Projeto 1 – Aprendizagem por Reforço[1]

Agentes Inteligentes

Índice

Introdução

Neste projeto você irá implementar iteração de valor e Q-learning. Você irá testar seus agentes primeiramente no Gridworld (visto em aula), e então aplica-los-á ao robô rastejante (Crawler) e o Pacman.

Este projeto inclui um autoavaliador (*autograder*) para que você possa avaliar suas soluções na sua própria máquina. Você pode rodá-lo em todas as questões com o comando:

```
python autograder.py
```

Ele também pode ser executado para uma questão em particular, tal como q2, por:

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

Pode-se executar um caso de teste específico com comandos do tipo:

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

O código para esse projeto contém os arquivos seguintes, disponíveis em um arquivo zip:

Arquivos que você irá editar:

- `valueIterationAgents.py` — Um agente que implementa iteração de valor para resolver MDPs conhecidos.
- `qlearningAgents.py` — Agentes de Q-learning para o Gridworld, Crawler e Pacman.
- `analysis.py` — Um arquivo para colocar suas respostas para perguntas feitas no projeto.

1 Este projeto é adaptado do “Project 3: Reinforcement Learning”, disponível em http://ai.berkeley.edu/reinforcement.html

Arquivos que você deve ler, mas NÃO deve editar:

- `mdp.py` — Define métodos gerais para MDPs.
- `learningAgents.py` — Define as classes base `ValueEstimationAgent` e `QlearningAgent`, que seus agentes irão estender.
- `util.py` — Utilitários, tais como `util.Counter`, que é particularmente útil para Q-learners.
- `gridworld.py` — A implementação do Gridworld.
- `featureExtractors.py` — Classes para extrair features de pares (estado, ação). Usado pelo agente de Q-learning aproximado (em `qlearningAgents.py`).

Arquivos que você pode ignorar:

- `environment.py` — Classe abstrata para ambientes de aprendizagem por reforço genéricos. Usado por `gridworld.py`.
- `graphicsGridworldDisplay.py` — Interface gráfica do Gridworld.
- `graphicsUtils.py` — Utilitários gráficos.
- `textGridworldDisplay.py` — Plug-in para a interface textual do Gridworld.
- `crawler.py` — O código do crawler e infraestrutura de teste. Você irá rodá-lo, mas não editá-lo.
- `graphicsCrawlerDisplay.py` — GUI para o robô rastejante.
- `autograder.py` — Autoavaliador do projeto.
- `testParser.py` — Processa os arquivos de teste e de solução para o autoavaliador.
- `testClasses.py` — Classes de teste gerais do autoavaliador.
- `test_cases/` — Diretório contendo os casos de teste para cada questão.
- `reinforcementTestClasses.py` — Classes de teste de autoavaliação específicas para este projeto.

Arquivos para Editar e Submeter: Você preencherá partes de `valueIterationAgents.py`, `qlearningAgents.py` e `analysis.py` durante esse projeto. Você deve enviar apenas esses três arquivos de código, contendo seu código e comentários. Por favor, não envie qualquer outro arquivo de código além desses. Esses três arquivos por você enviados substituirão os originais da distribuição .zip na execução do seu projeto para avaliação. Você deverá entretanto acrescentar ao seu .zip submetido slides de apresentação do seu projeto, em formato PDF (ver item “Entrega e Apresentação” ao final).

Avaliação: seu código será autoavaliado para correção técnica. Por favor, não altere os nomes de quaisquer funções ou classes fornecidas com o código, ou você vai causar estragos no autoavaliador. No entanto, a correção de sua implementação – e não os julgamentos do autoavaliador - será o juiz final de sua pontuação. Se necessário, revisaremos e daremos notas individuais para garantir que você receba o devido crédito pelo seu trabalho.

