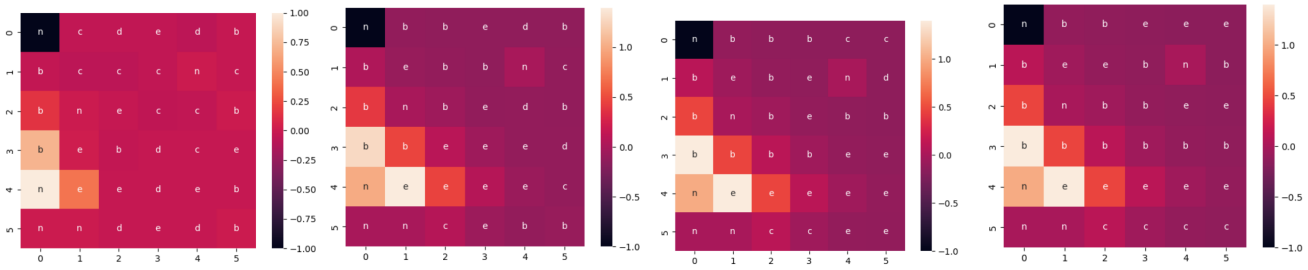
Para a análise de resultados, iremos usar os exemplos fornecidos no fórum de avisos da disciplina como base e avaliaremos o comportamento do agente apresentado no GIF e a imagem relatando os melhores movimentos para cada estado do **Grid**. Nesse sentido, iremos analisar os impactos isolando quatro variações dos parâmetros do problema (número de iterações, taxa de aprendizado, fator de desconto, recompensa e fator *epsilon-greedy*).

2.1 Variando o Número de Iterações

Estas observações foram feitas com testes extensivos para os três exemplos fornecidos, tomando o número original de iterações, depois este número x 5, depois x 10 e depois x 100. Contudo, para efeitos de exemplificação, para este os demais parâmetros, vamos apresentar apenas a imagem resultante do terceiro exemplo, cujo *input* padrão é:

```
300 0.3 0.4 -0.09 0.8
6
4 0 0 0 0 0
0 0 10 0 -1 0
0 -1 0 0 0 0
0 0 0 0 0 0
7 0 0 0 0 0
-1 -1 0 0 0 0
```

2.1.1 Resultados



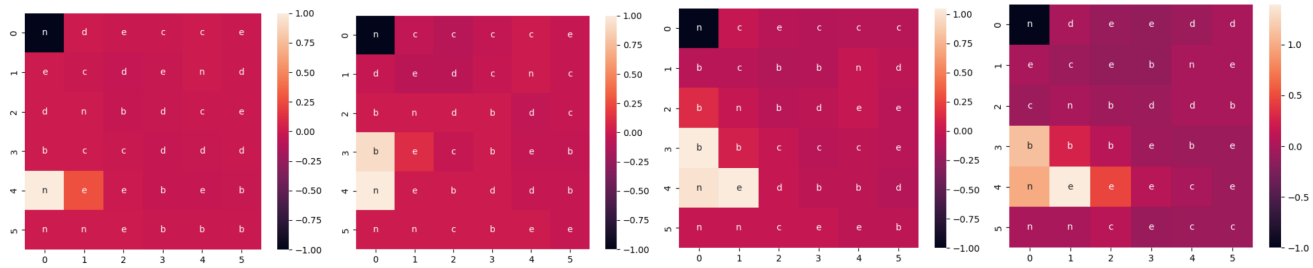
(a) 300 Iterações (b) 1500 Iterações (c) 3000 Iterações (d) 30000 Iterações

2.1.2 Análise

De longe, é o fator que mais influencia no rendimento do agente para atingir o **GOAL** e evitar o **DEFEAT**. Com um maior número de iterações, mais estados ao redor de **GOAL** possuem recompensas melhores para as ações razoáveis, isto é, aquelas que levam o agente na direção genéria do objetivo. Ademais, é notório o fato de que o número de passos de cada episódio decresce com o aumento do número de iterações, devido ao fato de que um agente que se movimenta mais, "espalha" mais e melhor os valores de recompensa para os estados próximos ao **GOAL** e, bem como, forma políticas melhores e mais uniformes.

