Trabalho Prático 3: Processos de Decisão de Markov e Aprendizado por Reforço

Prof. Luiz Chaimowicz

2016-11-28

Sumário

Esta especificação foi adaptada dos exercícios de Pac-Man de Berkeley, originalmente criados por John DeNero e Dan Klein.

Introdução

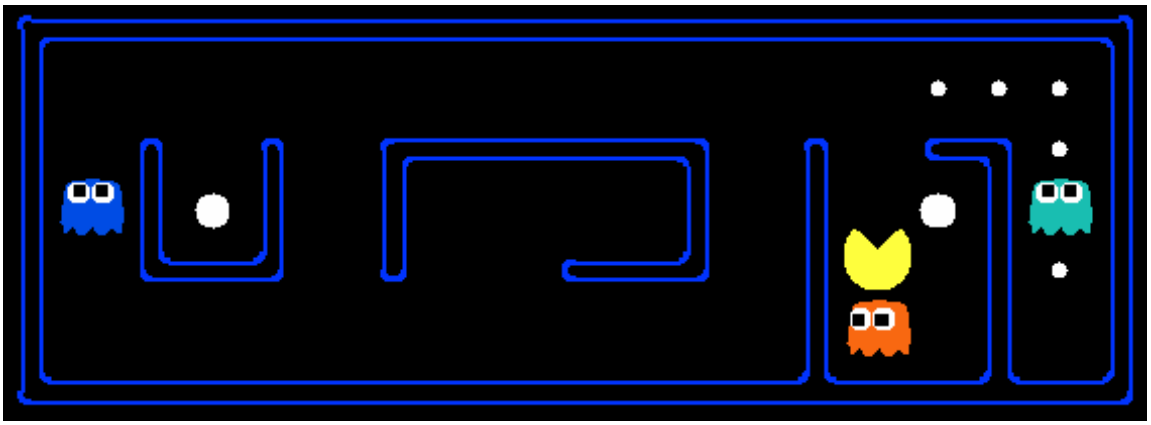

Figura 1: Pac-Man busca recompensas. Ele deve comer ou fugir?

Neste projeto você implementará value iteration e Q-learning. Primeiramente você testará os seus agentes no Gridworld (slides da disciplina, capítulo 17) e então testará em um controlador robótico simulado (Crawler) e em Pac-Man.

Como nos projetos anteriores, este projeto possui um avaliador automático para que você verifique suas respostas e o seu desempenho no projeto. Como no projeto anterior, o autograder é executado com o comando:

```
python autograder.py
```

Note que alguns sistemas operacionais mais recentes podem ter apenas o Python 3 instalado. É possível verificar qual versão está instalada com o comando `python --version`. Não é incomum, quando o padrão for o Python 3, existir um binário `python2` disponível. Se esse for seu caso, substitua os comandos `python` deste documento por `python2`. A instalação do interpretador Python está fora do escopo deste documento.

Se você usa Linux ou outro Unix e o binário `python` apontar para o python3, é possível resolver o problema facilmente no shell criando um alias com:

```
alias python=python2
```

O código deste projeto consiste de vários arquivos Python, alguns dos quais você precisará ler e compreender para completar esse trabalho; outros arquivos você poderá ignorar com segurança. É possível baixar o código do moodle da disciplina.

Para avaliar uma pergunta específica, como a `q2`, use:

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

Também é possível executar um teste em particular com comandos da forma:

```
python autograder.py -t test_cases/q2/1-bridge-grid
```

Por padrão o autograder exibe uma interface gráfica com a opção `-t`, e não a exibe com a opção `-q`. Você pode forçar a presença ou ausência de gráficos com as opções `--graphics` e `--no-graphics`, respectivamente.

Sobre o autograder

Para evitar quaisquer dúvidas, deixaremos explícito: por mais que sua nota não dependa apenas do autograder, ele é fornecido para que você possa se auto-avaiar. Se, por exemplo, o autograder disser que seu código não passou em nenhum teste, não espere que sua nota seja muito maior do que zero. Da mesma forma, se o autograder disser que você acertou todas as questões, a sua nota não será muito menor do que a maior nota possível.

Arquivos que você editará

- `valueIterationAgents.py`: A value iteration agent for solving known MDPs.
- `qlearningAgents.py`: Q-learning agents for Gridworld, Crawler and Pacman.
- `analysis.py`: A file to put your answers to questions given in the project.

Arquivos que você deveria ler, mas NÃO editar

- `mdp.py`: Defines methods on general MDPs.
- `learningAgents.py`: Defines the base classes ValueEstimationAgent and QLearningAgent, which your agents will extend.
- `util.py`: Utilities, including util.Counter, which is particularly‘:useful for Q-learners.
- `gridworld.py`: The Gridworld implementation.
- `featureExtractors.py`: Classes for extracting features on (state,action) pairs. Used for the approximate Q-learning agent (in qlearningAgents.py).

