Port para Python 3 do exercício simplificado de Aprendizado por Reforço de Berkeley. O material original, escrito em Python 2, está em:

http://ai.berkeley.edu/reinforcement.html.

Para este trabalho, você deve utilizar o código base disponível neste link.

Introdução

Neste projeto, você implementará **iteração de valor** e **Q-learning**. Você irá testar seus agentes primeiro no Gridworld (da aula) e, em seguida, aplicá-los a um controlador de robô simulado (Crawler) e Pacman.

Este projeto inclui um **autograder** para você verificar suas soluções em sua máquina. Ele pode ser executado em todas as questões com o comando:

```
python autograder.py
```

Ele pode ser executado para uma questão em particular, como a q2, com:

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

Ele pode ser executado para um teste em particular com comandos com a seguinte forma:

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

Consulte o tutorial do autograder para obter mais informações sobre como usar o autograder.

O código para este projeto contém os seguintes arquivos:

Arquivos que você vai editar:

- **valueIterationAgents.py**: um agente de iteração de valor para resolver MDPs conhecidos.
- **qlearningAgents.py**: agentes Q-learning para o Gridworld, Crawler e Pacman.
- **analysis.py**: um arquivo para preencher com suas respostas às perguntas deste projeto.

Arquivos que você deve ler, mas NÃO editar:

- mdp.py Define métodos gerais de MDPs (Markov decision processes).
- learningAgents.py Define as classes base ValueEstimationAgent e QLearningAgent, que seus agentes irão estender.
- util.py Utilitários, incluindo util.Counter, que é particularmente útil para Q-learners.

- gridworld.py A implementação do Gridworld.
- featureExtractors.py Classes para extrair recursos em pares (estado, ação). Usado para o agente Q-learning aproximado (em qlearningAgents.py), para o exercício extra.

Arquivos que você pode ignorar:

- environment.py: Classe abstrata para ambientes gerais de aprendizagem por reforço. Usado por gridworld.py.
- graphicsGridworldDisplay.py: Exibição gráfica do Gridworld.
- graphicsUtils.py: utilitários gráficos.
- textGridworldDisplay.py: Plug-in para a interface de texto Gridworld.
- crawler.py: O código do crawler e o artefatos de teste. Você vai executá-lo, mas não vai editá-lo.
- graphicsCrawlerDisplay.py: GUI para o robô rastreador.
- autograder.py: autograder do projeto.
- testParser.py: Parser de arquivos de solução e testes do autograder
- testClasses.py: Classes de teste gerais de autocorreção (autograding)
- test_cases/: diretório contendo os casos de teste para cada questão
- reforcementTestClasses.py: Classes de teste de autograding específicas do Projeto 3

Arquivos para editar e enviar:

Você preencherá partes de valueIterationAgents.py, qlearningAgents.py e analysis.py durante o exercício. Você deve enviar esses arquivos com seu código e comentários. Não altere os outros arquivos nesta distribuição nem envie qualquer um de nossos arquivos originais que não sejam esses arquivos.

Avaliação:

o autograder será executado no seu código para correção técnica. Não altere os nomes de quaisquer funções ou classes fornecidas dentro do código, ou você causará estragos no autograder. No entanto, a corretude de sua implementação - não os julgamentos do autograder - será o juiz final de sua pontuação. Revisaremos e avaliaremos os envios individualmente para garantir que você receba o devido crédito pelo seu trabalho.

Desonestidade Acadêmica: iremos comparar seu código com outros envios para verificação de plágio. Os detectores de plágio são muito difíceis de enganar, então, por favor, não tente.

Conseguindo ajuda: Você não está sozinho(a)! Se você tiver dificuldades em algo, entre em contato com a equipe do curso para obter ajuda. O canal dúvidas, da equipe da disciplina no teams pode ser utilizada para discussão de dúvidas.

Discussão: Por favor tome cuidado para não postar spoilers (trechos de código com a solução).

