# MAC0425/5739 - Inteligência Artificial

## *Exercício-Programa 3*

## *Aprendizado por Reforço*

Prazo limite de entrega: 9/7/23 às 23h55

### 1) Introdução

Neste exercício-programa, vocês irão implementar os algoritmos Iteração de Valor e Q-learning. Os agentes serão testados primeiro no Gridworld (exemplo das aulas), depois em um simulador de robô (Crawler) e no Pacman.

Pac-Man procura a recompensa.
Ele deve correr ou deve comer?
Na dúvida, deve aprender.

**Os objetivos deste exercício-programa são:**

- Familiarizar-se com um Processo de Decisão Markoviano (MDP) e compreender o compromisso entre custos/recompensas e as probabilidades de sucesso/falha de um problema de decisão.
- Implementar o algoritmo de Iteração de Valor.
- Implementar o algoritmo Q-Learning de Aprendizado por Reforço.

**Instalação:**

Faça o download do arquivo ep3.zip disponível no eDisciplinas. Descompacte o arquivo e rode na raiz do diretório os seguintes comandos para testar a instalação:

```
python gridworld.py -m
```

```
python pacman.py
```

### 2) Códigos

**Arquivos que serão editados:**

- `valueIterationAgents.py`: um agente que executa o algoritmo de iteração de valor para um MDP conhecido;
- `qlearningAgents.py`: agentes que executam o algoritmo Q-learning para o Gridworld, Crawler e Pacman;
- `analysis.py`: um arquivo para colocar as suas respostas às perguntas abaixo em relação a parâmetros do algoritmo.

**Arquivos que devem ser lidos mas não editados:**

- `mdp.py`: define métodos para MDPs gerais;
- `learningAgents.py`: define as classes-base *ValueEstimationAgent* e *QLearningAgent*, que os seus agentes devem estender;
- `util.py`: funções auxiliares que podem ser utilizadas no trabalho, incluindo *util.Counter*, que é especialmente útil para o q-learning;
- `gridworld.py`: a implementação do Gridworld;
- `featureExtractors.py`: classes para extrair atributos de pares (estado,ação). Usadas para o agente de q-learning aproximado (em *qlearningAgents.py*).

**Arquivos que podem ser ignorados**

- `environment.py`: classe abstrata para ambientes gerais de aprendizagem por reforço. Usada por *gridworld.py;*
- `graphicsGridworldDisplay.py`: visualização gráfica do Gridworld;
- `graphicsUtils.py`: funções auxiliares para visualização gráfica;
- `textGridworldDisplay.py`: plug-in para a interface de texto do Gridworld;
- `crawler.py`: o código do agente crawler. Será executado, mas não modificado;
- `graphicsCrawlerDisplay.py`: GUI para o robô Crawler.

## 3) Parte prática

Você deverá implementar algumas funções nos arquivos *valueIterationAgents.py* e *qlearningAgents.py*, além de modificar algumas funções do arquivo *analysis.py*. Não esqueça de remover o código *util.raiseNotDefined()* ao final de cada função.

## MDPs

Para começar, execute o Gridworld no modo de controle manual, que usa as setas:

```
python gridworld.py -m
```

Você deve ver o exemplo que vimos em sala de aula. O ponto azul é o agente. Note que, quando você pressiona a seta pra cima, o agente só se move pra cima 80% das vezes, de acordo com a característica do ambiente.

Vários aspectos da simulação podem ser controlados. Uma lista completa de opções pode ser obtida com:

```
python gridworld.py -h
```

O agente default se move aleatoriamente:

```
python gridworld.py -g MazeGrid
```

Você deve ver o agente aleatório passear pelo grid até encontrar uma saída.

*Nota:* O MDP do gridworld foi implementado de tal forma que você primeiro deve entrar em um estado pré-terminal (as caixas duplas mostradas no grid) e depois executar a ação especial 'exit' para que o episódio realmente termine (o agente entra no estado `TERMINAL_STATE`, que não é mostrado na interface). Se você executar um episódio manualmente, o seu retorno pode ser menor do que o esperado devido à taxa de desconto (`-d` para mudar; 0.9 por default).