Arquivos que você pode ignorar

- `environment.py`: Abstract class for general reinforcement learning environments. Used by `gridworld.py`.
- `graphicsGridworldDisplay.py`: Gridworld graphical display.
- `graphicsUtils.py`: Graphics utilities.
- `textGridworldDisplay.py`: Plug-in for the Gridworld text interface.
- `crawler.py`: The crawler code and test harness. You will run this but not edit it.
- `graphicsCrawlerDisplay.py`: GUI for the crawler robot.
- `autograder.py`: Project autograder

- `testParser.py`: Parses autograder test and solution files
- `testClasses.py`: General autograding test classes
- `test_cases/` Directory‘:containing the test cases for each question
- `reinforcementTestClasses.py`: Project 3 specific autograding test classes

Mecânica e submissão

Você irá completar partes de `valueIterationAgents.py`, `qlearningAgents.py` e `analysis.py` durante a elaboração deste trabalho. Você deve submeter esses arquivo com seu código e comentários. Por favor *não modifique nem submeta* nenhum outro arquivo desta distribuição.

Por favor veja a seção “Submissão” para saber como submeter seu código. Projetos submetidos fora do padrão serão penalizados. Em outras palavras: use o script `prepare-submission.py`.

Avaliação

Seu código será avaliado automaticamente para verificação de corretude. Por favor *não modifique* o nome de nenhuma função ou classe dentro do código. Note, no entanto, que sua pontuação será baseada também na avaliação do seu código e não somente no julgamento do avaliador automático. Se necessário, revisaremos sua submissão individualmente para garantir que você receba o devido crédito.

Este trabalho consiste de duas partes: uma parte obrigatória e uma parte opcional. A parte obrigatória valerá 10 pontos, com a questão 1 valendo 6 pontos, a questão 2 valendo 1 ponto e a questão 3 valendo 3 pontos. Já a parte opcional valerá 5 pontos, um ponto para cada questão. Logo, a nota máxima possível é 15 pontos.

Honestidade acadêmica

Nós verificaremos as submissões par-a-par em busca de redundância. Se você copiar o código de alguém e submetê-lo com pequenas mudanças, nós saberemos. Os detectores de plágio são difíceis de enganar. Portanto, nem tente. Não nos desaponte, pois confiamos que vocês submeterão somente o fruto de seu trabalho.

Este trabalho é individual e, portanto, compartilhar sua submissão com a de um colega não será aceito.

Obtendo ajuda

Se você tiver alguma dúvida, contate a organização do curso: temos o moodle e possibilidade de agendamento para tirar dúvidas. O objetivo deste projeto é que ele seja compensador e que você aprenda, não frustrante. No entanto, só saberemos como ajudar se você nos perguntar.

Discussão

Ao postar suas dúvidas ou discutir com colegas, tente não dar respostas diretas. O aprendizado é melhor quando entendemos o problema a ponto de elaborar respostas.

Parte obrigatória

Questões nesta parte do documento descrevem componentes obrigatórias para avaliação deste trabalho prático.

Processos de Decisão de Markov (MDPs)

Após baixar, descompactar e mudar para o diretório do código, execute `GrudWorld` em modo de controle manual:

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

Você verá o layout de duas saídas visto em aula. O ponto azul é o agente. Note que quando você pressionar para cima, o agente moverá para norte apenas em 80% do casos. Assim é a vida de um agente do `GridWorld`!

É possível controlar diversos aspectos da simulação. Uma lista completa das opções é disponibilizada com:

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

O agente padrão se move aleatoriamente

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

Você deveria ver o agente aleatório se debater pelo grid até que aconteça dele encontrar uma saída. Não é bem o agente mais inteligente.

Nota: O MDP Gridworld é configurado de forma tal que você deve primeiro entrar no estado pré-terminal (o com borda dupla na interface) e depois tomar a ação 'exit' para que o episódio termine (no verdadeiro estado terminal `TERMINAL_STATE`, não exibido na interface). Se você executar um episódio manualmente, seu retorno total pode ser menor do que o que você esperava devido à taxa de desconto (-d para mudar; 0.9 por padrão).

Veja a saída de texto que acompanha a saída gráfica (ou use o argumento `-t` para uma interface puramente texto). Você verá todas as mudanças de estado do agente (para desabilitar, use `-q`).

Assim como em Pac-Man, posições são representadas por coordenadas cartesianas (x, y) e vetores são indexados como `[x][y]`, com 'norte' sendo a direção em que y é incrementado, etc. Por padrão, a maioria das transições recebe uma recompensa zero, mas você pode mudar isso com a opção `-r`.

Parte 1: Value Iteration

Escreva um agente de value iteration em `ValueIterationAgent`, que foi parcialmente escrito para você em `valueIterationAgents.py`. Seu agente value iteration é um planejador offline, não um agente de aprendizado por reforço e, portanto, a opção de treinamento relevante é o número de iterações que value iteration deveria ser executado (opção `-i`) na sua fase de planejamento inicial. `ValueIterationAgent` recebe um MDP em sua construção e executa value iteration pelo número de iterações especificado antes que o construtor retorne.

