Inteligência Artificial - Trabalho Prático II
Aprendizado por Reforço: *Q-Learning*

Hugo Araújo de Sousa
(2013007463)

Departamento de Ciência da Computação
Instituto de Ciências Exatas
Universidade Federal de Minas Gerais

hugosousa@dcc.ufmg.br

1. Introdução

Em Inteligência Artificial, existem vários tipos de agentes, cada um com características adequadas para as mais diversas situações. Agentes de aprendizado, por exemplo, procuram aprender, durante sua interação com o ambiente em que estão inseridos, como tomar decisões melhores, isto é, que levam ou esperam levar a melhores resultados. Um agente de aprendizado supervisionado precisa de informações sobre quais ações são as corretas em cada estado. Entretanto, a disponibilidade desse tipo de informação é rara. Dessa forma, uma alternativa é o chamado **agente de aprendizado por reforço**, que executa ações em um ambiente e observa as recompensas ou punições que recebe por tais ações. Assim, esse tipo de agente consegue saber o que é bom ou ruim para cada situação [Russell and Norvig 2016]. A Figura 1 ilustra esse processo.

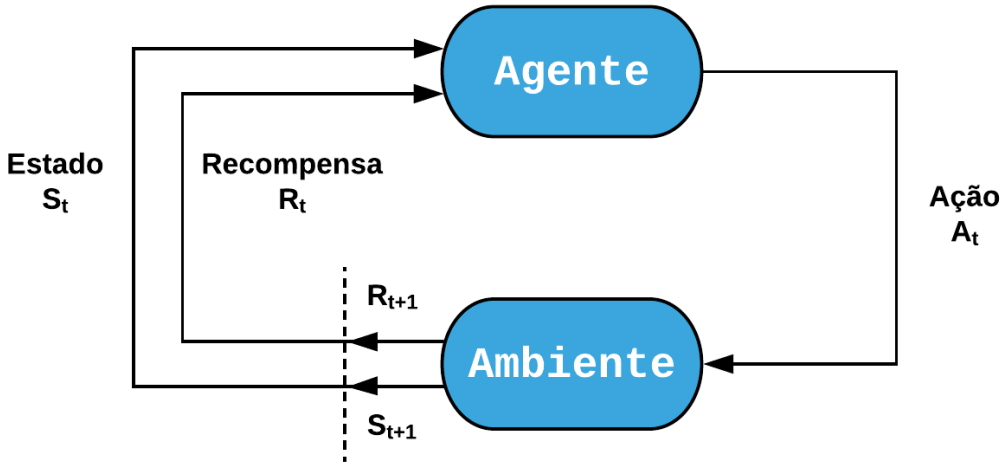

Figura 1. Funcionamento de um agente simples de aprendizado por reforço. Adaptado de [Sutton et al. 1998].

Um tipo específico de agente de aprendizado por reforço é o agente de ***Q-Learning***, que utiliza uma função de ação-utilidade para calcular a utilidade esperada de tomar uma determinada ação em um estado. O conceito de utilidade refere-se aos ganhos preferidos por um agente.

Neste trabalho, um agente de *Q-Learning* será implementado para resolver o problema de busca em mapas 2D. Mais especificamente, o agente implementado será uma versão simplificada do jogador de Pac-Man [DeNero and Klein 2010], procurando por pastilhas no mapa e evitando encontrar fantasmas, que não se movem.

2. Implementação

Este trabalho foi implementado com a linguagem Python 3. Foram criadas duas classes, uma para definir a estrutura dos mapas utilizados pelo algoritmo principal e outra para implementar o algoritmo propriamente dito, assim como a estrutura geral de um problema de decisão de Markov. As classes e arquivos de código do trabalho são descritas a seguir.

