Trabalho 3: Aprendizagem por Reforço

Este trabalho é parte do Pacman Project desenvolvido na UC Berkeley disciplina CS188 – Artificial Intelligence. Tradução realizada pelo prof. Eduardo Bezerra, CEFET/RJ). ***Nota: as questões 4 e 5 foram omitidas deliberadamente.***

Pacman procura a recompensa.
Ele deve correr ou deve comer?
Na dúvida, deve aprender.

Introdução

Neste trabalho, você irá implementar os algoritmos iteração de valor e q-learning. Os agentes serão testados primeiro no Gridworld (exemplo das aulas), depois num simulador de robô (Crawler) e no Pacman.

O código deste projeto contém os seguintes arquivos, que estão disponíveis em um arquivo zip:

Arquivos que serão editados

valueIterationAgents.py	Um agente que executa o algoritmo de iteração de valor para um PDM 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 PDMs 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.

O que deve ser entregue:

Os arquivos valueIterationAgents.py, qlearningAgents.py e analysis.py serão modificados no trabalho. Você deve submeter apenas esses arquivos. Por favor, não modifique nenhum outro arquivo. Cada aluno deve entregar esses arquivos e deve entregar também um relatório mostrando o resultado de cada execução. O trabalho é individual.

PDMs

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

```
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 PDM do gridworld foi implementado de tal forma que você primeiro deve entrar em um estado pré-terminal (i.e., 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`).

QUESTÃO 1 (4 PONTOS) – VALUE ITERATION

Escreva um agente de iteração de valor em `ValueIterationAgent`, que está parcialmente especificado 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 PDM e executa o algoritmo de iteração de valor pelo número especificado de iterações no próprio construtor.

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

- `computeActionFromValues(state)` retorna a melhor ação para um dado estado a partir dos valores dos estados em `self.values`.
- `computeQValueFromValues(state, action)` retorna o q-valor do par (`state, action`) a partir dos valores em `self.values`.

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

***Importante**:* 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"

onde os valores são atualizados continuamente. Essa diferença é discutida em Sutton & Barto no sexto parágrafo da Seção 4.1 (página 91).

Nota: Uma política obtida a partir 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 π_{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.

Dica: você pode usar opcionalmente a classe `util.Counter` em util.py, que é um dicionário com valor padrão zero. No entanto, tome cuidado com `argMax`: o argmax real que você deseja pode ser uma chave que não está no contador!

Nota: Certifique-se de lidar com o caso quando um estado não tem ações disponíveis em um MDP (pense no que isso significa para recompensas futuras).

Para testar sua implementação, execute o autograder:

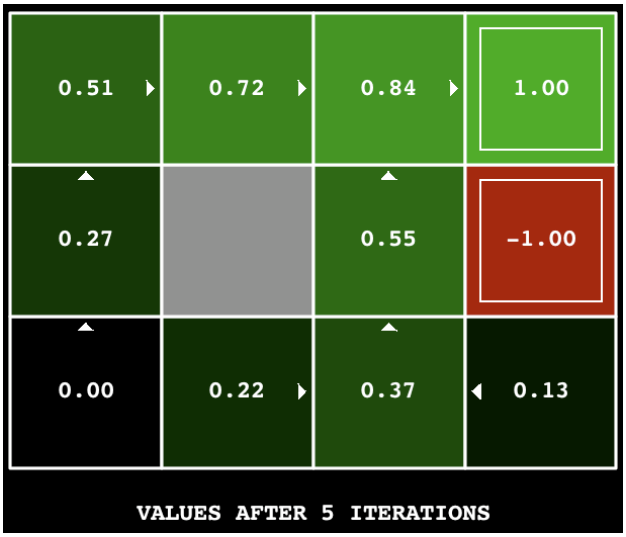
```
python autograder.py -q q1
```

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 (que são impressos após 10 rodadas de execução) devem ser bem próximos.

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

Dica: No grid default BookGrid, rodar o algoritmo Iteração de Valor por 5 iterações deve produzir a seguinte saída:

```
python gridworld.py -a value -i 5
```



Dica: Use a classe `util.Counter` em util.py, que é um dicionário com valor default 0. 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!

QUESTÃO 2 (1 PONTO) – TRAVESSIA DE PONTE

`BridgeGrid` é um mapa em grade com um estado terminal de baixa recompensa e um estado terminal de alta recompensa separados por uma “ponte” estreita, em cada lado da qual está um abismo de alta recompensa negativa. O agente começa próximo ao estado de baixa recompensa. No `BridgeGrid` com o desconto default de 0.9 e o ruído default de 0.2, a política ótima não atravessa a ponte. Modifique somente UM dos parâmetros: ou o desconto ou o ruído de tal forma que a política ótima faça com que o agente tente atravessar a ponte. Coloque sua resposta em `question2()` do `analysis.py`. (Ruído é a probabilidade de que o agente vá parar no estado "errado" quando ele executa uma ação). Os parâmetros default correspondem a:

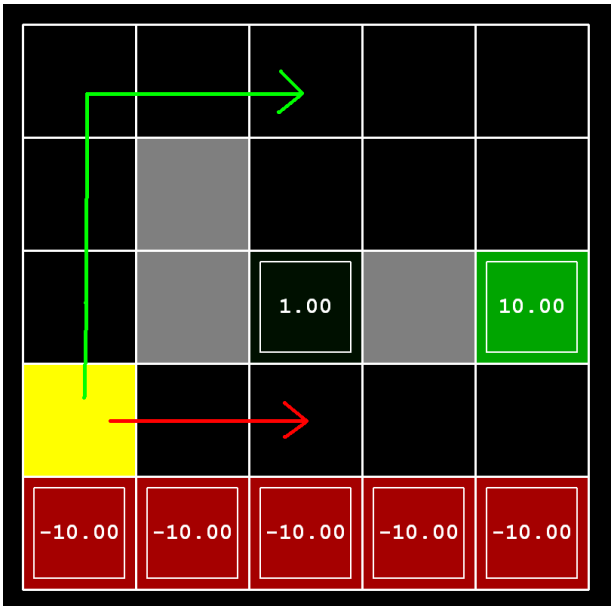
```
python gridworld.py -a value -i 100 -g BridgeGrid --discount 0.9 --noise 0.2
```

Para testar sua implementação, execute o autograder:

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

QUESTÃO 3 (5 PONTOS) - POLÍTICAS

Considere o `DiscountGrid` mostrado abaixo. Este 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.



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

`Abaixo está a lista de tipos de políticas ótimas que você deve tentar produzir:`

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)

Para testar sua implementação, execute o autograder:

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

As respostas dos itens 3(a) a 3(e) acima devem ser colocadas em analysis.py nas definições `question3a()` a `question3e()`.

Nota: Você pode verificar suas políticas na interface gráfica. Por exemplo, no item 3(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.

Nota: Em algumas máquinas, você pode não ver uma seta. Nesse caso, pressione um botão no teclado para alternar para a exibição de qValue e calcule mentalmente a política tomando o arg max dos qValues disponíveis para cada estado.

QUESTÃO 6 (4 PONTOS) – Q-LEARNING

O seu agente de iteração de valor não aprende a partir da própria experiência. Em vez disso, ele usa o seu modelo PDM para calcular uma política completa antes de interagir com o ambiente. Quando ele interage com o ambiente, ele simplesmente segue uma política previamente 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 PDM verdadeiro.

Você agora irá criar um agente Q-learning, que aprende a partir de interações com o ambiente por meio 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'`. Nessa questão, você deve implementar os métodos `update`, `computeValueFromQValues`, `getQValue` e `computeActionFromQValues`.

Nota: Para `computeActionFromQValues`, você deve resolver empates aleatoriamente para produzir um comportamento melhor. A função `random.choice()` será útil para 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: Se assegure de que nas funções `computeValueFromQValues` e `computeActionFromQValues` você faça acesso aos Q-valores apenas utilizando o método `getQValue`. Essa abstração será útil na Questão 10, na qual você irá sobrescrever `getQValue` para usar características de pares estado/ação em vez de usar pares estado/ação diretamente.

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

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

Lembre-se de 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 ele se move, e "deixa o aprendizado em seu rastro". Dica: para ajudar na depuração, você pode desligar o ruído usando o parâmetro `--noise 0.0` (embora isso obviamente torne o Q-learning menos interessante).

Para testar sua implementação, execute o autograder:

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

QUESTÃO 7 (1 PONTO) – ϵ-GREEDY

Complete o seu agente Q-learning implementando a seleção de ações epsilon-greedy em `getAction`, significando que ele escolhe ações aleatórias com probabilidade ϵ, e segue os melhores q-valores com probabilidade $1 - \epsilon$.

Você pode escolher um elemento aleatoriamente de uma lista chamando a função `random.choice`. Você pode simular uma variável binária aleatória 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 também pode observar as seguintes simulações para diferentes valores de épsilon. Esse comportamento do agente corresponde ao que você esperava?

```
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 testar sua implementação, execute o autograder:

```
python autograder.py -q q7
```

Sem modificar nada, você deve ser capaz de executar o robô Crawler, que também aprende com q-learning:

```
python crawler.py
```

Se não funcionar, você deve ter feito algo específico para o `GridWorld` e deve consertar o código para que ele seja genérico para qualquer PDM.