Value iteration computa k-passos de estimativas de valores ótimos, V_k. Além de executar value iteration, implemente os seguintes métodos em `ValueIterationAgent` usando V_k.

- `computeActionFromValues(state)` computa a melhor ação de acordo com a função de valor dada por `self.values`.
- `computeQValueFromValues(state, action)` retorna o valor Q do par `(estado, ação)` dada a função de valor em `self.values`.

Importante: Use a versão "batch" de value iteration, onde cada vetor V_k é computado a partir de um vetor fixo V_{k-1}, não a versão "online" onde um único vetor de pesos é atualizado. Isso quer dizer que quando o valor de um estado é atualizado na iteração k baseado no valor de seus estados sucessores. Os valores de estados sucessores usados na computação de update devem ser aqueles da iteração $k-1$ (mesmo que os valores da iteração k já tenham sido computados). A diferença é discutida em Sutton & Barto no sexto parágrafo do capítulo 4.1.

Nota: Uma política sintetizada de valores de profundidade k (que reflete as próximas k recompensas) na verdade refletirá as próximas $k+1$ recompensas (ou seja, você retornará π_{k+1}). Similarmente, os valores Q também refletirão uma recompensa a mais que os valores (ou seja, você retornará Q_{k+1}).

Você deve retornar a política π_{k+1}.

Dica: Use a class `util.Counter` de `util.py`, que é um dicionário com valor padrão de zero. Métodos como `totalCount` devem simplificar seu código. No entanto, tenha cuidado com `argMax`: o argmax que você quer pode não ser uma chave no contador.

Nota: Certifique-se de tratar o caso quando um estado não tiver ações disponíveis em um MDP (pense no que isso pode significar para recompensas futuras).

Para testar sua implementação, execute o autograder:

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

O comando a seguir carrega o seu `ValueIterationAgent`, que computará e executará uma mesma política 10 vezes. Pressione uma tecla para chavear entre valores, valores Q e a simulação. Você deveria ver que o valor do estado inicial (`V(start)`, que você pode ler na tela) e a recompensa média resultante (impressa assim que os 10 rounds de execução terminarem) são bem similares.

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

Dica: No `BookGrid` padrão, executar value iteration por 5 iterações deveria exibir a saída abaixo:

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

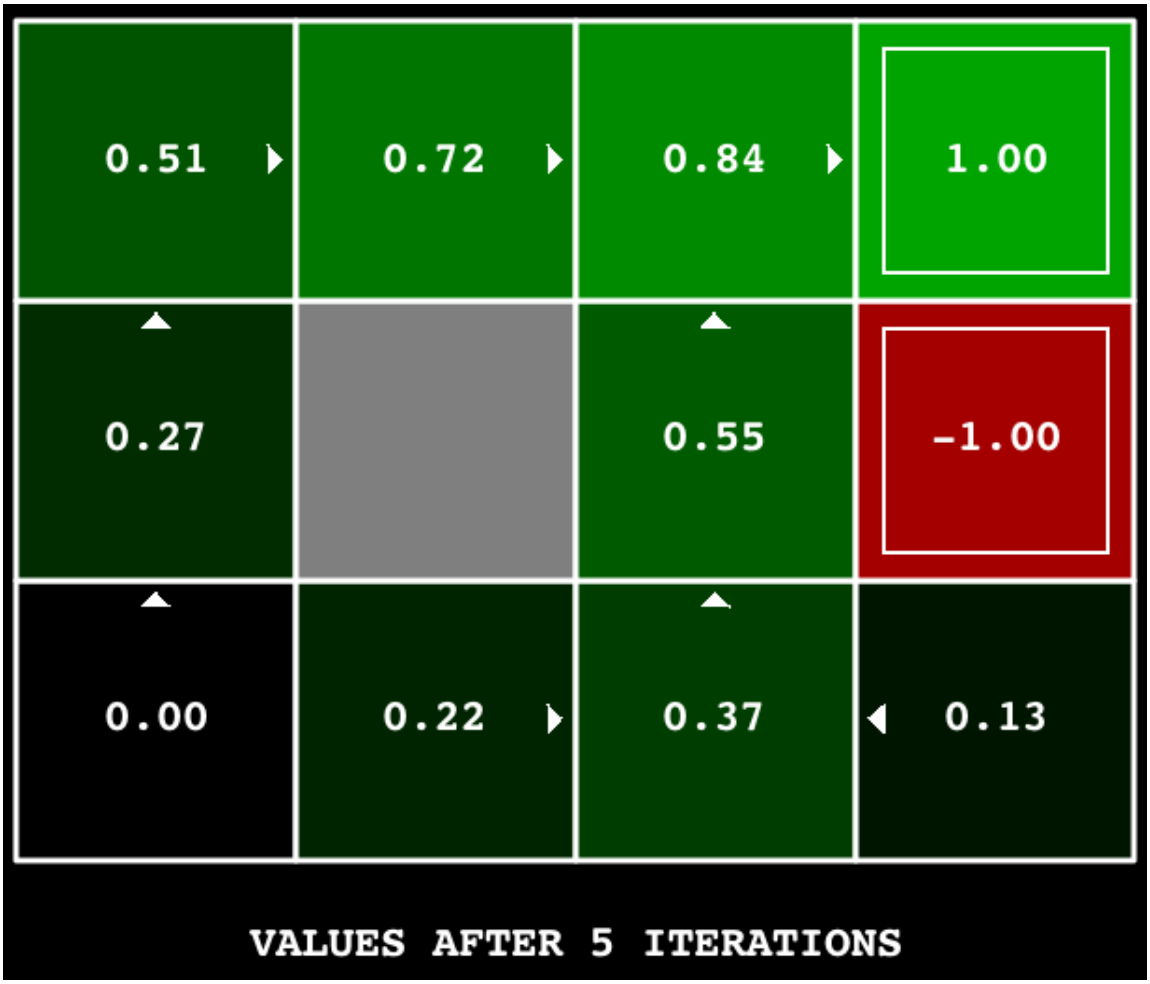

Figura 2: Valores após 5 iterações

Pontuação: Seu agente de value iteration será avaliado em um novo grid. Nós verificaremos seus valores, Q, e políticas após números fixos de iterações e na convergência.

Parte 2: Análise da travessia de ponte

`BridgeGrid` é um mapa de grid em que o estado terminal de recompensa baixa e o estado terminal de recompensa alta estão separados por uma ponte estreita e fora da ponte há abismos de recompensa negativa. O agente iniciará próximo ao estado de baixa recompensa. Com o desconto padrão

de 0.9 e o ruído padrão de 0.2, a política padrão não atravessa a ponte. Mude apenas **UM** dos dois parâmetros de modo que a política ótima faça o agente tentar cruzar a ponte. Coloque sua resposta na função `question2()` de `analysis.py`. Ruído se refere a quão frequentemente o agente para sem querer em um estado sucessor quando ele executa uma ação. O padrão corresponde a:

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

Avaliação: Verificaremos se você mudou apenas um dos parâmetros e, dada essa mudança, que um agente correto de value iteration será capaz de cruzar a ponte.

Para verificar sua resposta, execute o autograder:

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

Figura 3: Valores após 100 iterações

Parte 3: Políticas

Considere o layout `DiscountGrid`, exibido abaixo. Esse grid possui dois estados terminais com payoff positivo (na fileira do meio), uma saída próxima com payoff +1 e uma saída distante com payoff +10. A fileira de baixo do grid consiste de estados terminais com payoff negativo, exibido em vermelho. Cada estado nesta região de penhasco possui payoff -10. O estado inicial é o quadrado amarelo. Distinguimos dentre dois tipos de caminhos: (1) caminhos que arriscam a viagem próxima à fileira de baixo do grid; esses caminhos são curtos, mas arriscam receber um grande payoff

negativo e são representados pela seta vermelha na figura. (2) Caminhos que são mais longos, mas que possuem menos chance de obter grandes payoffs negativos. Esses caminhos são representados pela seta verde na figura.

Nesta parte você escolherá configurações dos parâmetros de desconto (*discount*), ruído (*noise*) e recompensa (*living reward*) neste MDP para produzir políticas ótimas de vários tipos diferentes. Suas configurações dos parâmetros para cada parte devem possuir a propriedade de, se o agente segui-la sem qualquer ruído, ele deveria exibir o comportamento esperado. Se qualquer comportamento não pode ser obtido por nenhuma configuração de parâmetros, responda que a política é impossível retornando a string `NOT POSSIBLE`.

As políticas que você deve tentar produzir são:

1. Preferir a saída próxima (+1), arriscando cair no penhasco (-10)
2. Preferir a saída próxima (+1), evitando o penhasco (-10)
3. Preferir a saída distante (+1), arriscando cair no penhasco (-10)
4. Preferir a saída distante (+1), evitando o penhasco (-10)
5. Evitar tanto as saídas quanto penhasco (logo, o episódio não deveria terminar nunca)

Para verificar suas resposta, rode o autograder:

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

As funções `question3a()` até `question3e()` devem retornar uma tupla de três elementos `(discount, noise, living reward)` em `analysis.py`.

Nota: Você pode verificar suas políticas na interface. Por exemplo, usando uma resposta correta para 3(a), a seta em `(0,1)` deveria apontar para leste, a seta em `(1,1)` também deveria apontar para leste e a seta em `(2,1)` deveria apontar para o norte.

Nota: Em alguns computadores não é possível ver uma seta. Nesse caso, pressione uma tecla no teclado para mudar para o modo de exibição `qValue` e calcule mentalmente a política tomando o argmax dos valores `qValue` disponíveis para cada estado.

Avaliação: Verificaremos se em cada caso a política especificada é retornada.

Parte opcional

Todas as seções abaixo se referem a partes opcionais do trabalho. Se você as fizer, ganhará pontos extra.