MDPs

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

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

Você verá o layout do gridworld trabalhado em aula com duas saídas. O ponto azul é o agente. Observe que quando você pressiona para cima, o agente só se move para o norte 80% do tempo. Assim é a vida de um agente do Gridworld!

Você pode controlar muitos aspectos da simulação. Uma lista completa de opções está disponível executando:

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

O agente default se move aleatoriamente:

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

Você deve ver o agente aleatório se batendo pelo grid até que aconteça de achar uma saída. Essa é uma vida ruim para um agente de IA.

Nota: O MDP do Gridworld é tal que você deve primeiro entrar em um estado pré-terminal (as caixas duplas mostradas na GUI) e então realizar a ação especial de sair ('exit') antes que o episódio realmente termine (no verdadeiro estado do terminal chamado TERMINAL_STATE, que não é mostrado na GUI). Se você executar um episódio manualmente, seu retorno total pode ser menor do que o esperado, devido à taxa de desconto (-d para alterar; 0,9 por padrão).

Observe a saída do console que acompanha a saída gráfica (ou use -t para tudo em texto). Você será informado sobre cada transição que o agente experimenta (para desligar, use -q).

Como no Pacman, as posições são representadas por coordenadas cartesianas (x,y) e quaisquer matrizes são indexadas por [x][y], com 'norte' sendo a direção de aumento de y, etc. Por padrão, a maioria das transições receberá uma

recompensa de zero, embora você possa alterar isso com a opção de recompensa por viver (-r).

Questão 1: Iteração de Valor

Escreva um agente de iteração de valor em `ValueIterationAgent`, que foi parcialmente especificado para você em `valueIterationAgents.py`. Seu agente de iteração de valor é um planejador offline, não um agente de aprendizado por reforço e, portanto, a opção de treinamento relevante é o número de iterações do algoritmo de iteração de valor que ele deve executar (opção -i) em sua fase de planejamento inicial. `ValueIterationAgent` usa um MDP na construtora e executa a iteração de valor para o número especificado de iterações antes de retornar.

A iteração de valor calcula estimativas de k passos dos valores ótimos, V_k. Além de executar a iteração de valor, implemente os seguintes métodos para `ValueIterationAgent` usando V_k.

- `computeActionFromValues(state)` calcula a melhor ação de acordo com a função de valor fornecida por `self.values`.
- `computeQValueFromValues(state, action)` retorna o valor-Q do par (estado, ação) dado pela função de valor em `self.values`. Lembre-se que, uma vez que o valor de cada estado está determinado, e conhecendo as probabilidades de transição (obtidas com getTransitionStatesAndProbs), podemos calcular os valores Q de pares estado-ação.

Essas quantidades são todas exibidas na GUI: os valores são números nos quadrados, os valores-Q são números nos triângulos (um para cada ação) e a política são as setas em cada quadrado.

Importante: Use a versão em "batch" da iteração de valor, onde os valores da iteração k (V_k) é calculado a partir dos valores da iteração k-1 (V_{k-1}), não a versão "online" onde os valores são atualizados "in place". Isso significa que quando o valor de um estado é atualizado na iteração k com base nos valores de seus estados sucessores, os valores dos sucessores usados no cálculo devem ser aqueles da iteração k-1 (mesmo se alguns dos estados sucessores já tivessem sido atualizado na iteração k). A diferença é discutida em Sutton & Barto no 6º parágrafo do capítulo 4.1.

Dica: Use a classe `util.Counter` em `util.py`, que é um dicionário com valor padrão zero. Métodos como `totalCount` devem simplificar seu código. No entanto, tome cuidado com `argMax`

Para testar sua implementação, execute o autograder:

