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

DCC642 - Introdução à Inteligência Artificial 2020/2
Professor: Luiz Chaimowicz
Monitor: Ronaldo Vieira

Trabalho Prático 2: MDPs e Aprendizado por Reforço

Valor: 20 pontos
Data de entrega: 26 de Março de 2021

Detalhes do problema

O objetivo deste trabalho é praticar os conceitos de processo de decisão de Markov (em inglês, *Markov decision process*, ou MDP) e os diversos algoritmos que os resolvem, inclusive os de aprendizado por reforço. Para tal, disponibilizamos no Moodle da disciplina o código-fonte de uma plataforma de experimentação de processos de decisão de Markov em diversos ambientes (inclusive no jogo Pac-Man). Será necessário conhecimento básico de Python 2[1] e uso de terminal ou prompt de comando. Além disso, sua instalação do Python 2 deve possuir a biblioteca Tkinter.[2]

Neste trabalho, você implementará os algoritmos **value iteration** e **Q-learning** (e o Q-learning aproximado, caso queira pontos extra). Você irá usar estes algoritmos primeiramente no Grid World (como visto na aula 9) e depois para jogar o jogo Pac-Man (dessa vez com fantasmas!). Tudo isso foi separado em passos e, a seguir, cada um destes passos será explicado em mais detalhes. Não se assuste com o tamanho deste documento – apenas os passos 2, 5, 6 e 9 requerem implementação. **Leia com atenção. :)**

Passo 1: Familiarizar-se com o código-fonte

Vá ao Moodle da disciplina, e faça o download do arquivo .zip "Código-fonte Inicial", na seção "Trabalho Prático 2: MDPs e Aprendizado por Reforço'. Em seguida, extraia seu conteúdo em uma pasta de sua preferência. Dentre os arquivos extraídos, haverá diversos arquivos Python (.py) e duas pastas. Não é necessário entender o conteúdo de todos os arquivos (mas se quiser pode também). Os arquivos que importam são:

- O arquivo `valueIterationAgents.py`, onde você fará a implementação do algoritmo value iteration. **Este é um dos arquivos que você deve alterar.**
- O arquivo `qlearningAgents.py`, onde você fará a implementação do algoritmo Q-learning (e Q-learning aproximado). **Este é um dos arquivos que você deve alterar.**
- O arquivo `analysis.py`, onde você resolverá alguns dos passos deste trabalho sugerindo parâmetros para os algoritmos. **Este é um dos arquivos que você deve alterar.**
- O arquivo `util.py`, que contém a estrutura de dados `Counter`, que funciona como um dicionário (`dict`) cujo valor padrão é zero. Em outras palavras, ao acessar uma chave não existente, o `Counter` retornará zero enquanto um dicionário padrão lançaria uma exceção `KeyError`. O uso desta estrutura durante as implementações não é obrigatório, mas pode ser conveniente.
- O arquivo `mdp.py`, que contém a especificação da classe `MarkovDecisionProcess`, que será usada em alguns passos deste trabalho.
- O script `gridworld.py`, usado para executar o ambiente Grid World.

[1] Um guia básico da linguagem está disponível em `http://df.python.org.br/pycubator/index.html`.
[2] Instruções de instalação estão disponíveis em `https://riptutorial.com/tkinter/example/3206/installation-or-setup`

- O script `pacman.py`, usado para executar partidas do Pac-Man.
- O script `autograder.py`, usado para verificar a corretude da implementação dos algoritmos, assim como sua pontuação nos passos 2 a 9 deste trabalho. Use `python2 autograder.py` para verificar todos estes passos ao mesmo tempo ou `python2 autograder --question passoX` para verificar o passo `X`, onde `X` $\in \{2, 3, 4, 5, 6, 7, 8, 9\}$. Ele fará alguns testes e, caso todos resultem em `PASS`, sua implementação/resposta está correta e você completou o(s) passo(s) em questão. Se algum resultar em `FAIL`, há algo errado e ele mostrará o que é. Ele também mostrará quantos pontos você ganhou no(s) passo(s).