Part 4: Q-Learning

Observe que o agente de value iteration não aprende realmente com a experiência. Em vez disso, pondera seu modelo MDP para chegar a uma política completa antes interagindo com um ambiente

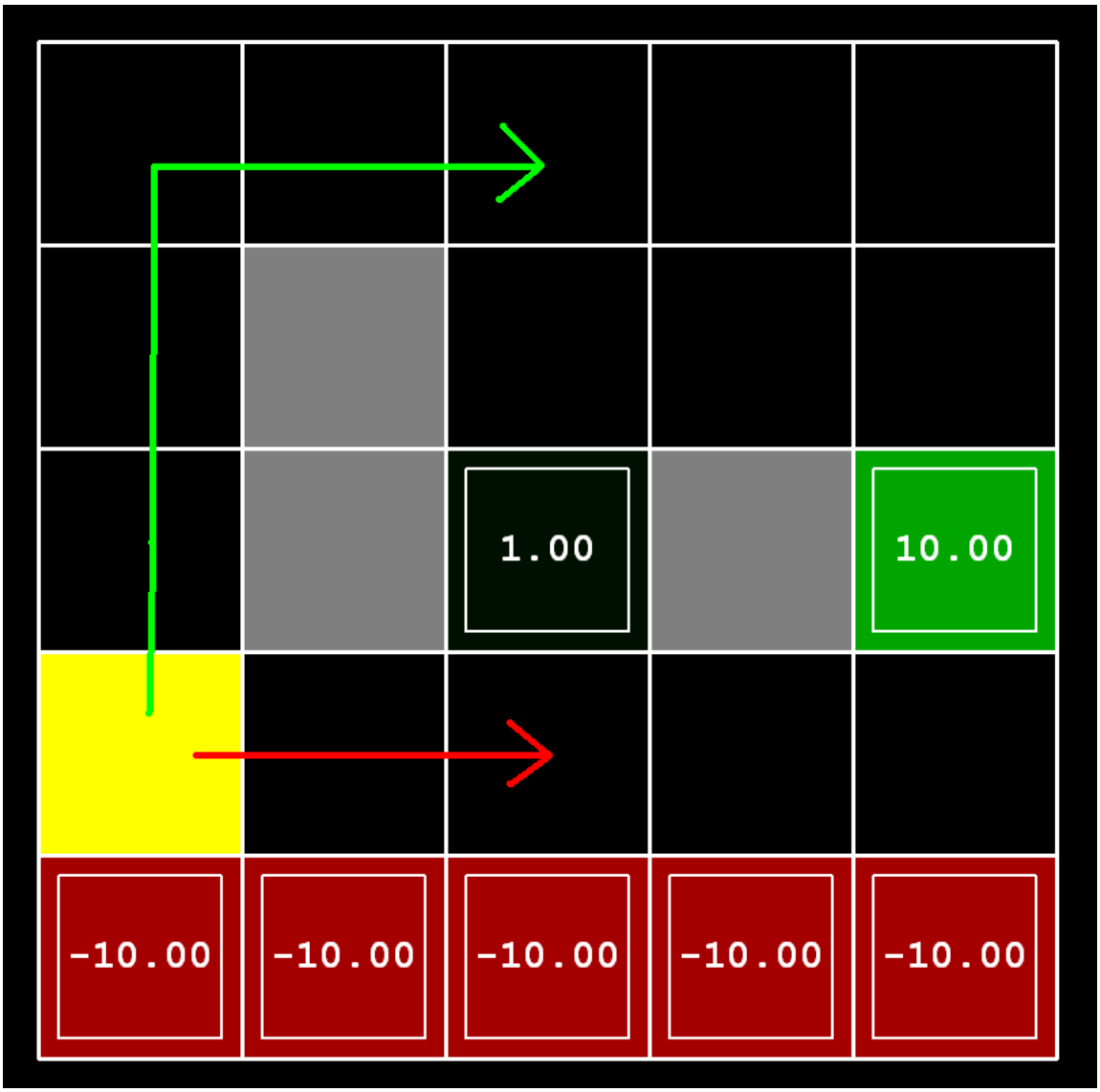

Figura 4: Values after 100 iterations

real. Quando interage com o ambiente, ele simplesmente segue a política pré-computada (por exemplo, torna-se um agente reflex). Esta distinção pode ser sutil em um ambiente simulado como um `Gridword`, mas é muito importante no mundo real, onde o MDP real não está disponível.

Você vai agora escrever um agente Q-learning, que faz muito pouco no construtor mas aprende por tentativa e erro em interações com o ambiente através do seu método `update(state, action, nextState, reward)`. Um pedaço de um Q-learner é especificado em `QLearningAgent` em `qlearningAgents.py` e você pode selecioná-lo com a opção `-a q`. Para esta parte, você deve implementar os métodos `update`, `computeValueFromQValues`, `getQValue` e `computeActionFromQValues`.