2.2 Variando a Taxa de Aprendizado

2.2.1 Resultados



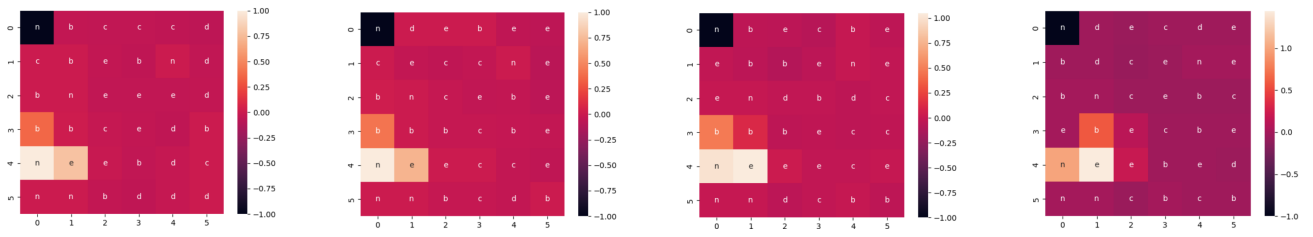
(e) taxa = 0.1 (f) taxa = 0.3 (g) taxa = 0.5 (h) taxa = 0.8

2.2.2 Análise

Como esperado, a "inteligência" do agente cresce com a taxa de aprendizado, isolando todos os outros fatores. Isto é, a taxas de aprendizado pequenas, o agente apresenta comportamento aparentemente aleatório e uniforme em estados longe do **GOAL**, ou seja, não há grandes diferenças nos valores $Q(s, a)$ e as melhores ações não seguem um padrão razoável. Com o aumento da taxa, mais estados mais próximos ao **GOAL** recebem uma maior valorização, refletindo o aprendizado do agente, bem como há um melhor contraste para as casas mais distantes do **GOAL**, com menores valores de $Q(s, a)$ e ações mais bem direcionadas ao **GOAL**.

2.3 Variando o Fator de Desconto

2.3.1 Resultados



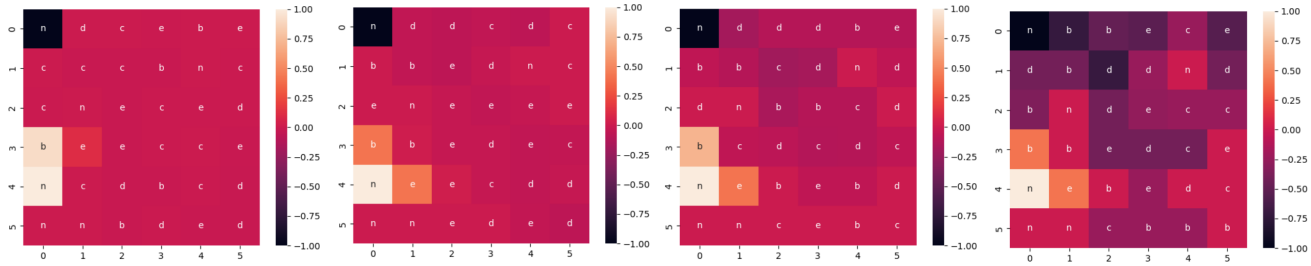
(i) fator = 0.2 (j) fator = 0.4 (k) fator = 0.6 (l) fator = 0.9

2.3.2 Análise

Com o crescimento do fator de desconto, as casas próximas ao **GOAL** tiveram um crescimento menor do valor de Q mas, a característica principal da variação deste fator, é o maior decrescimento dos valores Q distantes do **GOAL**, refletindo a maior perda de valor com o tempo desses estados. De forma semelhante, o aumento do fator de desconto também refletiu na diminuição da recompensa média nos testes feitos.

2.4 Variando a Recompensa

2.4.1 Resultados



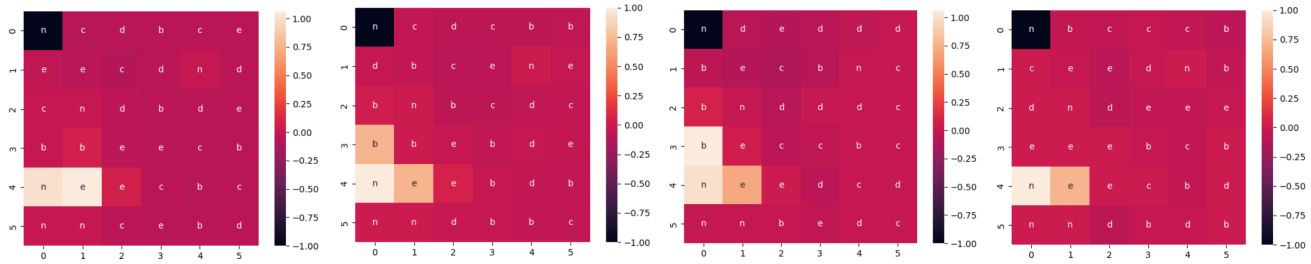
(m) r = -0.02 (n) r = -0.09 (o) r = -0.2 (p) r = -0.8