Para controlar um agente manualmente no Grid World, use `python2 gridworld.py --manual`. Para escolher outro agente, use o parâmetro `--agent nome_do_agente`. Os agentes disponíveis são: `random` (agente padrão), `value` e `q`.[3] A fase padrão usada pelo script é a mesma utilizada nas aulas. Para especificar outra fase, use o parâmetro `--grid nome_da_fase`. As fases disponíveis são: `BookGrid` (fase padrão), `BridgeGrid`, `CliffGrid` e `MazeGrid`. Caso você esteja em um computador sem ambiente gráfico disponível (e.g. acessando um computador por SSH), use o parâmetro `--text` para que o Grid World seja renderizado via texto. Por último, para ver todos os parâmetros disponíveis no script, use o parâmetro `--help`.

Para controlar o Pac-Man manualmente, use `python2 pacman.py`. Para trocar a fase, use `python2 pacman.py --layout fase`, onde `fase` é o nome de um arquivo na pasta `layouts`. Por fim, para ver todos os parâmetros disponíveis no script, use `python2 pacman.py --help`.

Para completar o **Passo 1**, você deverá entender como executar episódios do Grid World e partidas do Pac-Man. Todos os comandos exemplificados acima devem executar sem erros, apresentando o resultado esperado. Além disso, você deverá ler o arquivo Python `mdp.py` e entender como interagir com os processos de decisão de Markov implementados neste código-fonte. Apesar de não valer pontos, este passo é essencial para que você consiga completar os passos a seguir. Em caso de dúvidas, não hesite em usar o fórum "Dúvidas / Discussão" do Moodle da disciplina ou contatar o monitor.

Passo 2: (6 pontos) Implementar o value iteration

No arquivo `valueIterationAgents.py`, você encontrará a classe `ValueIterationAgent`. Ela especifica um agente que age de acordo com o algoritmo value iteration, e é seu trabalho terminar de implementá-la. Para isso, você completará a implementação de três métodos:

- O método construtor da classe (`__init__`), que recebe um processo de decisão de Markov (parâmetro `mdp`, da classe `MarkovDecisionProcess`), um valor γ (parâmetro `discount`) e o número de iterações a serem realizadas (parâmetro `iterations`). Imediatamente após o código já existente no construtor, você deverá realizar o número pedido de iterações do algoritmo value iteration, guardando a função de valor $V(s)$ resultante em algum atributo da classe. (Para isso, o dicionário `self.values` já vem inicializado para você, mas se preferir usar outra estrutura, sinta-se livre. Apenas lembre-se de usar o prefixo `self.`, para que a estrutura se torne de fato um atributo da classe e não apenas uma variável dentro do construtor.)
- O método `computeQValueFromValues`, que recebe um estado (parâmetro `state`) e uma ação (parâmetro `action`) e deve retornar o Q-valor daquela ação naquele estado. Em outras palavras, o método deve retornar $Q(s, a) = \sum_{s'} T(s'|s, a)(r(s, a, s') + \gamma V(s'))$, onde T é a função de transição, $r(s, a, s')$ é a recompensa obtida no estado s' (vindo do estado s após a ação a) e $V(s')$ é a função de valor resultante do value iteration para o estado s'. Dica: quais métodos do `self.mdp` representam T e r?
- O método `computeActionFromValues`, que recebe um estado (parâmetro `state`) e deve retornar a melhor ação a ser feita naquele estado, de acordo com os Q-valores das ações disponíveis naquele

[3] Os agentes `value` e `q` usam, respectivamente, os algoritmos value iteration e Q-learning. Portanto, eles só funcionarão corretamente após sua implementação. ;)

estado. Em outras palavras, o método deve retornar $\pi(s) = \arg\max_a Q(s,a)$. Dica: use os métodos `self.mdp.getPossibleActions` e `self.getQValue`. Empates entre Q-valores de ações podem ser resolvidos da maneira que você achar melhor.