Nota: Para `computeActionFromQValues`, você deve resolver empates aleatoriamente para melhor comportamento. A função `random.choice()` te ajudará. Em um estado particular, ações que seu agente ainda não viu antes ainda tem um Q-value de zero e todas as ações que o seu agente já viu tem valor Q negativo.

Importante: Certifique-se de que nas suas funções `computeValueFromQValues` e `computeActionFromQValues`, você só acessa os valores Q chamando `GetQValue`. Esta abstração será útil para a pergunta 8 quando você sobrescrever `GetQValue` para usar características de pares de ação de estado em vez de pares de ação de estado diretamente.

Com a atualização Q-learning no lugar, você pode assistir o seu Q-learner aprender sob Controle manual, usando o teclado:

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

Lembre-se de que `-k` controlará o número de episódios que seu agente recebe para aprender. Veja como o agente aprende sobre o estado em que estava, e não o que ele se move, e "deixa o aprendizado em seu rastro". 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 manualmente guiar o Pac-Man norte e, em seguida, leste ao longo do caminho ideal para quatro episódios, você deve ver os seguintes Q-valores:

Avaliação: Executaremos seu agente Q-learning e verificaremos se ele aprende os mesmos Q-values e política que nossa implementação de referência quando apresentada ao mesmo conjunto de exemplos.

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

Parte 5: Epsilon Greedy

Complete seu agente de Q-learning, implementando a seleção de ação epsilon-greedy em getAction, o que significa que ele escolhe ações aleatórias uma fração epsilon do tempo e segue seus melhores valores Q atuais caso contrário. Observe que a escolha de uma ação aleatória pode resultar na escolha da melhor ação - ou seja, você não deve escolher uma ação sub-ótima aleatória, mas sim qualquer ação legal aleatória.