2.1. Classe `Map`

A classe `Map` representa mapas 2D formados por células. Essas células são representadas por caracteres, que definem seu estado. Uma posição (célula) em uma instância da classe `Map` pode ser dos seguintes tipos:

- **Livre**: o agente pode transitar livremente por essa posição. É representada por um caractere **"-"**.
- **Parede**: representa um bloqueio intransponível para o agente. É representada por um caractere **"#"**.
- **Pastilha:** o objetivo do agente no jogo é chegar a uma posição com uma pastilha. É representada por um caractere **"0"**.
- **Fantasma:** na versão simplificada do jogo Pac-Man que este trabalho tem como referência, o agente deve desviar das posições que possuem um fantasma. Os fantasmas são fixos em suas posições, isto é, não se movem. Essas posições são representadas pelo caractere **"&"**.

Para inicializar uma instância da classe `Map`, o nome de um arquivo com a representação do mapa deve ser passado como parâmetro. Esse arquivo deve conter, em sua primeira linha, o número de linhas e colunas do mapa, separados por um espaço. Em seguida, deve conter cada posição do mapa, representada por seu caractere correspondente. Os mapas dessa classe utilizam a notação matricial de programas para definir suas coordenadas, logo, a origem (posição (0, 0)), é o caractere superior esquerdo.

2.2. Classe `MDP`

Os processos de decisão de Markov a serem resolvidos com o algoritmo *Q-Learning* foram implementados através da classe `MDP`. Os atributos essenciais do processo de decisão de Markov deste trabalho são descritos a seguir:

- **S**: conjunto dos possíveis estados em que o agente pode se encontrar, como definidos na descrição da classe `Map`.
- **A**: conjunto das ações que o agente pode executar para se mover no mapa. São elas:
  - **Acima**: representada pelo caractere **"^"**.
  - **Abaixo**: representada pelo caractere **"v"**.
  - **Direita**: representada pelo caractere **">"**.
  - **Esquerda**: representada pelo caractere **"<"**.
- **R**: recompensas que o agente recebe para cada estado. São definidas pelo tipo da posição no mapa, da seguinte forma:
  - Em posições livres, a recompensa é -1.
  - Em posições com fantasma, a recompensa é -10.
  - Em posições com pastilha, a recompensa é 10.
- **Alfa** (α): taxa de aprendizado para o algoritmo *Q-Learning*.

- **Gama** (γ): fator de desconto para as recompensas no algoritmo *Q-Learning*.

Esses dados são utilizados pela classe `MDP` para executar o algoritmo *Q-Learning* no mapa escolhido, obtendo assim valores cada vez melhores para a função de valor Q, o que leva o agente a se aproximar da política ótima de ações. O algoritmo *Q-Learning* implementado neste trabalho tem como base as orientações de [Mnemosyne Studio 2012]. O pseudocódigo é mostrado a seguir.

Algoritmo 1 Algoritmo para agente *Q-Learning*

procedure Q-LEARNING(Mapa, Iterações)
  $Q \leftarrow$ INICIALIZAMATRIZQ$(Mapa)$
  $i \leftarrow 0$
  while $True$ **do** ▷ Executa até que o número máximo de iterações seja atingido
    $Estado \leftarrow$ SELECIONAESTADOINICIAL$(Mapa)$
    while $Estado \neq Terminal$ **do**
      $Acao, NovoEstado \leftarrow$ SELECIONAAÇÃO$(Q, Estado)$
      $MaxQ \leftarrow$ OBTEMMAIORQ$(Q, NovoEstado)$
      $Q \leftarrow$ ATUALIZAQ$(Q, Acao, Estado, MaxQ)$
      $Estado \leftarrow NovoEstado$
      $i \leftarrow i + 1$
      if $i = Iteracoes$ **then**
        break
      end if
    end while
  end while
end procedure

O método `InicializaMatrizQ` (implementado no trabalho no método `init_qmatrix`) simplesmente inicializa a tabela de valores Q utilizada pelo algoritmo. A escolha foi deixar esse passo o mais simples possível, dessa forma, os valores são inicializados com 0. Em seguida, o algoritmo executa um *loop* para o número escolhido de iterações. O primeiro passo é selecionar um estado inicial para cada episódio (um episódio termina quando o agente visita um estado terminal). Esse procedimento de escolha - `SelecionaEstadoInicial` - é implementado no trabalho no método `select_initial_state`, que simplesmente procura por uma coordenada aleatória cuja posição seja livre no mapa.