Para testar se sua implementação está correta, utilize o *autograder*:

```
python2 autograder.py --question passo2
```

Apesar da função de valor poder ser inicializada com quaisquer valores, é importante que você inicialize sua função de valor com zeros, para que o *autograder* funcione corretamente. Para completar o **Passo 2**, você deverá implementar os métodos descritos acima e obter pontuação máxima (6/6) no *autograder*. Para ver seu agente de value iteration funcionando na prática, use:

```
python2 gridworld.py --agent value --iterations 100 --episodes 10
```

Este comando executará 100 iterações do value iteration na fase `BookGrid` vista na aula 9 para obter uma função de valor $V(s)$ (e, por consequência, Q-valores $Q(s,a)$ e uma política π), e em seguida executará 10 episódios onde o agente seguirá a política encontrada.

Passo 3: (1 ponto) Atravessar a ponte

Neste passo, você usará a fase `BridgeGrid` do Grid World. Como mostra a Figura 1, esta fase possui dois estados terminais positivos cujas recompensas são, respectivamente, 1 e 10. Estes dois estados são separados por uma "ponte" estreita, de forma que "cair" em quaisquer dos estados terminais paralelos resulta em uma recompensa de -100. O estado inicial deste processo de decisão de Markov é o estado adjacente ao estado terminal de recompensa 1.

Figura 1: Exemplo de política e função de valor para a fase `BridgeGrid`.

Com os parâmetros padrão ($\gamma = 0.9$ e 20% de chance de "errar" o movimento), a política ótima encontrada pelo value iteration não consegue atravessar a ponte, isto é, ela não tenta chegar ao estado terminal de recompensa 10. Para completar o **Passo 3**, você deve mudar apenas **um** destes parâmetros de forma que a política ótima encontrada pelo value iteration tente atravessar a ponte ($\pi(s) = \text{direita}, \forall s$ não-terminal). Para tal, modifique as variáveis `answerDiscount` e `answerNoise` no método `passo3` do arquivo `analysis.py` e corrija sua resposta com o *autograder*:

```
python2 autograder.py --question passo3
```

Para ver o funcionamento do agente na prática, com quaisquer parâmetros, use:

```
python2 gridworld.py --agent value --iterations 100 --grid BridgeGrid
                     --discount X --noise Y
```

Onde `X` é o parâmetro γ (por padrão é 0.9) e `Y` é o parâmetro de ruído (chance de errar o movimento; por padrão é 0.2).

Passo 4: (2 pontos) Chegar ao pico da montanha

Neste passo, você usará a fase `DiscountGrid`. Como mostra a Figura 2, esta fase possui dois estados terminais positivos cujas recompensas são, respectivamente, 1 e 10. Considere esta fase como uma trilha em uma montanha, onde há uma pico menor com uma vista legal (de recompensa 1) e outro pico, mais longe, com uma vista *muito* legal (de recompensa 10). Há duas formas de chegar a estes dois picos: por um caminho longo porém seguro ou por um caminho curto porém ao lado de um penhasco (setas verde e vermelha, respectivamente, na Figura 2). Acidentalmente cair no penhasco resulta em uma recompensa de -10 (sai barato, até).