Isso invocará o robô rastejante usando seu Q-learner. Teste diferente valores dos parâmetros de aprendizagem para ver como eles afetam as políticas e ações do agente. Observe que o atraso da etapa é um parâmetro da simulação, enquanto a taxa de aprendizado e o épsilon são parâmetros de seu algoritmo de aprendizado e o fator de desconto é uma propriedade do ambiente.

QUESTÃO 8 (1 PONTO) – REVISITANDO A TRAVESSIA DE PONTE

Primeiro, treine um agente de q-learning completamente aleatório com a taxa de aprendizagem default no BridgeGrid (sem ruído) por 50 episódios e observe se ele encontra a política ótima.

```
python gridworld.py -a q -k 50 -n 0 -g BridgeGrid -e 1
```

Agora tente o mesmo experimento com o epsilon igual a 0. Existe algum epsilon e taxa de aprendizagem para os quais é altamente provável (chance maior que 99%) que a política ótima será aprendida depois de 50 iterações? Você deve retornar na definição da função `question8()` OU uma tupla (de dois valores) na forma `(epsilon, learning`

`rate)` OU a string `'NOT POSSIBLE'`. O epsilon é controlado pelo parâmetro `-e` e a taxa de aprendizagem pelo parâmetro `-l`.

Para testar sua implementação, execute o autograder:

```
python autograder.py -q q8
```

QUESTÃO 9 (1 PONTO) Q-LEARNING E PACMAN

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 Pacman 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 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`). Você receberá pontuação total por esta questão se o comando acima funcionar sem exceções e seu agente vencer pelo menos 80% das vezes. O autograder executará 100 jogos de teste após os 2.000 jogos de treinamento.

Dica: Se o see `QLearningAgent` funnciona 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 considera corretamente em alguns casos ações não vistas. Em particular, porque as ações não vistas têm por definição um Q-valor igual a zero, se todas as ações que foram vistas têm Q-valores negativos, uma ação não vista pode ser ótima. Cuidado com a função argmax do util.Counter!

Para testar sua implementação, execute o autograder:

```
python autograder.py -q q9
```

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.discount` e `self.alpha` dentro do agente.

Nota: Embora um total de 2010 jogos sejam 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 para entender o que está acontecendo, use o comando a seguir:

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

Durante o treinamento, você verá uma saída produzida 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 ocasionalmente fazer um movimento aleatório em direção a um fantasma. Para fins de comparação, deve demorar de 1.000 a 1.400 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).

Se assegure de entender o que está acontecendo aqui: o estado do PDM state é a configuração *exata* do tabuleiro, com a função de transição descrevendo todas as possíveis mudanças daquele estado, considerando simultaneamente tanto o quanto os fantasmas. As configurações intermediárias do jogo nas quais o Pacman se moveu, mas os fantasmas não responderam, não são estados MDP, mas estão agrupados nas transições.

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.

No entanto, você descobrirá que treinar o mesmo agente no aparentemente simples `mediumGrid` não funciona bem. Em nossa implementação, as recompensas médias de treinamento do Pacman permanecem negativas ao longo do treinamento. Na hora do teste, ele joga mal, provavelmente perdendo todos os seus jogos de teste. O treinamento também levará muito tempo, apesar de sua ineficácia.

Pacman não consegue vencer em layouts maiores porque cada configuração de tabuleiro é um estado separado com valores Q separados. Ele não tem como generalizar que encontrar um fantasma é ruim para todas as posições. Obviamente, essa abordagem não terá escala.

QUESTÃO 10 (3 PONTOS) – Q-LEARNING APROXIMADO

Implemente um agente de q-learning aproximado que aprende 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 `[f1(s,a), ..., fi(s,a), ... fn(s,a)]` de características. Funções de características são fornecidas 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.

A função Q aproximada tem a seguinte forma:

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

onde 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. O update dos pesos é feito de forma similar ao update dos q-valores:

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

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

Note que o termo `difference` é o mesmo usado no Q-learning normal.

Por default, o `ApproximateQAgent` usa o `IdentityExtractor`, que atribui uma característica para cada par `(estado,ação)`. Com esse extrator de característica, seu agente 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` em vez de acessar os q-valores diretamente, pra que os novos valores aproximados sejam utilizados.

Quando você estiver seguro de que seu agente está funcionando 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
```

Mesmo tabuleiros maiores podem ser aprendidos com o `ApproximateQAgent` (o comando a seguir pode demorar alguns minutos por conta do treinamento):

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

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

Para testar sua implementação, execute o autograder:

```
python autograder.py -q q10
```

Se você chegou até aqui, parabéns! você agora tem um agente Pacman que aprende com seus erros e acertos!