Uma vez que o estado inicial do agente é escolhido, o algoritmo entra em um outro *loop*, até que encontre um estado terminal. Nesse *loop*, o primeiro passo é escolher uma ação e verificar o novo estado a ser atingido após executá-la. O método que implementa esse processo - `SelecionaAção` - no trabalho é o `select_action`. Esse método seleciona uma ação aleatória entre as disponíveis e a simula, obtendo assim o próximo estado. O maior valor da matriz Q para o próximo estado é anotado e utilizado para atualizar o valor de Q do estado atual. Essa atualização segue a fórmula padrão do algoritmo [Russell and Norvig 2016]:

$$Q(s,a) = Q(s,a) + \alpha(R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a))$$

Onde **s** é o estado atual e **a** é a ação escolhida. É importante ressaltar que, caso a

ação escolhida para o agente o leve para fora do mapa ou para uma posição bloqueada, ele continuará na mesma posição em que se encontra.

***Exploration vs. Exploitation*:** A fim de balancear o algoritmo em termos de *eploration* - busca por diversidade de soluções - e *exploitation* - busca por melhorias na melhor solução encontrada em determinado momento -, também foi implementada a política ϵ-greedy [Whiteson et al. 2007]. Essa política afeta a forma como uma nova ação é selecionada durante o procedimento `SelecionaAção` mostrado no Algoritmo 1. Um parâmetro $0 \leq \epsilon \leq 1$ é escolhido de forma que uma ação é escolhida com probabilidade ϵ, enquanto a melhor ação possível é escolhida com probabilidade $(1 - \epsilon)$. Dessa forma, com um valor ϵ alto, estamos aumentando a *exploration*, pois a maioria das ações escolhidas será aleatória, e com ϵ baixo, estamos aumentando a *exploitation*, pois a maioria das ações escolhidas serão as melhores possíveis. O conceito de melhor ação refere-se à ação cujo valor de Q é o maior.

2.3. Programa principal

O programa principal (`tp2.py`) recebe argumentos de linha de comando, inicializa o módulo de geração de números pseudoaleatórios, cria uma instância da classe `MDP` com os dados passados como argumentos e inicia o algoritmo *Q-Learning*. A execução na linha de comando deve ser feita da seguinte forma:

```
./tp2.py map alpha gamma iter [-e EPSILON] [-s SEED] [-q
                               QSUMF]
```

Onde **map** define o nome do arquivo que contém as informações do mapa em que o agente será inserido, **alpha** é a taxa de aprendizado, **gamma** é o fator de desconto e **iter** é o número de iterações que o algoritmo *Q-Learning* deve executar. Além disso, o programa também aceita três parâmetros opcionais: **EPSILON**, definido pela *flag* **-e**, que especifica o parâmetro ϵ para a política ϵ-greedy, **SEED**, definido pela *flag* **-s**, que especifica a semente para as escolhas aleatórias realizadas durante a execução do algoritmo e **QSUMF**, definido pela *flag* **-q**, que especifica o nome do arquivo onde dados de teste devem ser salvos. Mais detalhes sobre esses dados serão dados na Seção 3.

Além da chamada mostrada acima, também foi criado um *script*, `qlearning.sh`, que abstrai alguns detalhes de parâmetros e utiliza somente os obrigatórios (mapa, taxa de aprendizado, fator de desconto e número de iterações). Esse *script* pode ser executado da seguinte forma:

```
./qlearning.sh map alpha gamma iter
```