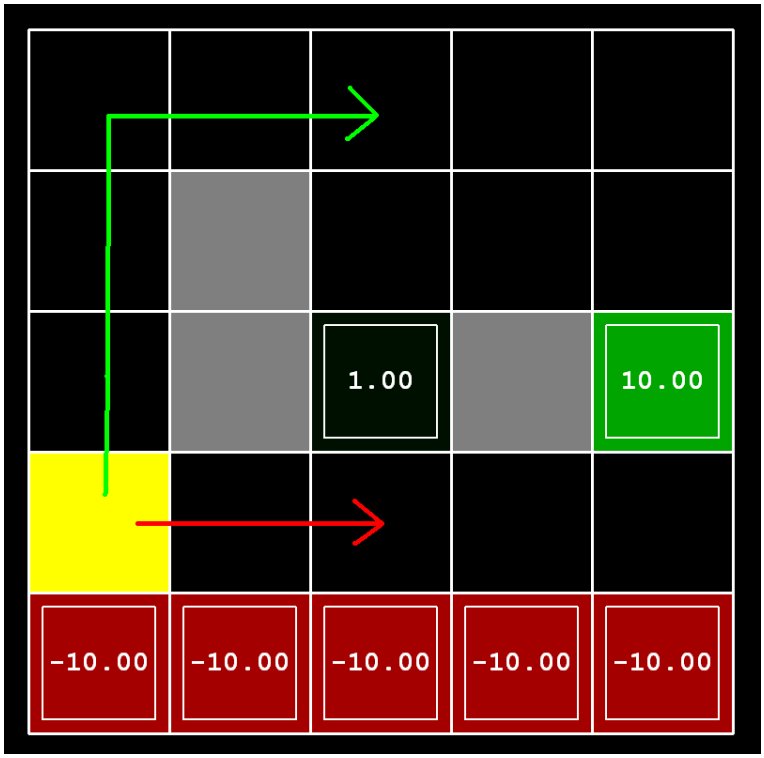

Figura 2: Ilustração da fase `DiscountGrid`.

Para completar o **Passo 4**, você deverá sugerir combinações de parâmetros (em específico, o γ, o ruído e a recompensa em estados não-terminais) de forma que a política ótima obtida pelo value iteration reproduza diferentes comportamentos na trilha. Estes comportamentos são:

a. Preferir o pico com vista legal (recompensa 1), indo pelo caminho ao lado do penhasco.

b. Preferir o pico com vista legal (recompensa 1), indo pelo caminho seguro.

c. Preferir o pico com vista *muito* legal (recompensa 10), indo pelo caminho ao lado do penhasco.

d. Preferir o pico com vista *muito* legal (recompensa 10), indo pelo caminho seguro.

e. Abandonar tudo e viver para sempre na trilha (nunca chegar aos picos nem ao penhasco).

Para sugerir os parâmetros, modifique as variáveis `answerDiscount`, `answerNoise` e `answerLivingReward` nos métodos `passo4X` do arquivo `analysis.py`, onde `X` é a letra referente ao comportamento pedido (`X` $\in \{$`a`, `b`, `c`, `d`, `e`$\}$). Se algum dos comportamentos for impossível, retorne a string `'NOT POSSIBLE'` no método adequado. Corrija sua resposta com o *autograder*:

```
python2 autograder.py --question passo4
```

Para ver o funcionamento do agente na prática, com quaisquer parâmetros, use:

```
python2 gridworld.py --agent value --iterations 100 --grid DiscountGrid
                --discount X --noise Y --livingReward Z
```

Onde `X` é o parâmetro γ, `Y` é o parâmetro de ruído (chance de errar o movimento) e `Z` é a recompensa obtida ao chegar em algum estado não-terminal.

Passo 5: (6 pontos) Implementar o Q-learning

Seu agente de value iteration usa todos os atributos do processo de decisão de Markov para construir uma política ótima. No entanto, em muitos problemas de decisão da vida real, não temos conhecimento da função de transição ou das recompensas. Nesse caso, uma forma possível de buscar uma política ótima é através de tentativa e erro. Neste passo, você implementará o Q-learning: um algoritmo que aprende políticas através da experiência, interagindo com o ambiente.