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

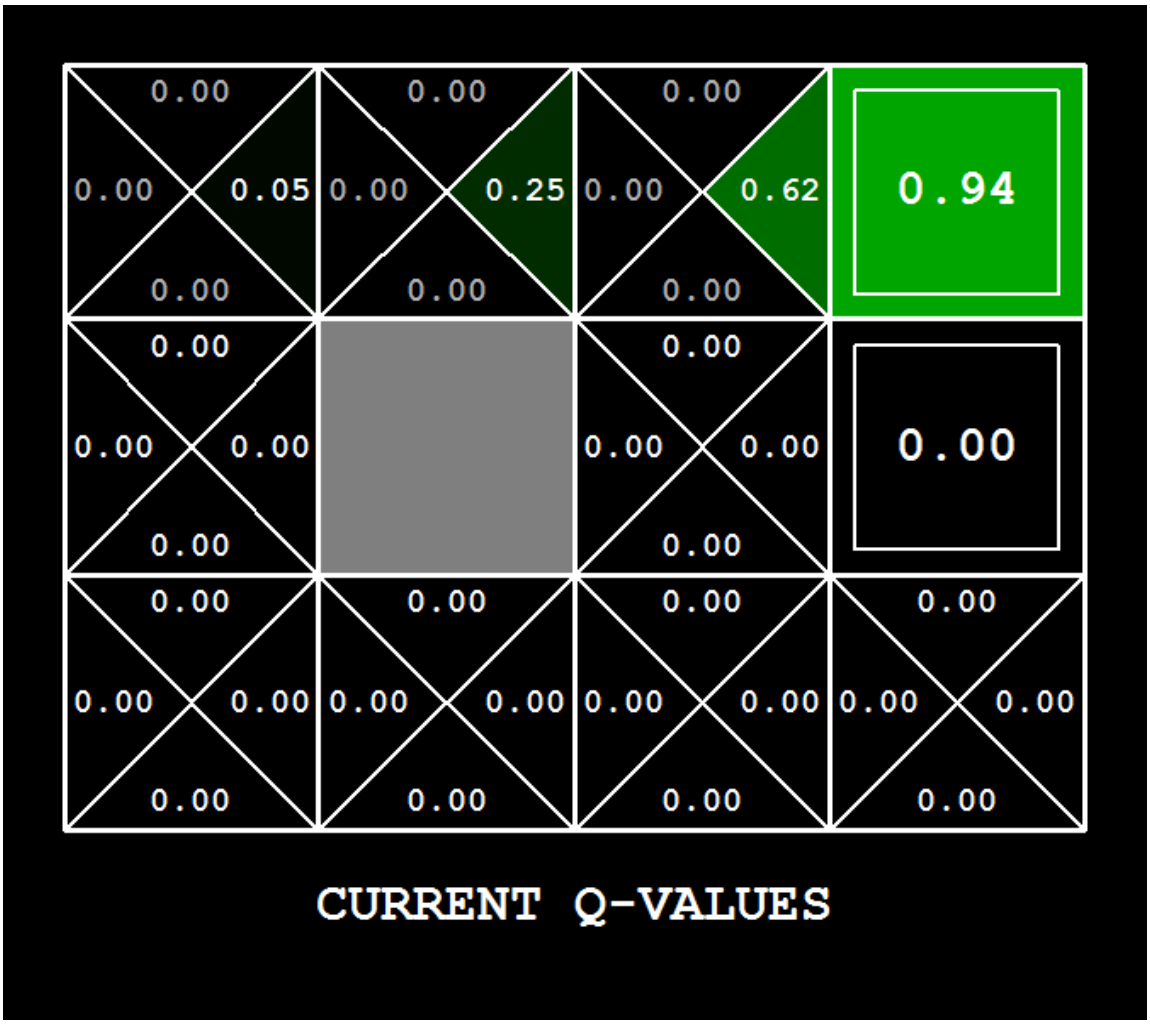

Figura 5: Q-Values

Seus valores Q finais devem se assemelhar aos de seu agente de value iteration especialmente ao longo de caminhos bem percorridos. No entanto, seus retornos médios serão menores do que o que os valores Q predizem por causa das ações aleatórias e da fase inicial de aprendizado.

Você pode escolher um elemento de uma lista uniformemente aleatoriamente 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`.

Para testar sua implementação, execute o autograder:

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

Sem nenhum código adicional, você deveria ser capa de executar o robô Q-learning:

```
python crawler.py
```

Se isso não funcionar, você provavelmente escreveu algum código muito específico para o problema `GridWorld` e você deve torná-lo mais geral para todos os MDPs.

Isso chamará o robô de rastreamento usando seu Q-learner. Brinque com os vários parâmetros de aprendizagem para ver como eles afetam as políticas e ações do agente. Observe que o atraso de passo é um parâmetro da simulação, enquanto que a taxa de aprendizado e epsilon são parâmetros de seu algoritmo de aprendizado e o fator de desconto é uma propriedade do ambiente.

Parte 6: Travessia de ponte revisitada

Primeiro, treine um Q-learner totalmente aleatório com a taxa de aprendizado padrão 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 epsilon zero. Existem epsilon e taxa de aprendizado em que é altamente provável (> 99%) que a política ótima será aprendida após 50 iterações? A função `question6()` em `analysis.py` deve retornar ou uma tupla de dois elementos `(epsilon, taxa de aprendizado)` ou a string `NOT POSSIBLE` se não houver nenhuma. Epsilon é controlado com o argumento `-e` e a taxa de aprendizado por `-l`.

Nota: Sua resposta não deveria depender no mecanismo de desempate usado para escolher ações. Isso quer dizer que sua resposta deve ser correta mesmo se o mundo da ponte for rotacionado de 90 graus.

Para avaliar sua resposta:

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

Parte 7: Q-Learning e Pacman

Hora de jogar Pacman! Pacman jogará em duas fases. Na primeira, trainamento, Pacman começará a aprender os valores de posições e ações. Como leva muito tempo para aprender Q-values corretos mesmo para grids pequenos, os jogos de treinamento rodarão em modo silencioso por padrão, sem interface ou console para exibir. Uma vez que o treinamento estiver completo, ele entrará no modo de teste. Durante o teste, `self.epsilon` e `self.alpha` serão `0.0`, parando o Q-learning e desabilitando exploração, de modo a permitir que Pacman se beneficie da política aprendida. Jogos de teste são exibidos na interface gráfica por padrão. Sem mudanças em seu código você deveria ser capaz de executar o Pacman de Q-learning para grids bem pequenos com:

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

Note que `PacmanQAgent` já foi definido para você em termos `QLearningAgent` que você já escreveu. A única diferença do `PacmanQAgent` é que os parâmetros de aprendizado padrão são mais eficientes para o problema de Pacman `(epsilon=0.05, alpha=0.2, gamma=0.8)`.

Dica: Se seu `QLearningAgent` funciona para `gridworld.py` e `crawler.py` mas não aprende uma boa política para Pacman no `smallGrid`, pode ser devido ao fato de seus métodos `getAction` e/ou `computeActionFromQValues` não consideram de forma apropriada ações que nunca foram vistas. Em particular, como ações não vistas têm, por definição, Q-value zero, o fato de todas as ações já vistas terem Q-value negativo, uma ação não vista será ótima. Cuidado com a função `argmax` de `util.Counter`!

Para verificar sua solução:

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

Nota: Se você quiser experimentar com parâmetros de aprendizado, você pode usar a opção `-a`. Por exemplo: `-a epsilon=0.1,alpha=0.3,gamma=0.7`. Esses valores então ficarão disponíveis como `self.epsilon`, `self.gamma` e `self.alpha` para o agente.

Nota: Apesar de 2010 jogos serem jogados, os primeiros 2000 não serão exibidos, já que a opção `-x 2000` é passada e determina que os primeiros 2000 jogos devem ser usados para treinamento (e sem saída). Logo, você só verá Pacman jogar os últimos 10 desses jogos. O número de jogos de treinamento é passado para o seu agente pela opção `numTraining`.