3. Metodologia de Testes

Considerando a quantidade de parâmetros disponíveis para execução do algoritmo implementado, foi necessário realizar vários testes para verificar como esses parâmetros afetam o resultado obtido. O primeiro passo para definir a estrutura dos testes foi estabelecer uma métrica que pudesse ser usada como referência para convergência de valores e aproximação da política ótima para cada mapa. Dessa forma, a métrica escolhida foi a **soma dos maiores valores de cada posição na tabela Q**, isto é:

$$m = \sum_{i,j} \max_{a} Q_{i,j,a}$$

Onde **i** representa o número de linhas na grade do mapa e **j** representa o número de colunas. Dessa forma, a ideia é que uma vez que esse valor convirja, o algoritmo encontrou uma política ótima e não está mais realizando alterações nos valores correspondentes na tabela Q. Como explicado na Seção 2.3, para ativar a coleta desses dados, é necessário utilizar a *flag* **-q**, especificando o nome do arquivo onde os valores de **m** devem ser salvos para cada iteração executada.

Uma vez que essa métrica foi definida, também foi necessário definir o número de repetições para cada execução do algoritmo, já que ele utiliza escolhas aleatórias, é necessário repetir as medições e obter a média. Assim, a escolha para os experimentos foi a de variar o valor da semente do módulo de aleatoriedade 5 vezes para cada medição. Com isso, o roteiro de testes foi:

1. **Determinar a política ótima para cada mapa e iterações necessárias para convergência**: utilizando a métrica **m** definida acima e fixando os parâmetros restantes, o primeiro experimento visa obter o número de iterações necessárias para convergência dos resultados para cada um dos mapas. Além disso, também podemos obter a política ótima para cada um. Os parâmetros utilizados nesse experimento foram:
    - $\alpha = 0.3$.
    - $\gamma = 0.9$.
    - $\epsilon = 1$.
    - Iterações $= 1000000$.
2. **Verificar o efeito do fator de desconto**: com o número de iterações fixado para cada mapa através do Experimento 1, esse experimento visa verificar qual é o efeito do fator de desconto (γ) no valor da métrica **m**. Dessa forma, os seguintes valores foram utilizados:
    - $\alpha = 0.3$.
    - $\gamma \in \{0.2, 0.5, 0.9\}$.
    - $\epsilon = 1$.
    - Iterações determinadas pelo Experimento 1.
3. **Determinar a melhor taxa de aprendizado:** com o fator de desconto fixado pelo Experimento 2, o próximo passo é determinar como a taxa de aprendizado altera os resultados obtidos e qual é a melhor a ser utilizada. Parâmetros:
    - $\alpha \in \{0.2, 0.5, 0.8, 1.0\}$.
    - γ determinado pelo Experimento 2.
    - $\epsilon = 1$.
    - Iterações determinadas pelo Experimento 1.
4. **Verificar o *trade-off* entre *exploration* e *exploitation***: por fim, vamos verificar o efeito do parâmetro ϵ no algoritmo, utilizando a política ϵ-greedy. Assim, temos os seguintes parâmetros para esse último experimento:
    - α determinado pelo Experimento 3.
    - γ determinado pelo Experimento 2.
    - $\epsilon \in \{0.5, 0.8, 1.0\}$.
    - Iterações determinadas pelo Experimento 1.

4. Resultados

Com a metodologia de testes definida na Seção 3, os testes puderam ser executados. Para aumentar a velocidade deles, o utilitário gnu-parallel [Tange 2011] foi utilizado. Além disso, para criação dos gráficos, foi utilizada a biblioteca matplotlib [Hunter 2007]. Os resultados são apresentados a seguir.