No arquivo `qlearningAgents.py`, você encontrará a classe `QLearningAgent`. Ela especifica um agente que age de acordo com o algoritmo Q-learning, e é seu trabalho terminar de implementá-la. Para isso, você completará a implementação de quatro métodos:

- O método construtor da classe (`__init__`). Imediatamente após o código já existente no construtor, você deverá inicializar a estrutura de dados em que os Q-valores serão guardados. Assim como no passo 2, o dicionário `self.qValues` já vem inicializado para você, mas se preferir outra estrutura, sinta-se livre para alterar.

- o método `getQValue`, que recebe um estado (parâmetro `state`) e uma ação (parâmetro `action`) e deve retornar o Q-valor $Q(s, a)$ daquela ação naquele estado. Em outras palavras, este método deve acessar a estrutura de dados criada no item anterior para obter o Q-valor do estado e ação desejados. Caso o Q-valor seja desconhecido, retorne zero. **Importante: nos demais métodos, use este método para acessar sua estrutura de dados quando quiser obter um Q-valor.**

- O método `computeValueFromQValues`, que recebe um estado (parâmetro `state`) e deve retornar a função de valor naquele estado. Em outras palavras, o método deve retornar $V(s) = \max_a Q(s, a)$. Caso o estado seja terminal, retorne zero. Dica: use `self.getLegalActions(state)` para obter a lista de ações possíveis em um estado.

- O método `computeActionFromQValues`, que recebe um estado (parâmetro `state`) e deve retornar a melhor ação a ser feita naquele estado, de acordo com os Q-valores das ações disponíveis naquele estado. Em outras palavras, o método deve retornar $\pi(s) = \arg\max_a Q(s, a)$. Caso o estado seja terminal, retorne `None`. Empates entre Q-valores de ações devem ser resolvidos aleatoriamente. Dica: use `self.getLegalActions(state)` para obter a lista de ações possíveis em um estado.

- O método `update`, que recebe uma transição (s, a, s', r): um estado (parâmetro `state`), uma ação (parâmetro `action`), um estado seguinte (parâmetro `nextState`) e uma recompensa (parâmetro `reward`). Neste método, o aprendizado de fato acontece. Você deverá atualizar o Q-valor da tupla `(state, action)` utilizando a equação de atualização de Bellman: $Q(s, a) = Q(s, a) + \alpha(r + \gamma V(s') - Q(s, a))$, onde α e γ podem ser acessados por `self.alpha` e `self.discount`, respectivamente.

Para testar se sua implementação está correta, utilize o *autograder*:

```
python2 autograder.py --question passo5
```

Apesar dos Q-valores poderem ser inicializados com quaisquer valores, é importante que você inicialize seus Q-valores com zero, para que o *autograder* funcione corretamente. Para completar o **Passo 5**, você deverá implementar os métodos descritos acima e obter pontuação máxima (6/6) no *autograder*.

Passo 6: (3 pontos) Implementar o epsilon-greedy

Durante o aprendizado do Q-learning, caso o agente escolha sempre a melhor ação, o aprendizado terá grandes chances de não chegar em uma política boa, uma vez que o agente recusará oportunidades de descobrir ações melhores. De fato, a prova matemática de que o Q-learning consegue alcançar políticas ótimas envolve a premissa de que o agente deve ter uma probabilidade não-nula de escolher todas as ações. No entanto, se o objetivo do agente é de maximizar as recompensas obtidas, escolher ações subótimas pode dificultar o trabalho dele. Isso é conhecido como o dilema de *exploration vs. exploitation*.[4]

Uma das formas mais comuns de abordar este dilema é através do *epsilon-greedy*: em cada iteração, com probabilidade ϵ, o agente escolhe uma ação aleatória; e com probabilidade $1 - \epsilon$, escolhe a melhor ação. Você deverá implementar o método `getAction` da classe `QLearningAgent`, no arquivo `qlearningAgents.py`, que recebe um estado (parâmetro `state`) e deve retornar uma ação seguindo o *epsilon-greedy*. Caso o estado seja terminal, retorne `None` (não há nenhuma ação possível). O ϵ (epsilon) pode ser acessado por `self.epsilon`.