Desonestidade Acadêmica: Verificamos seu código em relação a outras submissões, bem como a soluções disponíveis na internet, em busca de plágio. Se você copiar o código de outra pessoa e enviá-lo com pequenas alterações, saberemos. Detectores de plágio são difíceis de enganar, então, por favor, não tente. Confiamos em todos vocês para que enviem apenas seu próprio trabalho; por favor não nos decepcione. Se você fizer isso, buscaremos as consequências mais duras disponíveis para nós.

Obtendo Ajuda: caso não consiga progredir, entre em contato. Desejamos que os projetos sejam recompensadores e instrutivos, não frustrantes e desmoralizantes. Mas não sabemos quando ou como ajudar a menos que você pergunte.

Discussão: por favor, tenha cuidado para não postar spoilers. **Não hospede sua solução em repositórios públicos (e.g.: github), para que ele não acabe estragando a oportunidade de algum colega aprender por seus próprios esforços.**

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 com duas saídas do exemplo na aula. O ponto azul é o agente. Observe que, quando você pressiona *para cima*, o agente só se move realmente para o norte 80% do tempo. Essa 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 padrão se move aleatoriamente

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

Você deve ver o agente aleatório se movendo insistentemente pela grade até que aconteça de chegar na saída. A vida não é fácil para um agente do GridWorld.

Nota: O Gridworld MDP é tal que você primeiro deve entrar em um estado pré-terminal (os quadrados com borda dupla mostrados na GUI) e então tomar a ação especial 'exit' antes que o episódio realmente termine (no verdadeiro estado terminal chamado TERMINAL_STATE, que não é mostrado na GUI). Se você executar um episódio manualmente, sua recompensa total pode ser menor do que você esperava, devido ao fator de desconto (-d para alterar; 0.9 por padrão).

Observe a saída do console que acompanha a saída gráfica (ou use `-t` para executar tudo no modo texto). Você será informado sobre cada transição que o agente experimenta (para desativar isso, use `-q`).

Como no Pacman, as posições são representadas por coordenadas cartesianas (x, y) e todos arrays são indexados por [x][y], com 'norte' sendo a direção de y crescente. Por padrão, a maioria das transições receberá uma recompensa zero, embora você possa mudar isso com a opção de recompensa imediata (`-r`).

Questão 0 (1 ponto) : Quiz sobre a aula 2 de RL

Essa questão é o Quiz sobre a aula 2 de aprendizagem por reforço, que deveria ser respondido até às 11h59 da manhã do dia 18/03/2019.

Questão 1 (6 pontos): 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 de iteração de valor que ele deve executar (opção -i) em sua fase inicial de planejamento. `ValueIterationAgent` recebe um MDP na sua construção e executa a iteração de valor para o número especificado de iterações antes do retorno do construtor.

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

- `computeActionFromValues(state)` calcula a melhor ação de acordo com a função de valor dada por `self.values`.
- `computeQValueFromValues(state, action)` retorna o valor-Q para o par (estado, ação) fornecido pela função valor dada por `self.values`.

Essas quantidades são todas exibidas na GUI: os valores são números nos quadrados, os valores-Q são números em quadrantes dos quadrados e as políticas são a seta em cada quadrado.

Importante: Use a versão "em lote" da iteração de valor, em que cada vetor V_k é calculado a partir de um vetor fixo V_{k-1} (como na aula), não a versão "online" onde um único vetor de peso é atualizado *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 estados sucessores usados no cálculo da atualização do valor devem ser todos da iteração k-1 (mesmo que alguns dos estados sucessores já tenham sido atualizado na iteração k). Essa diferença é discutida no livro clássico de Sutton & Barto sobre aprendizagem por reforço, no sexto parágrafo da seção 4.1.

Nota: Uma política sintetizada a partir de valores de profundidade k (que refletem as próximas k recompensas) irá na verdade refletir as próximas k + 1 recompensas (ou seja, você retorna π_{k+1}. Da mesma maneira, os valores-Q também refletirão uma recompensa além dos valores (ou seja, você retorna Q_{k+1}).