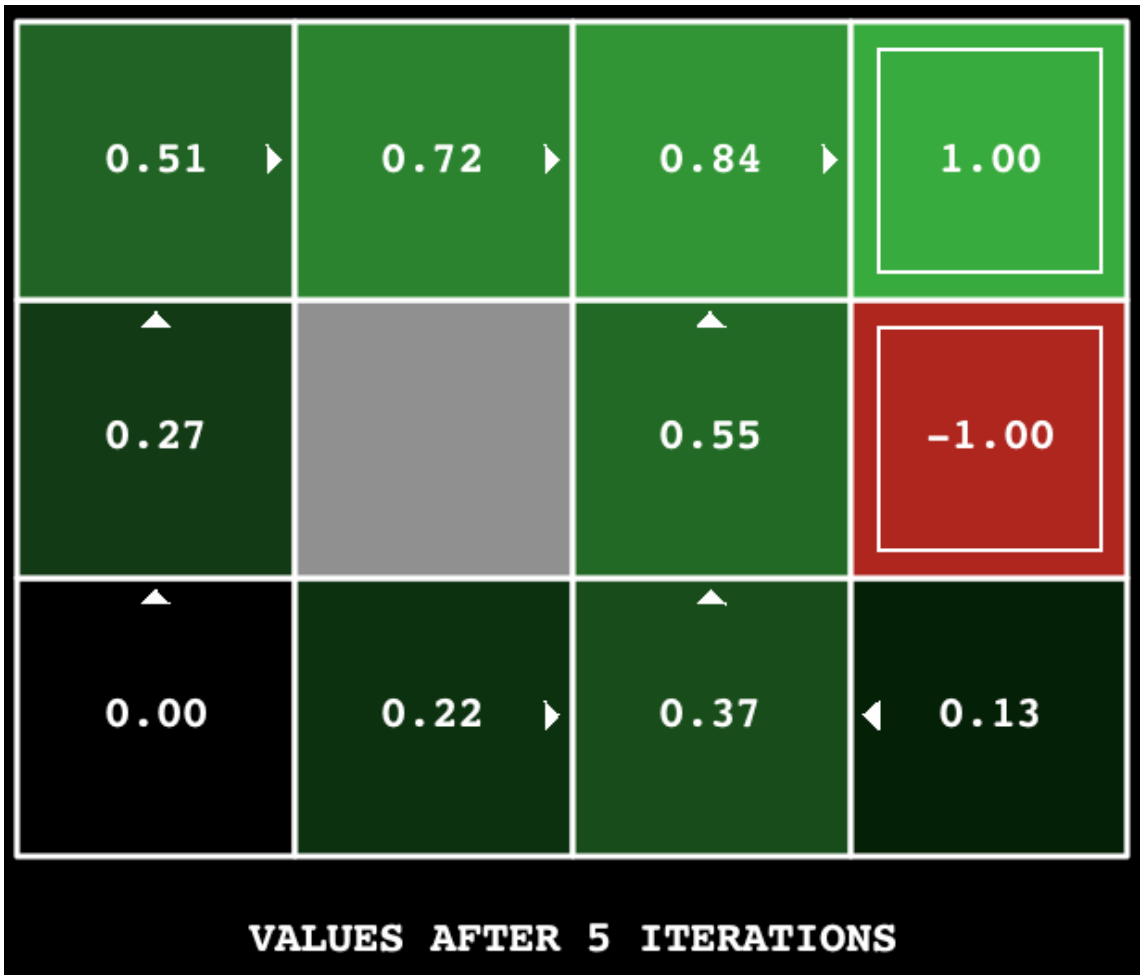
```
python autograder.py -q q1
```

O comando a seguir carrega seu `ValueIterationAgent`, que irá computar uma política e executá-la 10 vezes. Pressione uma tecla para percorrer os valores, valores-Q e a simulação. Você deve descobrir que o valor do estado inicial (V(início), que pode ser lido na GUI) e a recompensa média resultante empírica (impressa após o término das 10 rodadas de execução) são bastante próximos.

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

Dica: No BookGrid padrão, a iteração de valor em execução para 5 iterações deve fornecer esta saída:

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



***Avaliação**:* Seu agente de iteração de valor será avaliado em um grid novo. Verificaremos seus valores, valores Q e políticas após números fixos de iterações e na convergência (por exemplo, após 100 iterações).