Dica 1: use `self.getLegalActions(state)` para obter a lista de ações possíveis em um estado.

Dica 2: o método `util.flipCoin(p)` retorna `True` com probabilidade `p` $\in [0, 1]$ e `False` caso contrário.

Dica 3: o método `random.choice(list)` retorna um elemento aleatório da lista `list`.

Para testar se sua implementação está correta, utilize o *autograder*:

```
python2 autograder.py --question passo6
```

Para completar o **Passo 6**, você deverá implementar o métodos descrito acima e obter pontuação máxima (3/3) no *autograder*. Para ver seu agente de Q-learning funcionando na prática, use:

```
python2 gridworld.py --agent q --episodes 100
```

Este comando executará 100 episódios de treinamento do Q-learning na fase `BookGrid` vista na aula 9 para obter Q-valores $Q(s, a)$ (e, por consequência, uma função de valor $V(s)$ e uma política π).

Passo 7: (1 ponto) Atravessar a ponte (de novo)

Neste passo, você usará novamente a fase `BridgeGrid` do Grid World. Como mostra a Figura 1 (no passo 3), esta fase possui dois estados terminais positivos cujas recompensas são, respectivamente, 1 e 10. Estes dois estados são separados por uma "ponte" estreita, de forma que "cair" em quaisquer dos estados terminais paralelos resulta em uma recompensa de -100. O estado inicial deste processo de decisão de Markov é o estado adjacente ao estado terminal de recompensa 1.

Primeiro, treine um agente de Q-learning totalmente aleatório ($\epsilon = 1$) na fase `BridgeGrid` sem ruído, e observe se ele consegue ou não encontrar a política ótima (atravessar a ponte):

```
python2 gridworld.py --agent q --episodes 50 --noise 0 --grid BridgeGrid
                     --epsilon 1 --learningRate 0.5
```

Depois, faça o mesmo experimento com $\epsilon = 0$:

[4] Em português, ambas as palavras se traduzem para "exploração". Uma tradução livre de qualidade duvidosa seria "investigação vs. aproveitamento".

```
python2 gridworld.py --agent q --episodes 50 --noise 0 --grid BridgeGrid
                     --epsilon 0 --learningRate 0.5
```

Para completar o **Passo 7**, você deverá sugerir um epsilon (ϵ) e uma taxa de aprendizado (α) que faça o agente atravessar a ponte ($\pi(s) = \text{direita}, \forall s$ não-terminal) após 50 iterações de aprendizado. Para tal, modifique as variáveis `answerEpsilon` e `answerLearningRate` no método `passo7` do arquivo `analysis.py` ou retorne a string `'NOT POSSIBLE'` caso seja impossível. Corrija sua resposta com o *autograder*:

```
python2 autograder.py --question passo7
```

Para ver o funcionamento do agente na prática, com quaisquer parâmetros, use:

```
python2 gridworld.py --agent q --episodes 50 --noise 0 --grid BridgeGrid
                     --epsilon X --learningRate Y
```

Onde `X` é o parâmetro epsilon e `Y` é a taxa de aprendizado (α) do algoritmo.

Passo 8: (1 ponto) Jogar o Pac-Man

Até agora, focamos no Grid World, que é um ambiente amigável para testarmos suas implementações e seu entendimento sobre os algoritmos. Estes, no entanto, são feitos para encontrar políticas ótimas em processos de decisão em Markov em geral, independente de qual problema está sendo modelado. Neste passo, apresentamos outro processo de decisão de Markov: jogar o Pac-Man. Nele, cada estado representa uma configuração distinta da fase (posição do Pac-Man, posição das comidas e posição dos fantasmas) e as ações representam os movimentos possíveis do Pac-Man (norte, sul, leste, oeste).