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

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

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

Para testar sua implementação, execute o autoavaliador:

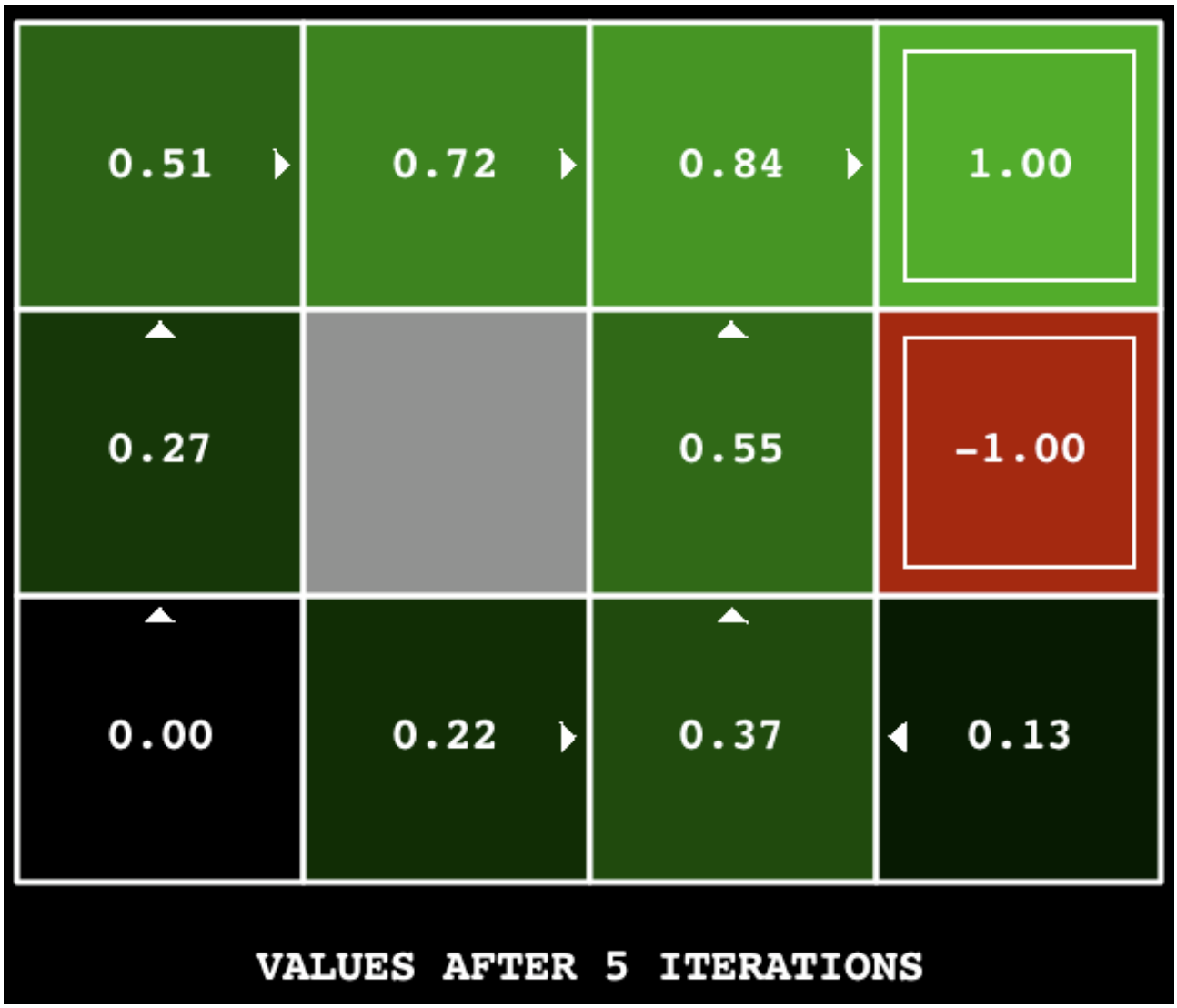
```
python autograder.py -q q1
```

O comando a seguir carrega seu `ValueIterationAgent`, que irá calcular uma política e executá-la 10 vezes. Pressione uma tecla para alternar entre valores, valores-Q e a simulação. Você deve encontrar que o valor do estado inicial (`V(start)`, que você pode ler da GUI) e a recompensa média empiricamente resultante (impressa após as 10 rodadas de execução terminarem) estão bem próximas.