Olhe para a saída na linha de comando python que fica por trás da visualização gráfica (ou use `-t` para suprimir a visualização gráfica). Você verá o que aconteceu em cada transição do agente (para desligar essa saída, use `-q`).

Como no Pacman, posições são representadas por coordenadas cartesianas `(x,y)` e os arrays são indexados por `[x][y]`, com `'north'` sendo a direção de aumento do `y`, etc. Por default, na maioria das transições (não-terminais) o agente vai receber recompensa 0, mas isso pode ser mudado com a opção (`-r`).

**Passo 1 (3 pontos):** escreva um agente de iteração de valor em `ValueIterationAgent`, que está parcialmente especificada no arquivo `valueIterationAgents.py`. O seu agente de iteração de valor é um planejador offline, não um agente de aprendizado por reforço; então a opção relevante nesse caso é o número de iterações do algoritmo de iteração de valor (opção `-i`) na fase de planejamento. O agente `ValueIterationAgent` recebe um MDP e executa o algoritmo de iteração de valor pelo número especificado de iterações no próprio construtor. Use a equação de atualização de estados:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s')\right]$$

Para rodar os casos de teste, execute o comando:

```
python autograder.py -q q1
```

Seu agente de iteração de valor será avaliado em um novo grid. Nós vamos checar seus valores, Q-valores e políticas depois de um número fixo de iterações e na convergência (por exemplo, após 100 iterações).

O algoritmo de iteração de valor calcula estimativas dos valores de utilidade ótimos considerando k passos, $V_k$. Além de implementar a iteração de valor, implemente os seguintes métodos para o agente `ValueIterationAgent` usando $V_k$:

- `computeActionFromValues(state)` retorna a melhor ação de acordo com os valores dados por `self.values`
- `computeQValueFromValues(state, action)` retorna o q-valor do par (state, action) de acordo com o valor dado por `self.values`

Essas quantidades são mostradas na GUI: valores/q-valores são os números dentro dos quadrados e políticas são representadas por setas em cada estado.

***Observações:***

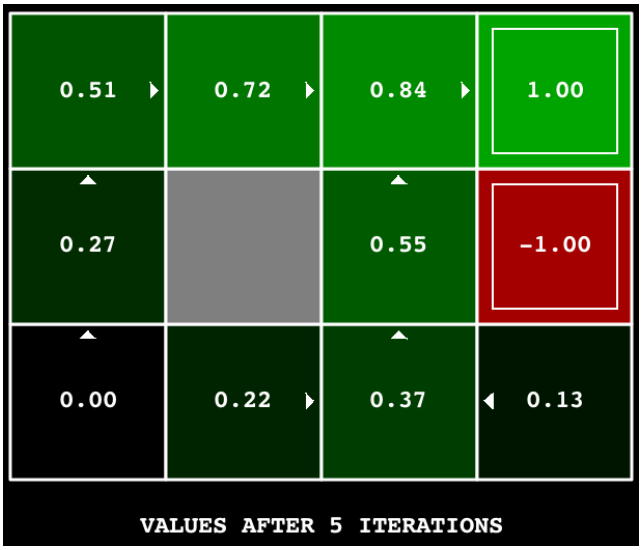
- use a versão "batch" da iteração de valor onde cada vetor $V_k$ é calculado a partir de um vetor fixo da iteração anterior $V_{k-1}$ (como na aula), e não a versão "online" em que os valores são atualizados continuamente. Essa diferença é discutida em Sutton & Barto no sexto parágrafo do capítulo 4.1;
- a política obtida dos valores de utilidade da iteração k (que levam em consideração as próximas k recompensas) deve refletir as próximas k+1 recompensas (isto é, você deve retornar $\pi_{k+1}$). Da mesma forma, os q-valores também devem refletir as próximas k+1 recompensas (isto é, você deve retornar $Q_{k+1}$). Isso acontece naturalmente quando utilizamos as fórmulas adequadas;
- você pode escolher qualquer ação ótima em caso de empate no valor da função Valor;
- note que o critério de parada do algoritmo é o número de iterações dado na instanciação da classe *ValueIterationAgent*.
- utilize a estrutura de dado *Counter* disponível no arquivo *util.py* para implementação de distribuição de probabilidades. Essa implementação é necessária para manter a compatibilidade com o aquivo de testes (*autograder.py*). Métodos como o `totalCount` podem ajudar a simplificar o seu código. Porém tome cuidado com o `argMax`: o argmax que você quer pode não ter chave no contador! Listas, tuplas, tuplas nomeadas, conjuntos e dicionários do Python podem ser utilizados sem problemas caso necessário.