Para completar o **Passo 8**, sua implementação dos passos anteriores deve ser genérica o suficiente para funcionar com outros processos de decisão de Markov, e, portanto, você deve conseguir executar o comando a seguir sem problemas e treinar um agente Pac-Man que vença a fase ao menos 80% das vezes:

```
python2 pacman.py --pacman PacmanQAgent --numTraining 2000
                  --numGames 2010 --layout smallGrid
```

Com este comando, 2010 partidas do Pac-Man na fase `smallGrid` serão jogadas, sendo destas 2000 partidas de treino (acontecerão silenciosamente) e o restante, 10, de teste (acontecerão usando a interface gráfica do Pac-Man e usarão $\epsilon = 0$). Para avaliar este passo, use:

```
python2 autograder.py --question passo8
```

Sinta-se livre para experimentar outros parâmetros para o Q-learning através do parâmetro `--agentArgs` (por exemplo, `--agentArgs epsilon=0.1,alpha=0.3,gamma=0.7`. Se quiser assistir 10 episódios de treinamento para ver o que acontece, use:

```
python2 pacman.py --pacman PacmanQAgent --numGames 10 --layout smallGrid --agentArgs
                  numTraining=10
```

Passo 9: (2 pontos – bônus!) Jogar o Pac-Man (muito bem)

Apesar do bom resultado na fase `smallGrid`, ao treinar o mesmo agente do passo anterior na fase `mediumGrid` (que é um pouco maior), não teremos o mesmo sucesso. Isso acontece porque, enquanto a menor fase possui 288 estados possíveis, a maior possui 5616. E pela quantidade de partidas de treinamento que usamos, o agente não teve tempo suficiente para estimar Q-valores precisos para a maioria dos estados.

É fácil perceber que, em fases maiores do Pac-Man ou em problemas mais complexos,[5] o Q-learning não conseguirá achar boas políticas. Para abordar esta limitação, foram inventadas diversas variações do Q-learning e de outros algoritmos que resolvem versões *aproximadas* dos processos de decisão de Markov. Neste passo, você deverá implementar um deles: o Q-learning aproximado.

No Q-learning aproximado, ao invés de estimar um Q-valor para cada estado e ação existente, escolhemos um conjunto de *características* (em inglês, *features*) da fase e estimamos um *peso* para cada característica. O conjunto de características escolhido deve fornecer informação relevante para a escolha da ação. Por exemplo, a característica "distância do fantasma mais próximo" certamente contribuirá mais para a construção de uma boa política do que a "cor das paredes da fase".

Dado um conjunto de características $F = \{f_1(s,a), f_2(s,a), ..., f_n(s,a) \,|\, f_i : (\mathcal{S}, \mathcal{A}) \to \mathbb{R}\}$ e o conjunto de pesos $W = \{w_1, w_2, ..., w_n \,|\, w_i \in \mathbb{R}\}$, o Q-valor de um estado s e uma ação a é dado por:

$$Q(s,a) = F \cdot W = \sum_{i=1}^{n} f_i(s,a) w_i.$$

A atualização dos pesos é feita a cada interação com o ambiente, de forma similar aos Q-valores do Q-learning normal: $w_i = w_i + \alpha(r + \gamma V(s') - Q(s,a))$, onde $V(s') = \max_{a'} Q(s', a')$. Para completar o **Passo 9**, você deverá implementar o cálculo do Q-valor e a atualização dos pesos em dois métodos do arquivo `qlearningAgents.py` e obter pontuação máxima no *autograder* (2/2). Os métodos são:

- o método `getQValue`, que recebe um estado (parâmetro `state`) e uma ação (parâmetro `action`) e deve retornar o Q-valor $Q(s,a)$ daquela ação naquele estado, conforme a fórmula acima. Para obter o dicionário de características, use `self.featExtractor.getFeatures(state, action)`. As chaves deste dicionário são os nomes das características e seus valores são os $f_i(s,a)$ respectivos. Para obter o dicionário de pesos, use `self.getWeights()`. As chaves deste dicionário são os nomes das características e seus valores são os w_i respectivos. **Importante: no método seguinte, use este método quando quiser obter um Q-valor.**
- O método `update`, que recebe uma transição (s, a, s', r): um estado (parâmetro `state`), uma ação (parâmetro `action`), um estado seguinte (parâmetro `nextState`) e uma recompensa (parâmetro `reward`). Você deverá atualizar cada um dos pesos utilizando a equação descrita acima, onde α e γ podem ser acessados por `self.alpha` e `self.discount`, respectivamente.