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

Dica: No `BookGrid` padrão, executar a iteração de valor por 5 iterações deve fornecer essa saída:

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



Classificação: Seu agente de iteração de valor será avaliado em uma nova grade. Verificaremos seus valores, valores-Q e políticas após números fixos de iterações e na convergência (por exemplo, após 100 iterações).

Questão 2 (1 ponto): Atravessando a Ponte

O `BridgeGrid` é um mapa do grid world 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 alta recompensa negativa. O agente começa perto do 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 entre os parâmetros de desconto e ruído para que a política ótima faça com que o agente tente atravessar a ponte. Coloque sua resposta em `question2()` de `analysis.py`. (Ruído refere-se à frequência com que um agente acaba em um estado 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
```

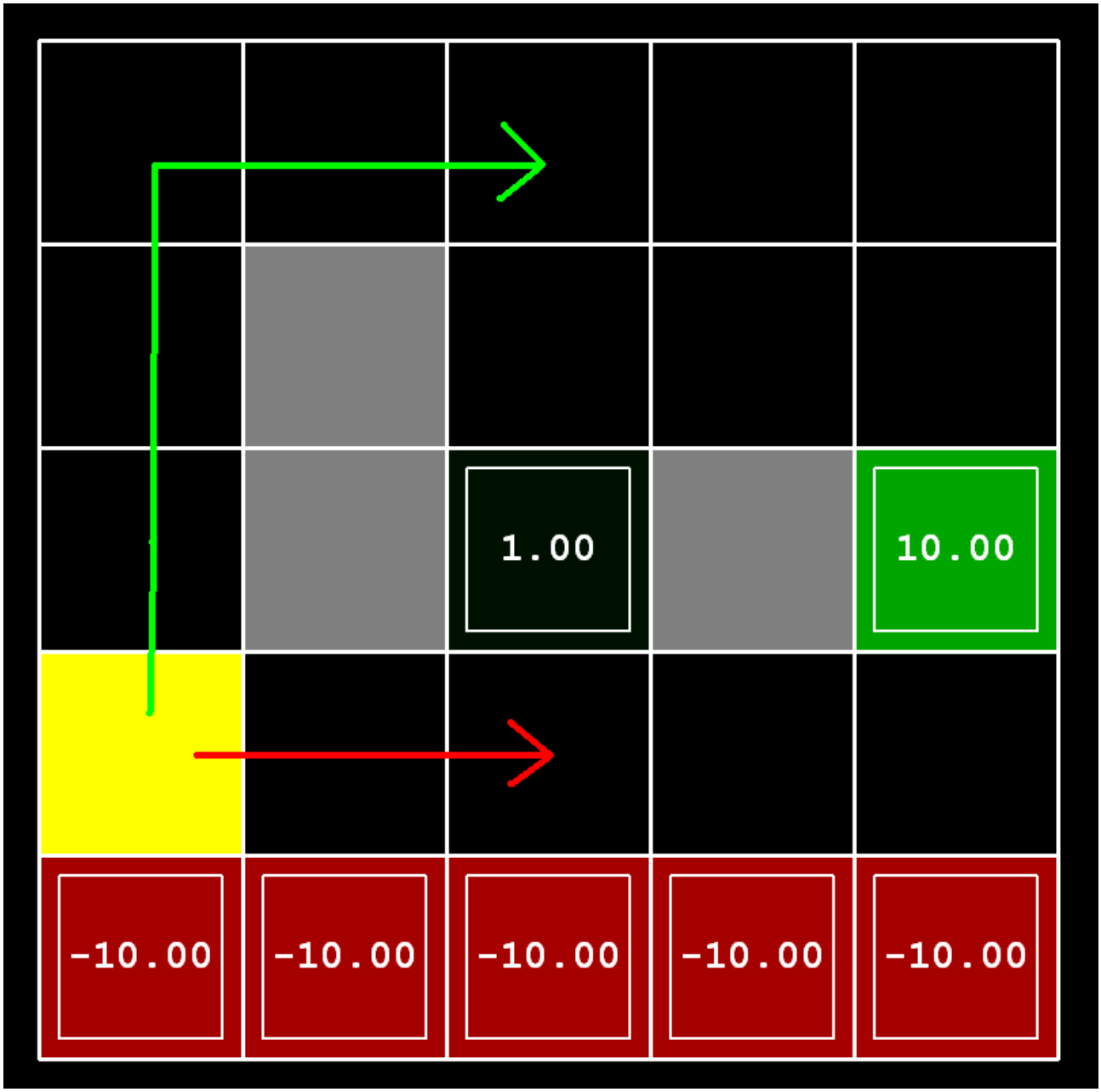
Avaliação: Verificamos que você alterou apenas um dos parâmetros mencionados e que, com essa alteração, um agente de iteração de valor correto deve cruzar a ponte. Para verificar sua resposta, execute o autoavaliador:

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

Questão 3 (5 pontos): Políticas

Considere o layout do `DiscountGrid`, mostrado abaixo. Essa grade tem dois estados terminais com recompensa positiva (na linha do meio), uma saída próxima com recompensa +1 e uma saída distante com recompensa +10. A linha inferior da grade consiste em estados terminais com recompensa negativa (mostrados em vermelho); cada estado nesta região "precipício" tem recompensa -10. O estado inicial é o quadrado amarelo. Nós distinguimos entre dois tipos de