4.1. Experimento 1: Convergência

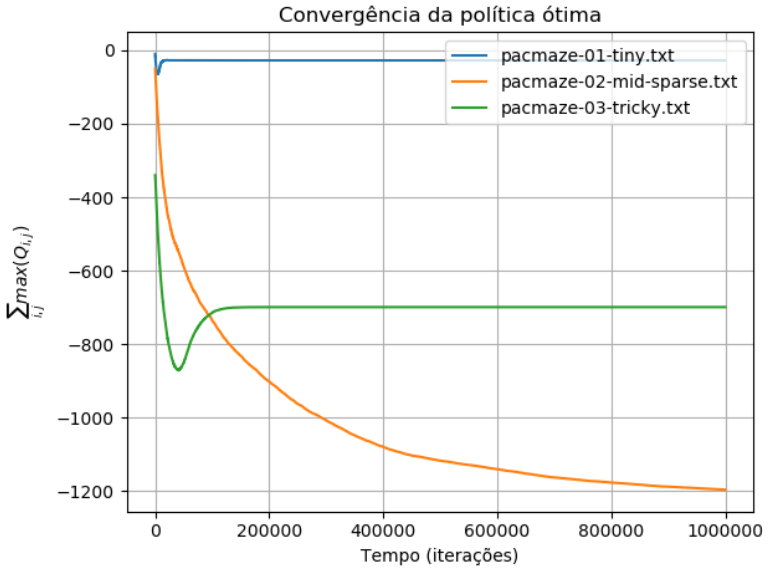

Figura 2. Experimento 1 - convergência da métrica m para os diferentes mapas.

pacmaze-01-tiny

```
###############
#>>>>>0<<<<<<#
#######^##^#^#
#>>>>>>^&>^#^#
#^###########&#
#^<<<<<<<<<<<#
###############
```

pacmaze-02-mid-sparse

```
####################
#v<<<<<<<<<<<<<#>>v#
#v<<&v<&v<&v<<<#>>v#
#v<<<<<<<<<<<<<#^#v#
#v################^#v#
#>>>>>>>>>>>>>>>^#v#
####################v#
#v<<<<<<<&>>>v<<<<<#
#v&v<########v<<&v<#
#v<<<<<<<<<<<<<<<<<#
#v#################^#
#v#>>>>v#0#>>>>>>>^#
#>>>^#>>>^#>>>>>>>^#
####################
```

pacmaze-03-tricky

```
####################
#v<<<&>>>v<<<<&>>>v#
#v&&&&&&&v&&&&&&&&v#
#>>>>>>>v&v&v<<<<<<<#
#>>v&>>v&v&v<<&v<<<#
#>>>>>>>v&v&v<<<<<<<#
#>>>>^&>>v<<&^<<<<<#
#>>>>>>>^&v&^<<<<<<<#
#>>^&>>^&v&^<<&^<<<#
#>>>>>>>^&0&^<<<<<<<#
####################
```

Figura 3. Políticas ótimas encontradas para cada mapa. Nas posições originalmente livres nos mapas, essa figura mostra a ação ótima a ser tomada pelo agente.

Os resultados do Experimento 1 são mostrados na Figura 2. Nela, podemos ver a convergência de **m** para o mapa **pacmaze-01-tiny.txt** aconteceu muito rapidamente (por volta de 30 mil iterações), devido a seu tamanho. Para o mapa **pamaze-03-tricky.txt**, a convergência acontece por volta de 200 mil iterações. Já para o mapa **pacmaze-02-mid-sparse.txt**, mesmo por volta de 1 milhão de iterações, a convergência ainda não tinha ocorrido. Por isso, o algoritmo foi executado novamente para esse mapa utilizando o valor de 5 milhões de iterações. Nesse caso, a convergência ocorre por volta de 4 milhões de iterações. Assim, para os próximos experimentos, o número de iterações para cada mapa foi fixado como mostrado na Tabela 1.

Com esse experimento também foi possível obter a política ótima aproximada para cada mapa. Elas são mostradas na Figura 3.

Mapa	Iterações
pacmaze-01-tiny	50.000
pacmaze-02-mid-sparse	5.000.000
pacmaze-03-tricky	200.000

Tabela 1. Número de iterações necessárias para convergência para cada mapa.

4.2. Experimento 2: Fator de Desconto

Os resultados do Experimento 2 são mostrados na Figura 4. A primeira observação feita é que, mesmo com os valores de **m** convergindo para valores diferentes, relação entre os valores de Q para um mesmo estado e diferentes ações tende a permanecer, o que, por sua vez, tende a manter a qualidade da política encontrada. Além disso, vemos (principalmente nos mapas **pacmaze-01-tiny** e **pacmaze-03-tricky**, que o algoritmo parece explorar melhor o espaço de soluções antes da convergência com $\gamma = 0.9$, o que é um ponto positivo. Dessa forma, fixamos $\gamma = 0.9$ para os próximos experimentos.