Não se preocupe em criar características – o código-fonte já vem com dois extratores de características definidos. O primeiro é o `IdentityExtractor`, que atribui uma característica para cada estado possível (e, consequentemente, acaba sendo a mesma coisa que o Q-learning normal). O segundo é o `SimpleExtractor`, que usa as características: (i) se a ação a fará o Pac-Man comer alguma comida; (ii) distância da comida mais próxima; (iii) se a ação a fará o Pac-Man colidir com um fantasma; e (iv) se há um fantasma a um quadrado de distância. Para testar se sua implementação está correta, utilize o *autograder*:

```
python2 autograder.py --question passo9
```

Com a implementação correta do Q-learning aproximado, o comando a seguir deve gerar o mesmo resultado que o agente do passo anterior:

```
python2 pacman.py --pacman ApproximateQAgent --numTraining 2000 --numGames 2010
        --layout smallGrid --agentArgs extractor=IdentityExtractor
```

O agente também não deve ter problemas em jogar a fase `mediumGrid`:

[5] É comum encontrarmos problemas cuja representação na forma de processo de decisão de Markov possui mais estados do que há átomos no universo.

```
python2 pacman.py --pacman ApproximateQAgent --numTraining 50 --numGames 60
        --layout mediumGrid --agentArgs extractor=SimpleExtractor
```

Ou, finalmente, a fase original do Pac-Man (talvez demore um pouco mais para treinar):

```
python2 pacman.py --pacman ApproximateQAgent --numTraining 50 --numGames 60
      --layout originalClassic --agentArgs extractor=SimpleExtractor
```

Se funcionou direitinho, parabéns: você criou um agente jogador de Pac-Man!

Entrega

Você deverá entregar um arquivo ZIP no Moodle da disciplina, no item "Entrega" da seção "Trabalho Prático 2: MDPs e Aprendizado por Reforço'. O arquivo ZIP deve conter:

(i) Um arquivo Python `valueIterationAgents.py` com a sua implementação do algoritmo value iteration, conforme pedido no passo 2; e

(ii) Um arquivo Python `qlearningAgents.py` com a sua implementação do algoritmo Q-learning com epsilon-greedy, conforme pedido nos passos 5 e 6, e do Q-learning aproximado, conforme pedido no passo 9, caso tenha o feito.

(iii) Um arquivo Python `analysis.py`, com sua respostas para as questões enunciadas nos passos 3, 4 e 7.

Você deverá efetuar a entrega até o dia **26 de Março de 2021** às **23:59**. Após o prazo, serão descontados $2^d - 1$ pontos, onde d é o número de dias de atraso arredondado para cima.

Considerações finais

- O trabalho é **individual**. Sinta-se livre para discuti-lo com seus colegas, mas o compartilhamento de código-fonte é plágio e será devidamente punido. O código-fonte submetido deve ser de sua autoria.
- Em caso de dúvidas, não hesite em perguntar na seção `Dúvidas / Discussão` do Moodle da disciplina (sua dúvida pode ser a dúvida de outras pessoas) ou procurar o monitor da disciplina via e-mail (`ronaldo.vieira@dcc.ufmg.br`).
- Segundo dados do MECM (Minha Experiência Como Monitor), os alunos que começam a desenvolver o trabalho mais cedo obtém melhores resultados. Comece cedo, se possível!

Bom trabalho!