caminhos: (1) caminhos que "arriscam-se no precipício" e andam perto da linha de baixo da grade; esses caminhos são mais curtos, mas correm o risco de receber uma grande recompensa negativa, eles estsão representados pela seta vermelha na figura abaixo. (2) caminhos que "evitam o precipício" e viajam ao longo da borda superior da grade. Esses

caminhos são mais longos, mas são menos propensos a incorrer em enormes recompensas negativas. Esses caminhos são representados pela seta verde na figura abaixo.



Nesta questão, você escolherá configurações dos parâmetros de desconto, ruído e recompensa imediata desse MDP para produzir políticas ótimas de vários tipos diferentes. Sua configuração dos valores de parâmetro para cada parte deve ter a propriedade de que, se seu agente seguisse a política ideal sem estar sujeito a nenhum ruído, exibiria o comportamento determinado. Se um comportamento específico não for alcançável para qualquer configuração dos parâmetros, declare que a política é impossível retornando a string 'NOT POSSIBLE'.

Aqui estão os tipos de políticas ótimas que você deve tentar produzir:

a) Prefira a saída próxima (+1), arriscando-se no penhasco (-10)

b) Prefira a saída próxima (+1), mas evitando o penhasco (-10)

c) Prefira a saída distante (+10), arriscando-se no penhasco (-10)

d) Prefira a saída distante (+10), evitando o penhasco (-10)

e) Evite as duas saídas e o precipício (assim, um episódio nunca deve terminar)

Para verificar suas respostas, execute o autoavaliador:

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

`question3a()` até `question3e()` deve cada uma delas retornar uma tupla de 3 itens (desconto, ruído, recompensa imediata) em `analysis.py`.

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