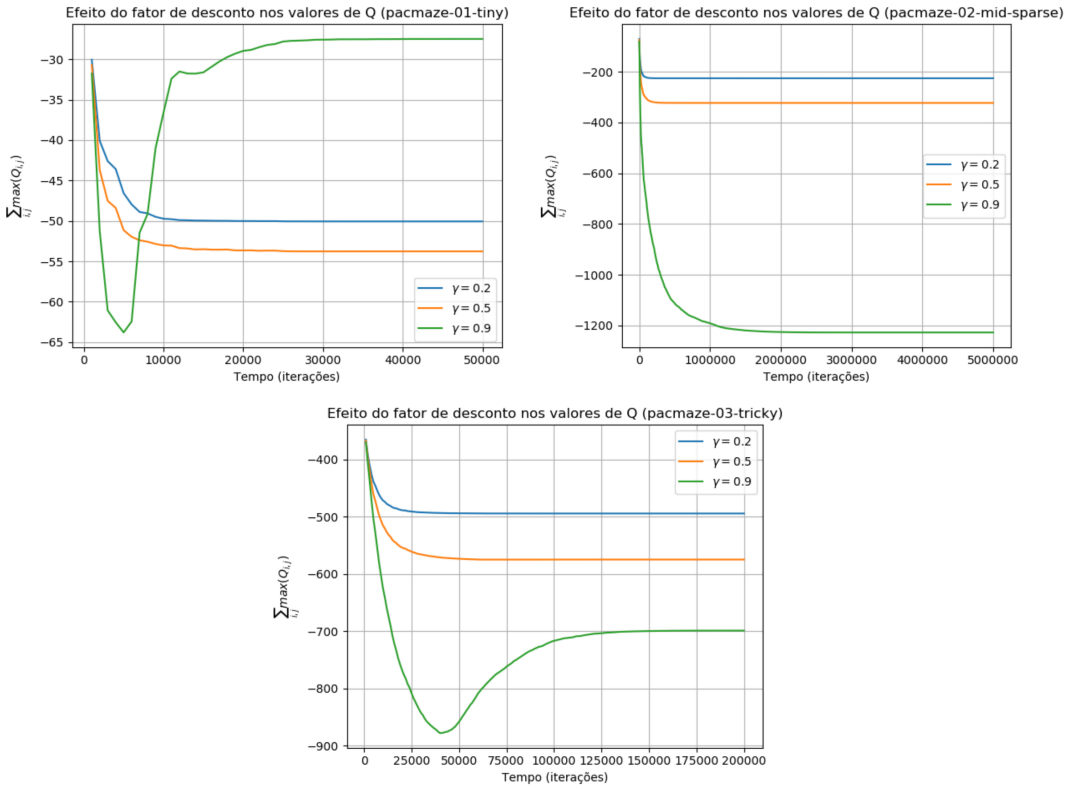

Figura 4. Efeito do fator de desconto na métrica m para os mapas analisados.

4.3. Experimento 3: Taxa de aprendizado

O próximo passo foi determinar como a taxa de aprendizado afeta os valores de Q encontrados pelo algoritmo. Os resultados obtidos com esse experimento são mostrados na Figura 5.

Em todos os gráficos, vemos que a taxa de aprendizado não altera o fato de que os valores de **m** eventualmente convergirão. Além disso, vemos que o principal efeito desse