  Você pode supor que 100 iterações são suficientes para convergência nas perguntas abaixo.

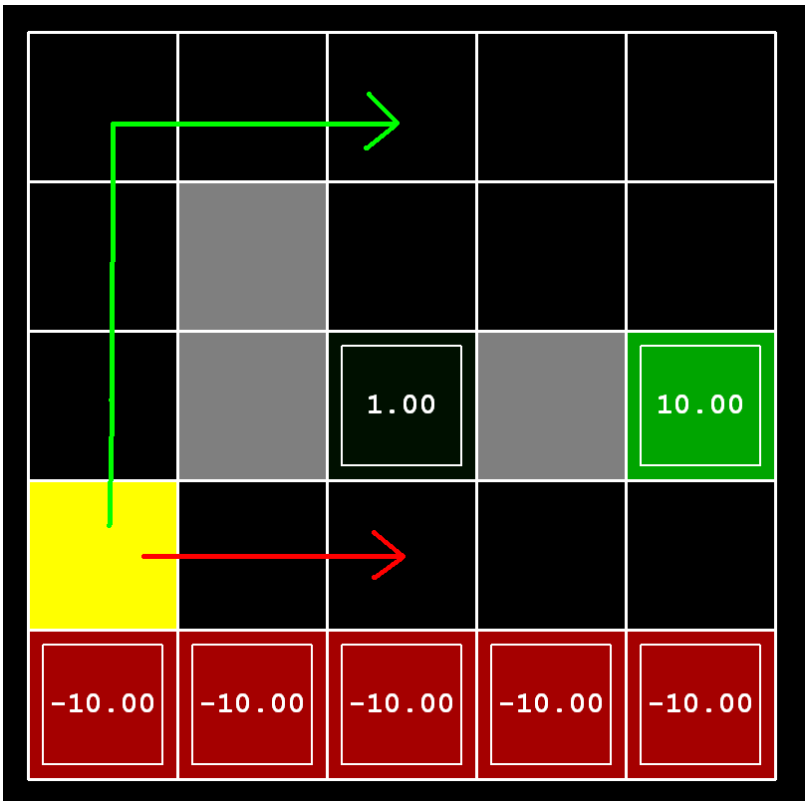
O comando seguinte carrega o seu agente `ValueIterationAgent`, que irá calcular uma política e executá-la 10 vezes. Aperte qualquer tecla pra ver os valores, q-valores e a simulação. Você deve ver que o valor do estado inicial (`V(start)`) e a média empírica das recompensas devem ser bem próximos.

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

*Dica:* No grid default BookGrid, rodar a iteração de valor por 5 iterações deve dar a seguinte saída (figura na próxima página): `python gridworld.py -a value -i 5`




**Passo 2 (3 pontos):** considere o `DiscountGrid`, mostrado abaixo. Esse grid tem dois estados terminais com recompensa positiva (mostrados em verde), uma saída próxima com recompensa +1 e uma saída mais distante com recompensa +10. A linha mais baixa do grid consiste de estados terminais com recompensa negativa (mostrados em vermelho); cada estado nessa região de "precipício" tem recompensa -10. O estado inicial é o quadrado amarelo. Podemos distinguir entre dois tipos de caminho: (1) caminhos que "arriscam o precipício" e passam perto da linha mais baixa do grid; estes caminhos são mais curtos mas têm risco maior de uma recompensa negativa, e são representados pela seta vermelha na figura abaixo; (2) caminhos que "evitam o precipício" e passam ao longo da parte de cima do grid; estes caminhos são mais longos mas tem menos risco de levar a grandes recompensas negativas, e são representados por uma seta verde na figura abaixo:



Dê uma atribuição de valores de parâmetros para desconto, ruído e recompensa que produzam os seguintes tipos de política ótima ou diga que a política é impossível, retornando a string `'NOT POSSIBLE'`. O default corresponde a:

```
    python gridworld.py -a value -i 100 -g DiscountGrid --discount 0.9 --noise 0.2 --livingReward 0.0
```

a. Preferir a saída mais próxima (+1), arriscando cair no precipício (-10)
b. Preferir a saída mais próxima (+1), evitando o precipício (-10)
c. Preferir a saída mais distante (+10), arriscando cair no precipício (-10)
d. Preferir a saída mais distante (+10), evitando o precipício (-10)
e. Evitar ambas as saídas, também evitando o precipício (logo, um episódio nunca temina)

As respostas dos itens 2(a) a 2(e) acima devem ser colocadas em `analysis.py` nas definições `question2a()` a `question2e()`.

*Nota:* Você pode verificar suas políticas na GUI. Por exemplo no item 2(a), a seta em (0,1) deve apontar para a direita, a seta em (1,1) também deve apontar para a direita, e a seta em (2,1) deve apontar para cima. Em algumas máquinas, a seta pode não aparecer. Neste caso, pressione uma tecla para mudar a visualização para qValue e verifique a política observando o valor máximo.

Para rodar os casos de teste, execute o comando:

```
python autograder.py -q q2
```

Para a nota, checaremos se a política desejada é devolvida em cada caso.

## Q-learning

O seu agente de iteração de valor não aprende a partir da própria experiência. Ao invés disso, ele usa o seu modelo MDP para calcular uma política completa antes de interagir com o ambiente. Quando ele interage com o ambiente, ele simplesmente segue uma política pré-calculada (isto é, ele se torna um agente reativo). Essa diferença pode parecer sutil em um ambiente simulado como o Gridword, mas é importante no mundo real onde nem sempre se conhece o MDP verdadeiro.

**Passo 3 (3 pontos):** você agora irá criar um agente q-learning que aprende a partir de interações com o ambiente através do método `update(state, action, nextState, reward)`. Um stub do q-learner foi especificado na classe `QLearningAgent` em `qlearningAgents.py`, e você pode selecioná-lo com a opção `'-a q'`. Nesse passo, você deve implementar os métodos `update`, `computeValueFromQValues`, `getQValue`, e `computeActionFromQValues`.

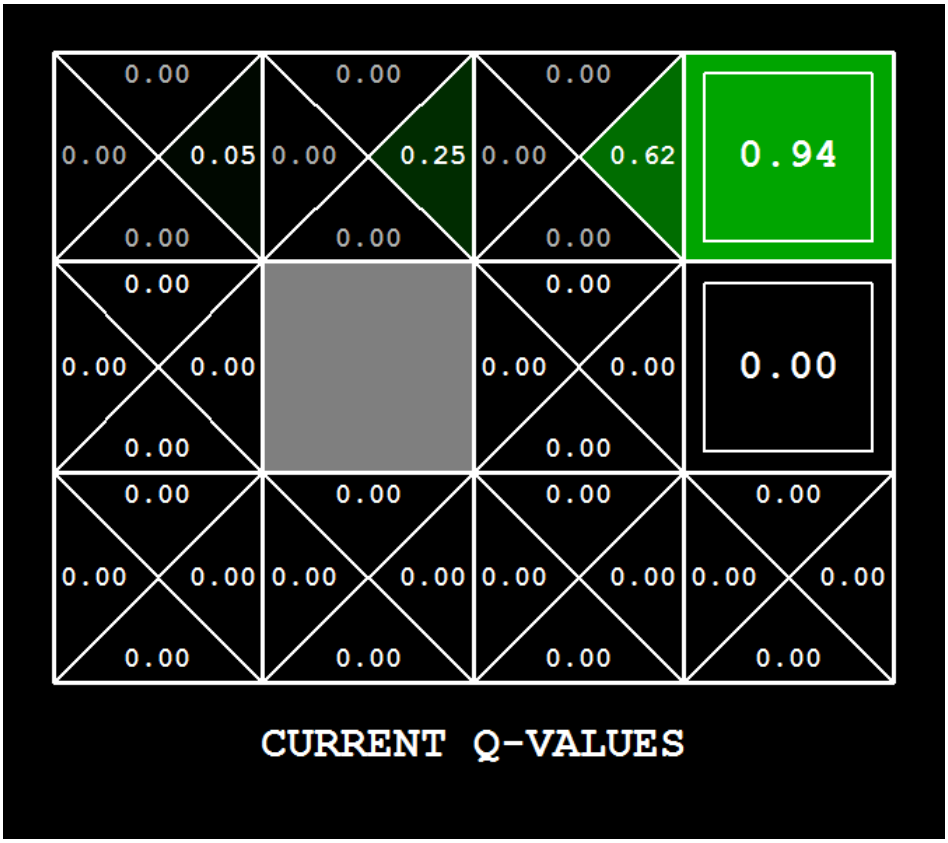
*Nota:* para `computeActionFromQValues` você deve resolver empates aleatoriamente para um comportamento melhor. A função `random.choice()` será útil pra isso. Em cada estado, ações que o agente ainda *não* executou devem ter um Q-valor de zero, e se todas as ações que o agente já tiver executado tiverem um Q-valor negativo, a ação não executada pode ser ótima.