Nota: Em alguns computadores, as setas podem não aparecer. Nesse caso, pressione uma tecla no teclado para alternar para a exibição de valores-q e calcule mentalmente a política, considerando o arg max dos valores-q disponíveis para cada estado.

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

Questão 4 (5 pontos): Q-Learning

Observe que seu agente de iteração de valor na verdade não aprende com a experiência. Em vez disso, ele analisa seu modelo de MDP para chegar a uma política completa antes de interagir com um ambiente real. Quando interage com o ambiente, ele simplesmente segue a política pré-computada (isto é, torna-se um agente reflexivo). Essa distinção pode ser sutil em um ambiente simulado como o Gridword, mas é muito importante no mundo real, onde o MDP real não está disponível.

Agora você vai escrever um agente de Q-learning, que faz muito pouco quando é construído, mas aprende por tentativa e erro a partir de interações com o ambiente através de seu método `update(state, action, nextState, reward)`. Um esqueleto de um Q-learner está especificado no `QLearningAgent` em `qlearningAgents.py`, e você pode selecioná-lo com a opção `-a q'. Para essa questão, você deve implementar os métodos `update`, `computeValueFromQValues`, `getQValue` e `computeActionFromQValues`.

Nota: Para `computeActionFromQValues`, você deve resolver os empates aleatoriamente para obter um melhor comportamento. A função `random.choice()` lhe será útil. Em um estado particular, as ações que seu agente *ainda não viu antes* ainda têm um valor-Q, mais especificamente um valor Q de zero, e se todas as ações que seu agente viu antes tiverem um valor-Q negativo, uma ação que ainda não foi vista pode ser ótima.

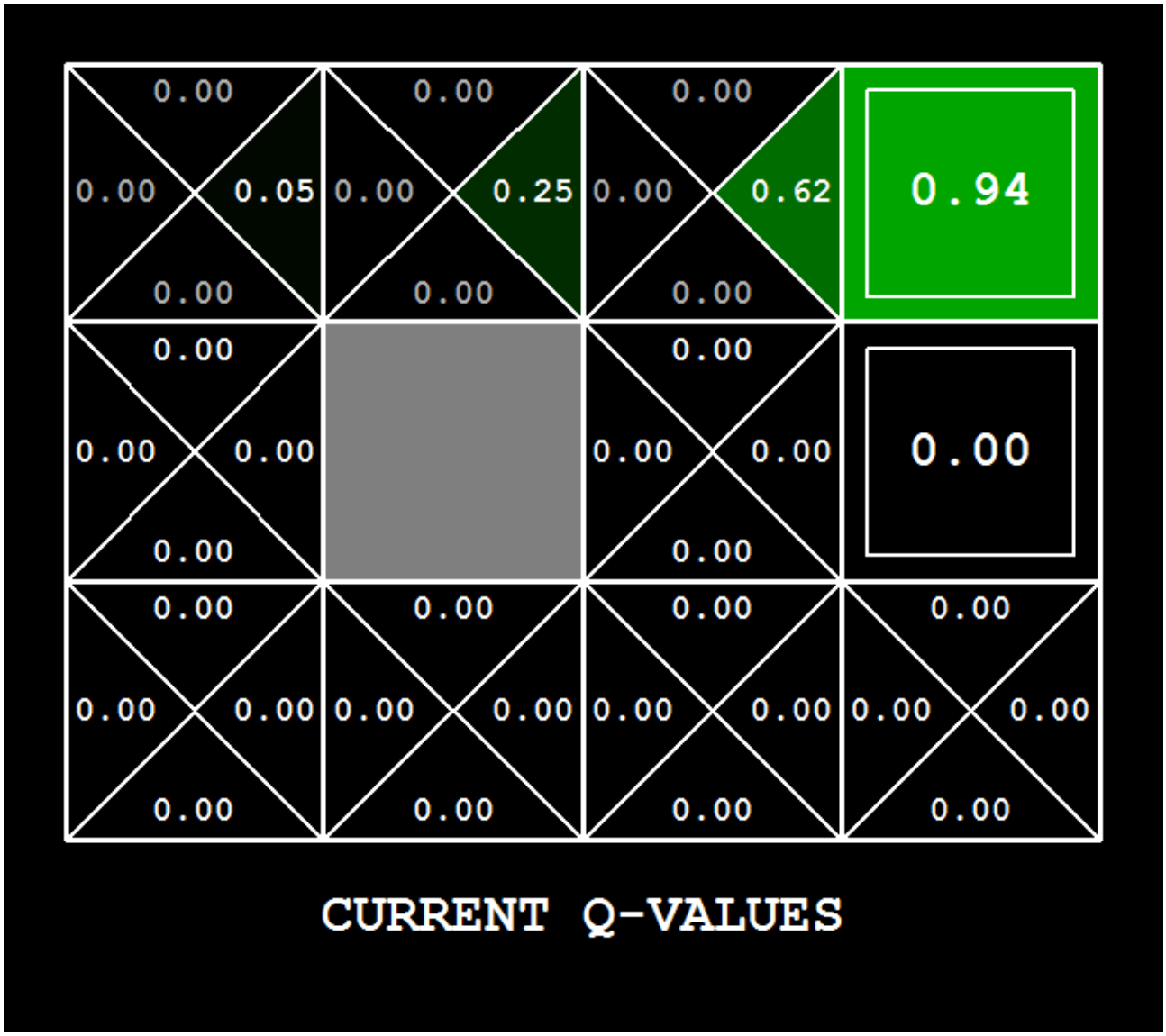