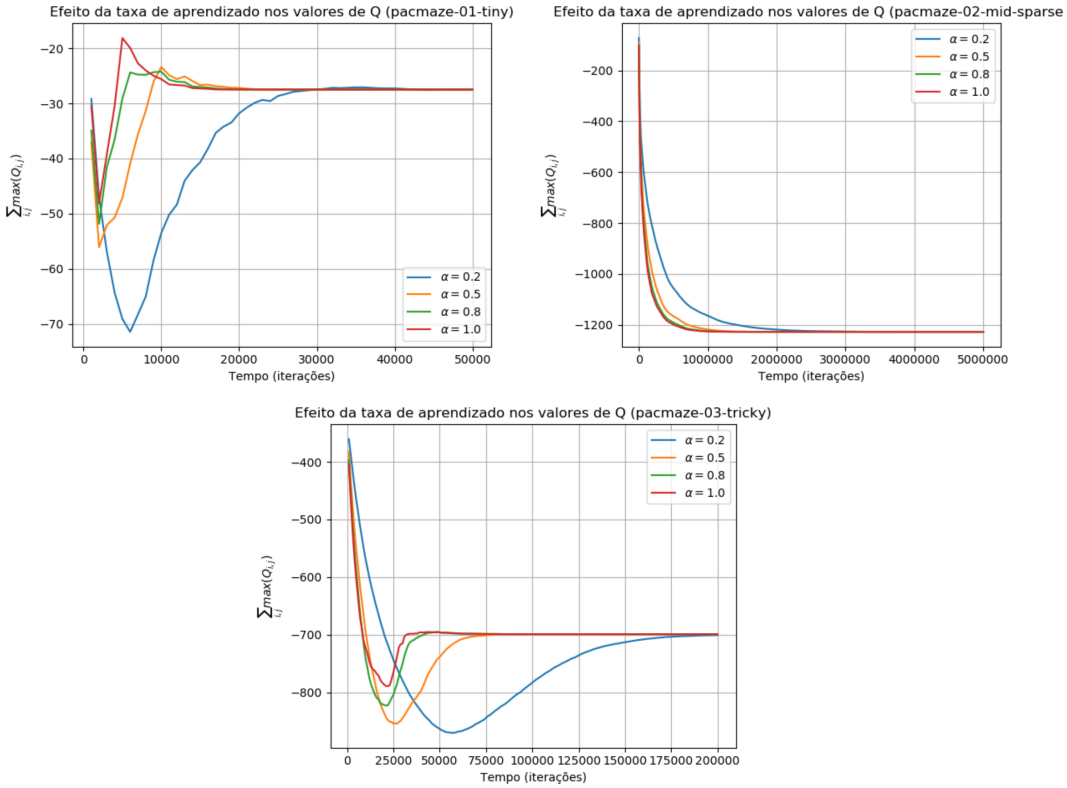

Figura 5. Efeito da taxa de aprendizado na métrica m para os mapas analisados.

parâmetro está relacionado à velocidade com que a convergência acontece, entretanto, como podemos ver nos gráficos para **pacmaze-01-tiny** e **pacmaze-03-tricky**, para todos os valores de α, o "desenho" do gráfico é similar. Dessa forma, de modo a balancear a velocidade de convergência e exploração de valores de Q, um bom valor para α e escolhido para o último experimento é $\alpha = 0.5$.

4.4. Experimento 4: ϵ-greedy

Por fim, observamos como a política ϵ-greedy se comporta para os mapas disponibilizados para o trabalho. A Figura 6 mostra os resultados desse experimento

É importante notar que para $\epsilon = 1$, os gráficos são iguais aos obtidos anteriormente, onde todos os experimentos utilizaram esse valor. Além disso, como esperado, para valores menores de ϵ, a convergência acontece mais rapidamente, uma vez que esses valores favorecem a escolha de ações com maiores valores de Q, durante a execução do algoritmo.

5. Considerações finais

Este trabalho apresentou a implementação do algoritmo *Q-Learning* para aprendizado por reforço. O algoritmo foi utilizado no contexto de um agente simples inserido em um ambiente 2D, similar ao jogo Pac-Man. Através do desenvolvimento deste trabalho, foi possível fixar vários conceitos vistos durante a disciplina Inteligência Artificial sobre agentes de aprendizado.