*Importante:* você só deve acessar os Q-valores utilizando o método `getQValue` nas funções `getValue` e `getPolicy`. Isso será útil no passo 9.

Agora você pode ver o agente aprendendo sob controle manual, usando o teclado:

```
python gridworld.py -a q -k 5 -m
```

Lembre que o parâmetro `-k` controla o número de episódios de aprendizagem. Observe como o agente aprende sobre o estado em que estava, não aquele para o qual se move. Dica: para ajudar na depuração, você pode desativar o ruído usando o parâmetro --noise 0.0 (embora isso obviamente torne o Q-learning menos interessante). Se você direcionar manualmente o agente para o norte e depois para o leste ao longo do caminho ideal por quatro episódios, deverá ver os seguintes valores Q:




Para rodar os casos de teste, execute o comando:

```
python autograder.py -q q3
```

Para a nota, vamos executar o seu agente q-learning e checar se ele aprende os mesmos Q-valores e política que a nossa implementação de referência quando cada um é submetido ao mesmo conjunto de exemplos.

**Passo 4 (1 ponto):** complete o seu agente de q-learning implementando a seleção de ações epsilon-gulosa em `getAction`, significando que ele escolhe ações aleatórias com probabilidade epsilon, e segue os melhores q-valores com probabilidade 1-epsilon. Observe que a escolha de uma

ação aleatória pode resultar na escolha da melhor ação - ou seja, você não deve escolher uma ação aleatória abaixo do ideal, mas sim qualquer ação legal aleatória.

Você pode escolher aleatoriamente um elemento de uma lista chamando a função `random.choice`. Você pode simular uma variável binária com probabilidade p de sucesso usando `util.flipCoin(p)`, que retorna True com probabilidade p e False com probabilidade 1-p. Após implementar o método `getAction`, observe o seguinte comportamento do agente no GridWorld (com epsilon = 0.3).

```
python gridworld.py -a q -k 100
```

Os q-valores finais devem ser parecidos com os do agente de iteração de valor, especialmente em caminhos por onde o agente passa muitas vezes. Porém, a soma das recompensas será menor do que os q-valores por causa das ações aleatórias e da fase inicial de aprendizagem.

Você pode observar as simulações abaixo para testar suas intuições:

```
python gridworld.py -a q -k 100 --noise 0.0 -e 0.1

python gridworld.py -a q -k 100 --noise 0.0 -e 0.9
```

Para rodar os casos de teste, execute o comando:

```
python autograder.py -q q4
```

Sem nenhum código adicional, você deveria conseguir rodar o Q-lerning com o robô crawler:

```
python crawler.py
```

Se isso não funcionar, você provavelmente escreveu algum código muito específico para o problema do GridWorld e deve torná-lo mais geral para todos os MDPs. Brinque com os vários parâmetros de aprendizado para ver como eles afetam as políticas e ações do agente. Observe que o atraso do passo é um parâmetro da simulação, enquanto a taxa de aprendizado e o epsilon são parâmetros do seu algoritmo de aprendizado e o fator de desconto é uma propriedade do ambiente.