Importante: Certifique-se de que nas suas funções `computeValueFromQValues` e `computeActionFromQValues`, você só acesse valores-Q chamando `getQValue`. Essa abstração será útil para a questão 8 quando você fizer um *override* em `getQValue` para usar features de pares de ação-estado em vez de pares de ação-estado diretamente.

Com a atualização do Q-learning implementada, você poderá assistir o seu Q-learner 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 terá para aprender. Veja como o agente aprende sobre o estado em que estava, não aquele para o qual se move, e

“deixa o aprendizado em seu caminho”. Dica: para ajudar com a 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, em seguida, para leste ao longo do caminho ideal por quatro episódios, você deverá ver os seguintes valores-Q:



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

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

Questão 5 (3 pontos): 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 no resto do tempo. Note que a escolha de uma ação aleatória pode resultar na escolha da melhor ação – ou seja, você não deve escolher uma ação subótima aleatória, mas sim *qualquer* ação legal aleatória.

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

Seus valores-Q finais devem se assemelhar aos de seu agente de iteração de valor, especialmente ao longo de caminhos bem percorridos. No entanto, suas recompensas médias serão menores do que os valores-Q predizem por causa das ações aleatórias e da fase inicial de aprendizado.

Você pode escolher um elemento de uma lista de forma 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 autoavaliador:

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

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