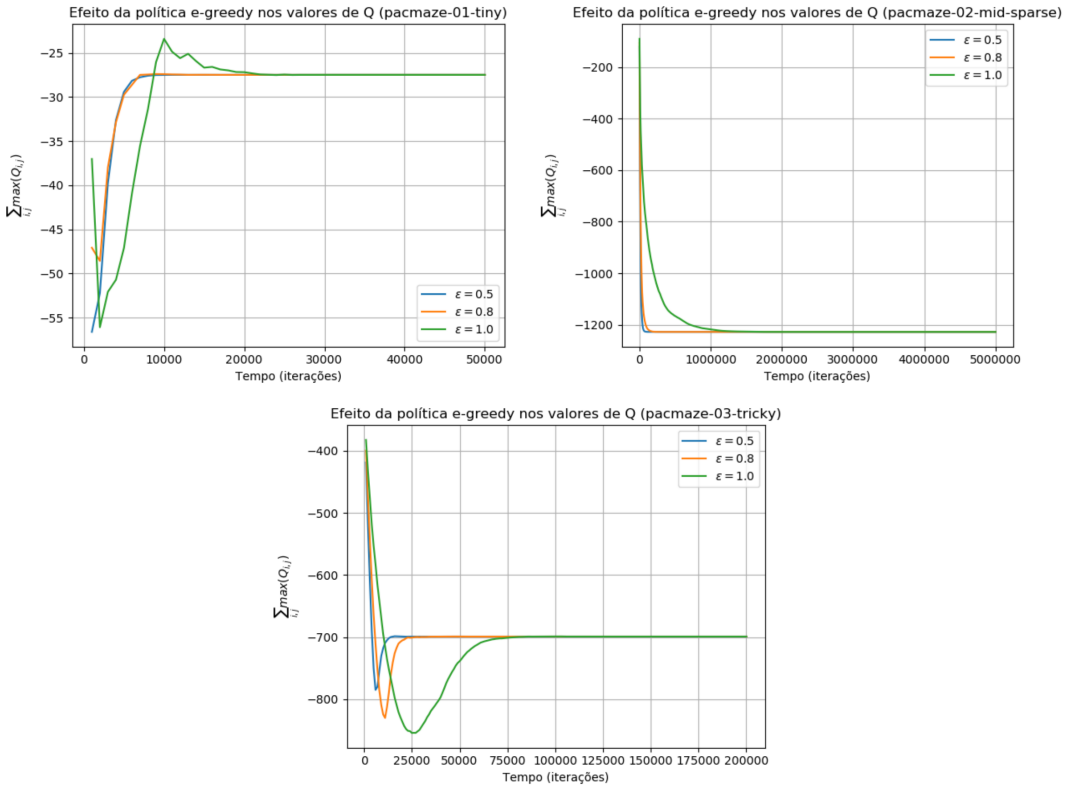

Figura 6. Efeito da política ϵ-greedy na métrica m para os mapas analisados.

De maneira geral, o desenvolvimento se deu sem maiores problemas, excepto pela dificuldade em encontra material sobre detalhes do algoritmo *Q-Learning*, principalmente sobre seu método de atualização da tabela Q. É importante ressaltar que teria sido útil ver aspectos sobre metodologia de experimentação durante as aulas da disciplina, uma vez que este trabalho pede maior cuidado com parâmetros estocásticos. Enquanto a implementação do código, em si, não demandou muito tempo, a execução dos testes se mostrou trabalhosa e demandante de bastante tempo.

Referências

DeNero, J. and Klein, D. (2010). Teaching introductory artificial intelligence with pac-man. In *Proceedings of the Symposium on Educational Advances in Artificial Intelligence*, pages 1–5.

Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95.

Mnemosyne Studio (2012). A Painless Q-Learning Tutorial. `http://mnemstudio.org/path-finding-q-learning-tutorial.htm`.

Russell, S. J. and Norvig, P. (2016). *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.

Sutton, R. S., Barto, A. G., et al. (1998). *Reinforcement learning: An introduction*. MIT press.

Tange, O. (2011). Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47.

Whiteson, S., Taylor, M. E., and Stone, P. (2007). Empirical studies in action selection with reinforcement learning. *Adaptive Behavior*, 15(1):33–50.