2.4.2 Análise

Progressivamente reduzindo a recompensa padrão de um estado não terminal, observamos, claro, uma diminuição mais significativa da recompensa média do agente. Mas, acerca da sua exploração do Grid, estados mais distantes do **GOAL** foram progressivamente mais penalizados, ainda mais que aumentando o fator de desconto. É notório, pelas imagens, também que a medida que a recompensa diminui, os caminhos mais utilizados pelo agente até o **GOAL** ficam mais evidentes com menores valores no *heatmap*, ao passo que com uma recompensa não tão negativa, a exploração (valores Q) fica mais uniforme.

2.5 Variando o *epsilon-greedy*

2.5.1 Resultados



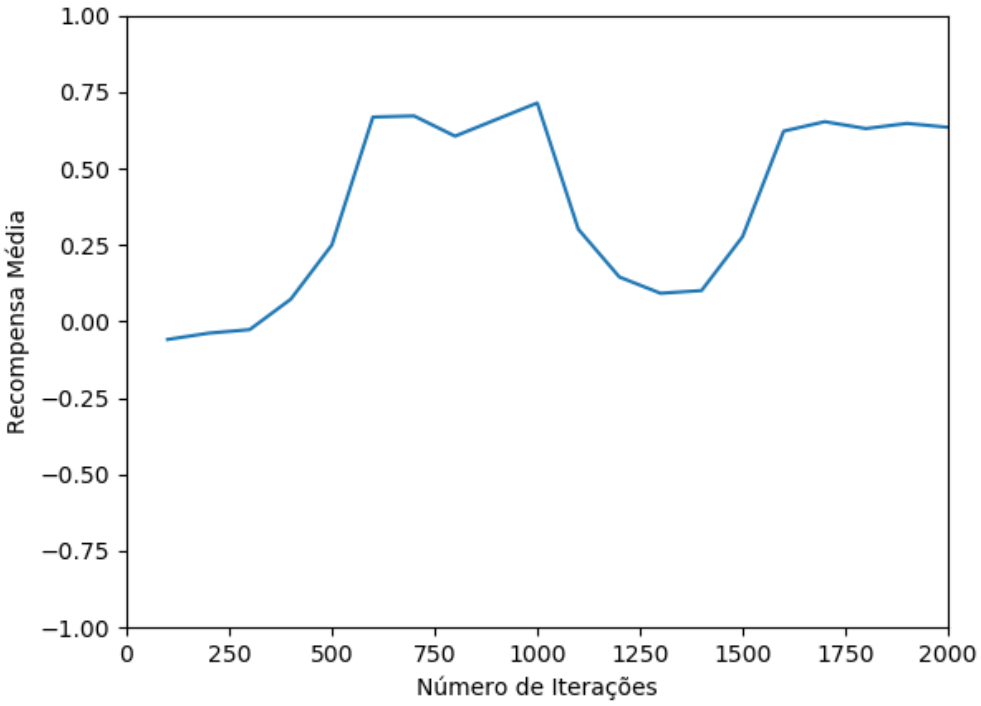
(q) Sem e-greedy (r) e = 0.2 (s) e = 0.8 (t) e = 0.9