**Sessão Bônus!**

## Q-learning Aproximado

**Passo** *5* **(***0.5* **ponto):** hora de jogar Pacman! O Pacman vai jogar jogos em duas fases. Na primeira fase, de *treinamento*, o Pacman vai começar a aprender os valores dos estados e ações. Mesmo para grids pequenos, o Pac-Man demora muito tempo para aprender os q-valores, por isso a fase de treinamento não é mostrada na GUI ou na linha de comando. Quando o treinamento termina,

começa a fase de teste. Na fase de teste, os parâmetros `self.epsilon` e `self.alpha` do Pacman serão fixos em 0.0, efetivamente parando o aprendizado (e a exploração) para que o Pacman possa explorar (aproveitar) a política aprendida. Essa fase é mostrada na GUI por default. Sem mudar nada no seu código você deve ser capaz de rodar um agente de q-learning para o Pacman em tabuleiros pequenos da seguinte forma:

```
python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid
```

Note que `PacmanQAgent` já está definido em termos do `QLearningAgent`. O `PacmanQAgent` só é diferente porque ele tem parâmetros mais eficientes para o Pacman (`epsilon=0.05, alpha=0.2, gamma=0.8`). Com esses parâmetros, o seu agente deve ganhar 80% dos últimos 10 episódios. O autograder vai rodar 100 jogos de teste após os 2000 de treinamento.

*Dica:* Se o seu `QLearningAgent` funciona para o `gridworld.py` e para o `crawler.py` mas não consegue aprender uma boa política para o Pacman no `smallGrid`, pode ser que o seu código dos métodos `getAction` e/ou `computeActionFromQValues` não consideram em alguns casos ações não executadas de maneira correta. Em particular, como as ações não vistas têm, por definição, um valor Q igual a zero, se todas as ações que foram vistas tiverem valores Q negativos, uma ação não vista pode ser ótima. Cuidado com a função `argMax` de `util.Counter`!

*Nota:* Se quiser mudar os parâmetros de aprendizagem, use a opção `-a`, por exemplo `-a epsilon=0.1,alpha=0.3,gamma=0.7`. Estes valores aparecerão como `self.epsilon`, `self.gamma` e `self.alpha` dentro do agente.

*Nota:* Embora um total de 2010 jogos vão ser jogados, os primeiros 2000 jogos não serão mostrados por causa da opção `-x 2000`, que designa os 2000 primeiros jogos para treinamento. Logo, você só verá o Pacman jogar os últimos 10 desses jogos, na fase de teste. O número de jogos de treinamento também pode ser passado para o agente com a opção `numTraining`.

*Nota:* Se você quiser ver 10 jogos de treinamento, use o comando:

```
python pacman.py -p PacmanQAgent -n 10 -l smallGrid -a numTraining=10
```

Durante o treinamento, você verá uma saída a cada 100 jogos com estatísticas sobre como o Pacman está se saindo. O epsilon é positivo no treinamento, então o Pacman vai jogar mal mesmo depois de ter aprendido uma boa política: isso porque ele vai ocasionamente fazer um movimento aleatório em direção a um fantasma. Deve demorar mais ou menos entre 1000 e 1400 jogos até que as recompensas do Pacman para um segmento de 100 episódios fiquem positivas, mostrando que ele começou a ganhar mais do que perder. Até o final do treinamento as recompensas devem continuar positivas e razoavelmente altas (entre 100 e 350).

Você deve entender que: o estado do MDP é a configuração *exata* do tabuleiro, com a função de transição descrevendo todas as possíveis mudanças daquele estado, considerando simultaneamente tanto o Pacman quanto os fantasmas. As configurações intermediárias em que o Pacman se moveu mas os fantasmas ainda não reagiram não são estados do MDP.