Nota: Se quiser assistir 10 jogos para ver o que está acontecendo, use o comando:

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

Durante o treinamento, você verá a saída a cada 100 jogos com estatísticas sobre como o Pacman está indo. Epsilon é positivo durante o treinamento, então Pacman vai jogar mal mesmo depois de ter aprendido uma boa política: isso é porque ele ocasionalmente faz um movimento exploratório aleatório em um fantasma. Como referência, deve levar entre 1.000 e 1.400 jogos antes das recompensas Pacman para um segmento de 100 episódios torna-se positivo, refletindo que ele começou a

ganhar mais do que perder. Ao final do treinamento, ele deve permanecer positivo e ser bastante alto (entre 100 e 350).

Certifique-se de que compreende o que está acontecendo aqui: o estado MDP é a configuração exata do tabuleiro, com as transições agora complexas descrevendo uma camada inteira de mudança para esse estado. As configurações intermediárias em que o Pacman se moveu mas os fantasmas não não são estados do MDP, mas são empacotados nas transições.

Uma vez que Pacman é treinado, ele deve ganhar consistentemente em jogos de teste (pelo menos 90% do tempo), já que agora ele está explorando sua política aprendida.

No entanto, você vai achar que treinar o mesmo agente no aparentemente simples middleGrid não funciona bem. Em nossa implementação, as recompensas médias de treinamento de Pacman permanecem negativas ao longo do treinamento. Na época 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 ganhar em layouts maiores porque cada configuração do tabuleiro é um estado separado com Q-values separados. Ele não tem como generalizar que correr para um fantasma é ruim para todas as posições. Obviamente, esta abordagem não é eficiente.

Parte 8: Q-Learning aproximado

Implemente um agente de Q-learning aproximado que aprende pesos para características (features) de estados, onde muitos estados podem compartilhar as mesmas features. Escreva sua implementação na classe `ApproximateQAgent` em `qlearningAgents.py`, que é uma subclasse de `PacmanQAgent`.

Nota: Q-learning aproximado supõe a existência de uma função de feature `f(s,a)` sobre os pares estado-ação que retorna um vetor $f_1(s,a),\ldots,f_i(s,a),\ldots f_n(s,a)$ de valores de features. Nós providenciamos funções de features para você em `featureExtractors.py`. Vetores de features são objetos `util.Counter` (parecidos com dicionários) contendo pares não-zero de features e valores. Todas as features omitidas têm valor zero.

A função Q aproximada possui a forma

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

onde cada peso w_i está associado a uma feature $f_i(s,a)$. No seu código, você deve implementar o vetor de peso como um dicionário mapeando features (que os extratores de features retornarão) para pesos. Você atualizará seus vetores de pesos de forma similar a como você atualizou os Q-values:

$$w_i \leftarrow w_i + \alpha \cdot \text{diferena} \cdot f_i(s,a)$$
$$\text{diferena} = (r + \gamma \max_{a'} Q(s',a')) - Q(s,a)$$

Note que o termo de diferença é o mesmo usado em Q-learning e r é a recompensa.

Por padrão, `ApproximateQAgent` usa `IdentityExtractor`, que atribui uma única feature para todo par `(state, action)`. Com esse extrator de features, seu agente de Q-learning aproximado deveria funcionar de forma idêntica ao `PacmanQAgent`. Você pode testar com o comando:

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

Importante: `ApproximateQAgent` é uma subclasse de `QLearningAgent` e, portanto, compartilha diversos métodos, como `getAction`. Certifique-se de que seus métodos em `QlearningAgent` chamam `getQValue` ao invés de acessar os `Q-values` diretamente, de modo que quando você sobrecarregar `getQValue` em seu agente aproximado, os novos q-values aproximados sejam usados para computar ações.

Uma vez que você estiver confiante de que o seu agente funciona corretamente com as features, execute-o com o extrator de features, que deveria ser capaz de aprender facilmente como ganhar:

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

Até layouts muito maiores deveriam ser moleza para o seu `ApproximateQAgent`, mas deve levar alguns minutos para treinar.

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

Se você não encontrar erros, seu agente de Q-learning aproximado deveria vencer a maioria dos jogos com essas features simples, mesmo que seja treinado com apenas 50 jogos.

Avaliação: Executaremos seu agente Q-learning e verificaremos se ele aprende os mesmos Q-values e política que nossa implementação de referência quando apresentada ao mesmo conjunto de exemplos.

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

Parabéns! Você tem um agente de Pacman capaz de aprender. `:-)`

Submissão

Você deve submeter sua implementação pelo moodle da disciplina. Para submissão execute o script `prepare-submission.py`. Ele coletará os arquivos que devem ser submetidos e gerará um arquivo zip para submissão no moodle. Por favor não tente usar qualquer programa de compressão. Em suma: não tente ser esperto. O script foi escrito para garantir consistência na submissão.

```
python prepare-submission.py
```

O script perguntará por seu número de matrícula e gerará um arquivo cujo nome consistirá de seu número de matrícula com a extensão `.zip`. Dentro desse arquivo será adicionado o arquivo `multiAgents.py`.