Questão 2: Análise da 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 há um abismo de recompensa altamente negativa. O agente começa próximo ao estado de baixa recompensa. Com o desconto padrão de 0,9 e o ruído padrão de 0,2, a política ótima não cruza a ponte. Altere apenas UM dos parâmetros de desconto e ruído para que a política ótima faça com que o agente tente cruzar a ponte. Coloque sua resposta em `question2()` de `analysis.py`. (O ruído se refere à probabilidade com que um agente termina em um estado de sucessor não intencional quando 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 alterou apenas um dos parâmetros fornecidos e, com essa alteração, um agente de iteração de valor correto deve cruzar a ponte. Para verificar sua resposta, execute o autograder:

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

Questão 3: Q-Learning

Observe que seu agente de iteração de valor não aprende realmente com a experiência. Em vez disso, ele considera seu modelo MDP para chegar a uma

política completa antes de interagir com um ambiente real. Quando ele interage com o ambiente, ele simplesmente segue a política pré-computada (e.g. torna-se um agente reflexivo). Essa distinção pode ser sutil em um ambiente simulado como um Gridword, mas é muito importante no mundo real, onde o as probabilidades de transição reais não estão disponíveis.

Agora você escreverá um agente Q-learning, que faz muito pouco na construtora, mas aprende por tentativa e erro a partir de interações com o ambiente por meio de seu método `update (state, action, nextState, recompensa)`. Um esboço de um Q-learner é especificado em `QLearningAgent` em `qlearningAgents.py`, e você pode selecioná-lo com a opção '-a q'. Para esta questão, você deve implementar os métodos `update`, `computeValueFromQValues`, `getQValue` e `computeActionFromQValues`.

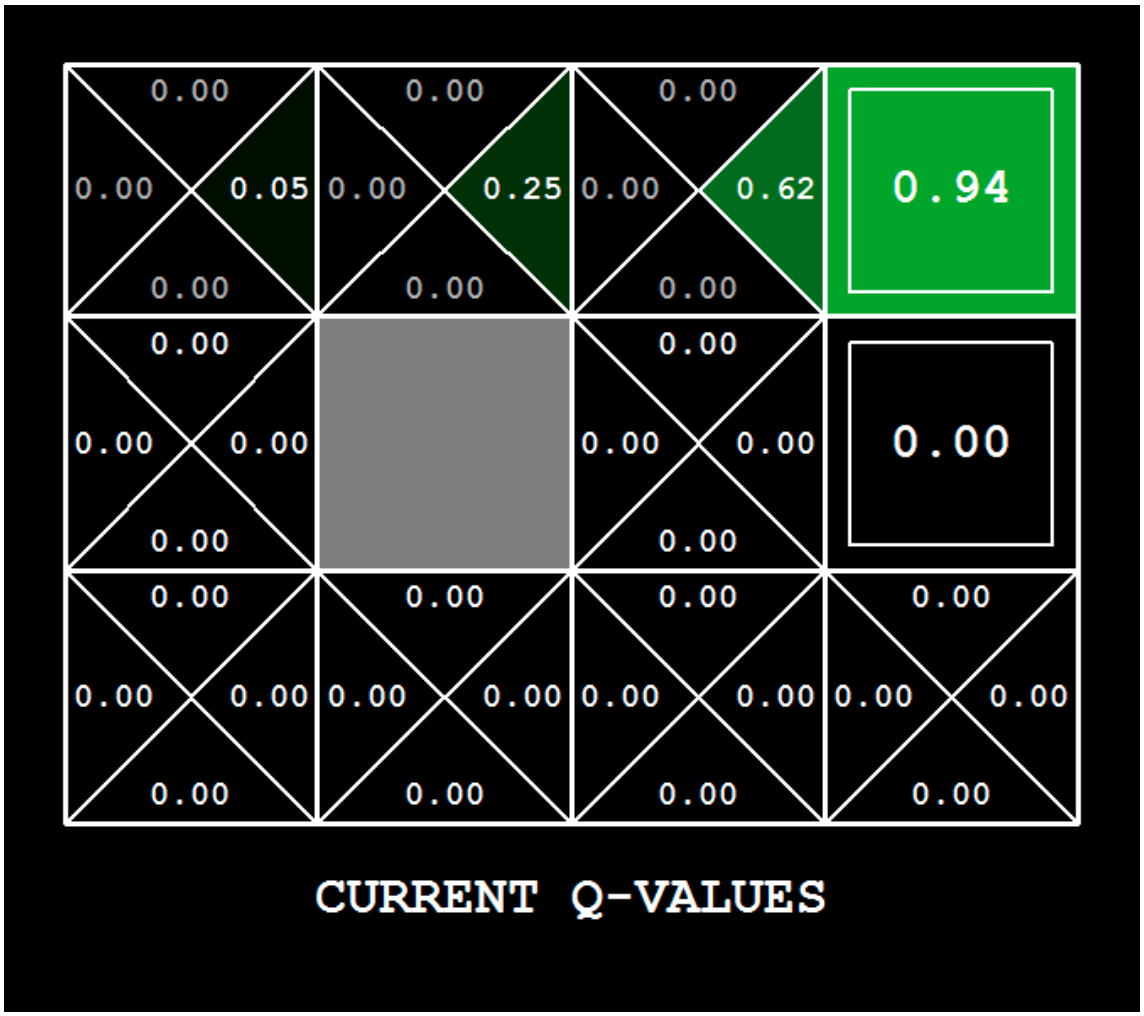
Nota: Para `computeActionFromQValues`, você deve quebrar empates aleatoriamente para um melhor comportamento. A função `random.choice()` ajudará. Em um determinado estado, mesmo as ações que seu agente não viu antes têm um valor-Q, especificamente um valor-Q de zero, e se todas as ações que seu agente viu antes tiverem um valor-Q negativo, a ação não vista pode ser ótima.