Quando o Pacman termina a fase de treinamento, ele deve passar a ganhar na fase de teste pelo menos 90% do tempo, já que ele estará utilizando a política aprendida. Porém, treinar o mesmo agente no `mediumGrid` pode não funcionar bem. Na nossa implementação, as recompensas médias do Pacman na fase de treinamento ficam sempre negativas. E, na fase de teste, ele provavelmente perde todos os jogos, mesmo levando muito tempo para o treinamento. Isso acontece em tabuleiros maiores porque cada configuração do tabuleiro é um estado separado com q-valores próprios. Ele não tem como fazer a generalização de que encostar em um fantasma é ruim em qualquer posição.

Para rodar os casos de teste, execute o comando:

```
python autograder.py -q q5
```

**Passo** *6* **(1 ponto):** implemente um agente de q-learning aproximado que aprenda pesos para características do estado, onde os estados compartilham características. Escreva sua implementação na classe `ApproximateQAgent` em `qlearningAgents.py`, que é uma subclasse de `PacmanQAgent`.

*Nota:* O q-learning aproximado supõe a existência de uma função de características f(s,a) que recebe pares estado-ação, e retorna um vetor [$f_1$(s,a), …, $f_i$(s,a), …, $f_n$(s,a) de características. Funções de características são dadas em `featureExtractors.py`. Vetores de características são objetos `util.Counter` que contém pares não nulos de característica-valor; todas as características omitidas tem valor zero. Então, ao invés de um vetor em que o índice do vetor define qual característica é qual, as chaves do dicionário identificam as características.

A função Q aproximada tem a seguinte forma:

$$Q(s,a) = \sum_{i}^{n} f_i(s,a) w_i$$

em que cada $w_i$ está associado com uma característica $f_i$(s,a). No seu código, você deve implementar o vetor de pesos como um dicionário mapeando características (que as funções extratoras de características retornarão) a pesos. A atualização dos pesos é feita de forma similar à dos q-valores:

$$w_i \leftarrow w_i + \alpha \cdot \text{difference} \cdot f_i(s,a)$$

$$\text{difference} = \left(r + \gamma \max_{a'} Q(s',a')\right) - Q(s,a)$$

Note que o termo de correção é o mesmo que o do Q-learning normal e r é a recompensa recebida.

Por default, o `ApproximateQAgent` usa o `IdentityExtractor`, que atribui uma característica para cada par `(estado,ação)`. Com esse extrator de característica, seu agente de q-learning aproximado deve funcionar de forma idêntica ao `PacmanQAgent`. Você pode testar isso com o seguinte comando:

```
python pacman.py -p ApproximateQAgent -x 2000 -n 2010 -l smallGrid
```

*Importante:* `ApproximateQAgent` é uma subclasse de `QLearningAgent`, logo ela herda vários métodos como `getAction`. Os métodos em `QLearningAgent` devem chamar `getQValue` ao invés de acessar os q-valores diretamente, pra que os novos valores aproximados sejam utilizados.

Quando o seu agente funcionar corretamente com o `IdentityExtractor`, rode o seu Q-learning aproximado com outros extratores e veja o Pacman ganhar:

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

Até tabuleiros maiores podem ser aprendidos com o `ApproximateQAgent` (pode demorar alguns minutos para treinar):

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

Se não houver erros, o seu agente de q-learning aproximado deve ganhar quase sempre com essas características simples, mesmo com só 50 jogos de treinamento.

Para rodar os casos de teste, execute o comando:

```
python autograder.py -q q6
```

Para a nota, nós vamos executar o seu agente de q-learning aproximado e checar se ele aprende os mesmos Q-valores e pesos das características que a nossa implementação de referência quando submetidos ao mesmo conjunto de exemplos.

## 4) Entrega

Uma das pessoas da dupla deve entregar um arquivo `ep3-SeusNomesVaoAqui.zip` contendo **APENAS** os arquivos *valueIterationAgents.py*, *qlearningAgents.py* e *analysis.py*, que serão modificados no trabalho. Não esqueça de identificar cada arquivo com seu nome e número USP! Para os códigos, coloque um cabeçalho em forma de comentário.

Desta vez, não haverá relatório a ser entregue.

## 5) Avaliação

O critério de avaliação dependerá principalmente dos resultados dos testes automatizados do `autograder.py`. Desta forma, você terá como avaliar por si só parte da nota que receberá para a parte prática.