2.5.2 Análise

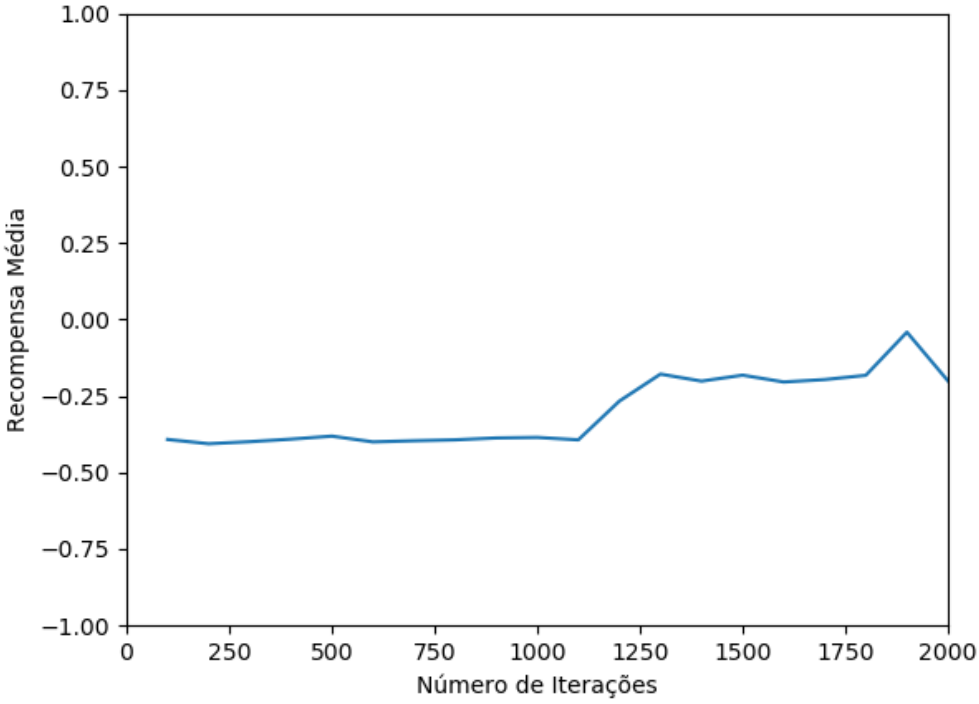
Sob análise das imagens e do GIF, um aumento progressivo do fator *epsilon-greedy* influenciou em um comportamento e histórico de movimentos do agente mais aleatório, com pouco feedback positivo nos estados mais próximos ao **GOAL**.

2.6 Gráficos de Número de Iterações vs Recompensa Média

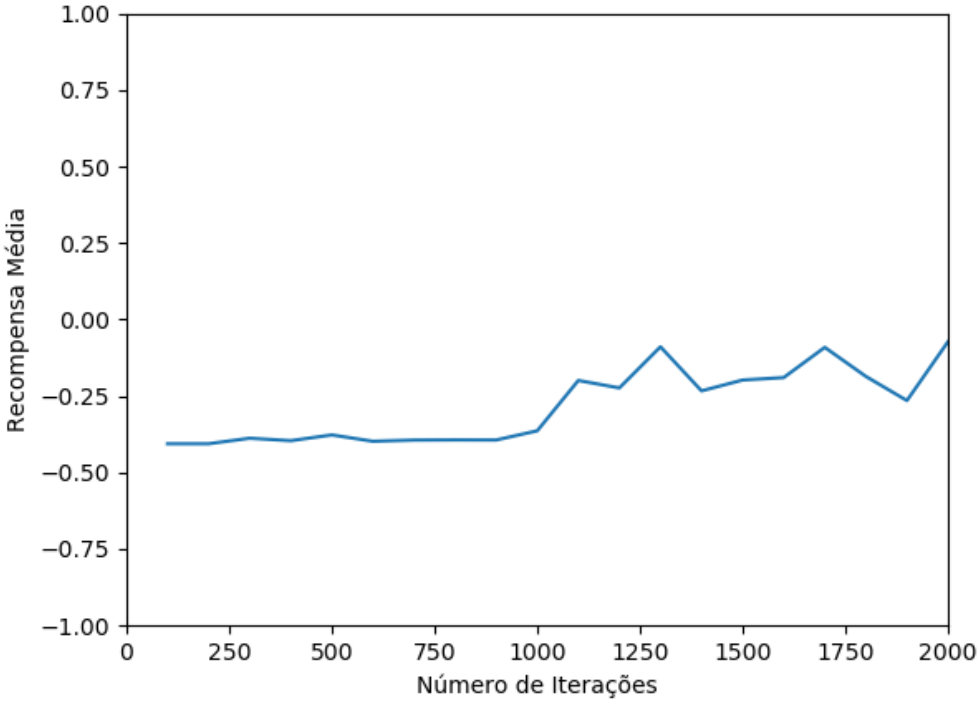
2.6.1 Grid do exemplo in1.txt (Matriz 5x5)



2.6.2 Grid do exemplo in2.txt (Matriz 7x7)



2.6.3 Grid do exemplo in3.txt (Matriz 6x6)



2.6.4 Análise dos Gráficos

Para todos os Grids considerados, testando múltiplas vezes, os valores de recompensa médios crescem gradativamente (sem uma proporção bem definida) a medida que o número de iterações aumenta. No entanto, em termos absolutos e relativos, a recompensa média para a matriz de menor dimensionalidade (in1.txt), apresenta maiores valores de recompensa média, o que pode ser explicado pelo fato de que o agente terá que caminhar menos estados em média para atingir o **GOAL**, de modo que os valores Q não sofrerão diminuições significativas devido às recompensas padrão e o fator de desconto.