Importante: Certifique-se de que em suas funções `computeValueFromQValues` e `computeActionFromQValues`, você só acessa valores-Q chamando `getQValue`. Esta abstração será útil para a questão 6 (extra) quando você sobrescrever `getQValue` para usar features dos pares estado-ação ao invés dos pares estado-ação diretamente.

Com a atualização do Q-learning implementada, você pode assistir ao 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 aprenderá. Observe como o agente aprende sobre o estado em que estava, não aquele para o qual se move, e "deixa o aprendizado por onde passar". 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). Se você direcionar o Pacman manualmente para o norte e depois para o leste ao longo do caminho ótimo para quatro episódios, deverá ver os seguintes valores Q:



Avaliação: executaremos seu agente Q-learning e verificaremos se ele aprende os mesmos valores-Q e política de nossa implementação de referência quando cada um é apresentado com o mesmo conjunto de exemplos. Para avaliar sua implementação, execute o autograder:

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

Questão 4: Epsilon Greedy

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

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

Seus valores-Q finais devem ser semelhantes aos de seu agente de iteração de valor, especialmente ao longo de caminhos bastante percorridos. No entanto, seus retornos médios serão menores do que os previstos pelos valores-Q por causa das ações aleatórias e da fase inicial de aprendizagem.

Você pode escolher um elemento de uma lista de maneira uniformemente aleatória 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 q4
```

Sem nenhum código adicional, agora você deve ser capaz de executar um robô rastejador (crawler) com Q-learning:

```
python crawler.py
```

Se isso não funcionar, você provavelmente escreveu algum código muito específico para o problema `GridWorld` e deve torná-lo mais geral para todos os MDPs. Por exemplo, para determinar o conjunto de ações legais em cada estado, você pode utilizar getLegalActions.

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

Questão 5: Q-Learning e Pacman

É hora de jogar Pacman! O Pacman vai jogar em duas fases. Na primeira fase, *treinamento*, Pacman começará a aprender sobre os valores das posições e ações. Como leva muito tempo para aprender valores-Q precisos, mesmo para grids minúsculos, os jogos de treinamento do Pacman são executados em modo silencioso por padrão, sem display GUI (ou console). Assim que o treinamento de Pacman for concluído, ele entrará no modo de *teste*. Durante o teste, `self.epsilon` e `self.alpha` do Pacman serão ajustados para 0.0, efetivamente interrompendo o Q-learning e desabilitando a exploração, a fim de permitir que Pacman tire proveito de sua política aprendida. Os jogos de teste são mostrados na GUI por padrão. Sem quaisquer alterações de código, você deve ser capaz de executar o Pacman Q-learning para grids muito pequenos, como a seguir:

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

Observe que `PacmanQAgent` já está definido para você em termos do `QLearningAgent` que você já escreveu. `PacmanQAgent` só é diferente por ter parâmetros de aprendizagem padrão que são mais eficazes para o problema Pacman (`epsilon = 0.05, alpha = 0.2, gamma = 0.8`). Você receberá crédito 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 seu `QLearningAgent` funciona para `gridworld.py` e `crawler.py`, mas não parece estar aprendendo uma boa política para Pacman em `smallGrid`, pode ser porque seu `getAction` e / ou os métodos `computeActionFromQValues` não consideram adequadamente, em alguns casos, ações não vistas. Em particular, porque ações não-vistas têm por definição um valor-Q de zero, se todas as ações que *foram* vistas têm valores Q negativos, uma ação não-vista pode ser ótima. Cuidado com a função argmax do `util.Counter`!

Observação: Para avaliar sua resposta, execute:

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

Nota: Se você quiser experimentar os parâmetros de aprendizagem, você pode usar a opção `-a`, por exemplo `-a epsilon=0.1,alpha=0.3,gamma=0.7`. Esses valores ficarão acessíveis como `self.epsilon`, `self.gamma` e `self.alpha` dentro do agente.

Nota: Embora um total de 2010 jogos sejam jogados, os primeiros 2.000 jogos não serão exibidos por causa da opção `-x 2000`, que designa os primeiros 2.000 jogos para treinamento (sem saída). Portanto, você só verá o Pacman jogar os últimos 10 desses jogos. O número de jogos de treinamento também é passado ao seu agente como a opção `numTraining`.

Nota: Se você quiser assistir a 10 jogos de treinamento 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 se saindo. Epsilon é positivo durante o treinamento, então Pacman jogará mal mesmo depois de ter aprendido uma boa política: isso ocorre porque ele ocasionalmente faz um movimento exploratório aleatório pra cima de um fantasma. Como referência, deve demorar entre 1.000 e 1.400 jogos antes que as recompensas de Pacman por um segmento de 100 episódios se tornem

positivas, refletindo que ele começou a ganhar mais do que perder. Ao final do treinamento, deve permanecer positivo e estar razoavelmente alto (entre 100 e 350).

Certifique-se de entender o que está acontecendo aqui: o estado do MDP é a configuração *exata* do mapa para o Pacman, com as transições agora complexas que descrevem um passo inteiro de mudança para esse estado. 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.

Assim que o Pacman terminar de treinar, ele deve ganhar com muita segurança em jogos de teste (pelo menos 90% das vezes), já que agora ele está tirando proveito de sua 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 durante o 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 do mapa é um estado diferente com valores-Q diferentes. Ele não tem como generalizar que encontrar um fantasma é ruim para todas as posições. Obviamente, essa abordagem não vai escalar bem.

Questão 6 (extra): Q-Learning aproximado

Implemente um agente Q-learning aproximado (com aproximação linear de funções) que aprenda pesos para features de estados e ações, 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 características (features) `f(s, a)` sobre pares de estado e ação, que produz um vetor $f_1(s,a)$, ...$f_i(s,a)$,... $f_n(s,a)$ de valores de features, onde $f_i(s,a)$ representa a i-ésima feature do par estado-ação (s,a). Note que esta forma de representar features é uma variação da forma vista em aula, em que extraímos features apenas dos estados. Aqui o conjunto de features é mais rico, representando características da situação do

estado, considerando a realização da ação. Mas a ideia é essencialmente a mesma vista em aula.

Fornecemos funções de features para você em `featureExtractors.py`. Os vetores de features são objetos `util.Counter` (como um dicionário) contendo os pares de features e valores diferentes de zero; todas as features omitidas têm valor zero.
A função-Q aproximada tem a seguinte forma:

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

onde cada peso w_i está associado a uma feature particular f_i(s, a). Em seu código, você deve implementar o vetor de pesos com um dicionário mapeando features (que os extratores de feature retornarão) para valores de peso. Você atualizará seus vetores de peso de maneira similar à qual atualizou os valores Q:

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

$$w_i \leftarrow w_i + \alpha \cdot \delta \cdot f_i(s,a)$$

Observe que o termo δ é uma parte da regra de atualização do Q-learning normal e r é a recompensa experimentada.

Por padrão, `ApproximateQAgent` usa o `IdentityExtractor`, que atribui uma feature única para cada par `(estado, ação)`. Com este extrator de features, 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` e, portanto, compartilha vários métodos como `getAction`. Certifique-se de que seus métodos em `QLearningAgent` chamam `getQValue` em vez de acessar os valores-Q diretamente, de modo que quando você substituir `getQValue` em seu agente aproximado, os novos valores-q aproximados sejam usados para calcular ações.
Quando você tiver certeza de que seu aprendiz aproximado funciona corretamente com os features-identidade, execute seu agente Q-learning aproximado com nosso extrator de recurso personalizado, que pode aprender a vencer com facilidade:

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

Mesmo layouts muito maiores não devem ser problema para o seu `ApproximateQAgent`. (*aviso*: isso pode levar alguns minutos para treinar).

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

Se você não tiver erros, seu agente Q-learning aproximado deve ganhar quase todas as vezes com essas features simples, mesmo com apenas 50 jogos de treinamento.

Avaliação: Executaremos seu agente Q-learning aproximado e verificaremos se ele aprende os mesmos valores-Q e pesos de features de nossa implementação de referência quando cada um é apresentado com o mesmo conjunto de exemplos. Para avaliar sua implementação, execute o autograder:

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

Parabéns! Você tem um agente que aprende Pacman!

Avaliação geral

Item	Percentual da nota
Questão 1: Iteração de Valor	30
Questão 2: Análise da Travessia de Ponte	5
Questão 3: Q-Learning	40
Questão 4: Epsilon Greedy	10
Questão 5: Q-Learning e Pacman	10
Estrutura de diretórios correta? Informações sobre o grupo no group.md completas?	5
Questão 6 (extra): Q-Learning aproximado	10

A realização da tarefa extra pode compensar eventuais descontos em outras questões.

Entrega

Você deverá enviar no moodle um arquivo .zip contendo 4 arquivos em sua raiz:

1. valueIterationAgents.py, com sua implementação do algoritmo de Iteração de Valor;

2. qlearningAgents.py, com sua implementação do Q-learning;

3. analysis.py, com respostas às questões pedidas;

4. group.md, um arquivo de texto sem formatação ou Markdown com os nomes, cartões de matrícula e turma dos integrantes do grupo (não é um Readme.md como nos outros trabalhos para evitar confusão com o Readme.md do repositório com o kit do trabalho).

Note que os 3 primeiros arquivos estão no kit do trabalho. Você deverá preenchê-los, rodar o autograder pra ver se sua implementação está correta, juntá-los com seu group.md no seu .zip e enviar esse .zip no link de entrega do moodle.

IMPORTANTE: todos os 4 arquivos devem estar na raiz do seu .zip!

Observações gerais

O trabalho deve ser feito em grupos.

● Fiquem atentos à política de plágio!

● É a corretude da sua implementação - e não o julgamento do verificador automatizado

(autograder) - que dará o veredito final da sua nota. Revisaremos os envios manualmente para garantir que seu grupo receba a pontuação apropriada para o trabalho.

Porém, note que é importante não haver erros provocados por pela não aderência à especificação do trabalho (e.g. coisas que “quebram” o autograder, como trocar o tipo dos argumentos ou do retorno das funções). Esse tipo de erro é considerado como implementação incorreta.