```
python crawler.py
```

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

Isso invocará o robô rastejador visto na aula usando o seu Q-learner. Brinque com os vários parâmetros de aprendizado para ver como eles afetam as políticas e ações do agente. Observe que o atraso do passo (step delay) é um parâmetro da simulação, enquanto que 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 6 (1 ponto): Atravessando a Ponte Novamente

Primeiramente, treine um `Q-learner` completamente aleatório com a taxa de aprendizado padrão no BridgeGrid sem ruído para 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 um epsilon de 0. Existe um epsilon e uma taxa de aprendizado para a qual é altamente provável (maior que 99%) que a política ótima seja aprendida após 50 iterações? `question6()` em analysis.py deve retornar OU uma tupla de 2 itens de (epsilon, taxa de aprendizado) OU (exclusivo) a string 'NOT POSSIBLE' se não houver nenhuma tal tupla. Epsilon é controlado por `-e`, e a taxa de aprendizado por `-l`.

Nota: Sua resposta não deve depender do mecanismo exato de desempate usado para escolher ações. Isso significa que sua resposta deve estar correta, mesmo se, por exemplo, girássemos o BridgeGrid completo em 90 graus.

Para avaliar sua resposta, execute o autoavaliador:

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

Questão 7 (1 ponto): Q-Learning e Pacman

Hora de jogar um pouco de Pacman! Pacman irá jogar em duas fases. Na primeira fase, de *treinamento*, Pacman começará a aprender sobre os valores de posições e ações. Como é preciso muito tempo para aprender valores-Q precisos, mesmo para grades pequenas, os jogos de treinamento do Pacman são executados, por padrão, no modo silencioso, sem a exibição da GUI (ou do console). Uma vez que o treinamento do Pacman esteja completo, ele entrará no modo de *teste*. No teste, os `self.epsilon` e `self.alpha` do Pacman serão setados como 0.0, interrompendo efetivamente o Q-learning e desabilitando a exploração, a fim de permitir que Pacman explore sua política aprendida. Jogos de teste são mostrados na GUI por padrão. Sem quaisquer alterações de código, você deve ser capaz de executar o Q-learning Pacman para grades bem pequenas da seguinte forma:

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

Observe que o `PacmanQAgent` já está definido para você em função do `QLearningAgent` que você já escreveu. `PacmanQAgent` é diferente apenas porque possui parâmetros de aprendizado padrão que são mais eficientes para o problema Pacman (epsilon = 0.05, alpha = 0.2, gama = 0.8). Você receberá crédito total por essa questão se o comando acima funcionar sem gerar exceções e seu agente vencer em pelo menos 80% do tempo. O autoavaliador executará 100 jogos de teste depois dos 2000 jogos de treinamento.

Dica: Se o seu `QLearningAgent` funcionar para `gridworld.py` e `crawler.py`, mas não parece estar aprendendo uma boa política para o Pacman no `smallGrid`, pode ser que seus métodos `getAction` e/ou `computeActionFromQValues` não considerem apropriadamente, em alguns casos, ações jamais vistas. Em particular, visto que ações jamais tentadas têm por definição um valor-Q de zero, se todas as ações que foram tentadas tiverem valores-Q negativos, uma ação ainda não tentada pode ser ótima. Cuidado com a função `argmax` do `util.Counter`!

Nota: Para avaliar sua resposta, execute:

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

Nota: Se você quiser fazer experimentos com os parâmetros de aprendizado, você pode usar a opção `-a`, por exemplo `-a epsilon=0.1,alpha=0.3,gamma=0.7`. Estes valores serão então acessíveis como `self.epsilon`, `self.gamma` e `self.alpha` dentro do agente.

Nota: Enquanto um total de 2010 jogos serão executados, os primeiros 2000 jogos não serão exibidos por causa da opção `-x 2000`, que destina os primeiros 2000 jogos para treinamento (sem saída). Assim, você verá apenas Pacman jogar os últimos 10 destes 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: isto é porque ele ocasionalmente faz um movimento exploratório aleatório para cima de um fantasma. Como referência, deve levar entre 1000 e 1400 jogos até que as recompensas de Pacman para um segmento de 100 episódios se tornem positivas, refletindo que ele começou a ganhar mais do que perder. No final do treinamento, as recompensas devem permanecer positivas e serem razoavelmente altas (entre 100 e 350).

Certifique-se de entender o que está acontecendo aqui: o estado do MDP é a configuração **exata** do jogo que agora o Pacman enfrenta, com as transições agora complexas descrevendo uma trajetória inteira de mudanças até esse estado. As configurações intermediárias do jogo em que Pacman se moveu, mas os fantasmas não responderam, não são estados do MDP, mas estão integradas nas transições.

Uma vez que Pacman tenha terminado o treinamento, ele deve ganhar de forma muito consistente nos jogos de teste (pelo menos 90% do tempo), já que agora ele está exploitando 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 permaneceram negativas durante o treinamento. No tempo de teste, ele joga mal, provavelmente perdendo todos os seus jogos de teste. O treinamento também levará muito tempo, apesar de sua ineficácia.

O Pacman não consegue vencer em layouts maiores porque cada configuração do jogo é um estado separado com valores-Q separados. Ele não tem como generalizar o fato de que ir para cima de um fantasma é ruim para todas as posições. Obviamente, essa abordagem será escalável.

Questão 8 (3 pontos): Q-Learning Aproximado

Implemente um agente de Q-learning aproximado que aprende pesos para 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 assume a existência de uma função de features f(s, a) sobre pares de estado e ação, que produz um vetor $f_1(s, a)$, ..., $f_j(s, a)$, ..., f_n (s, a) de valores de features. Nós fornecemos funções de features para você em `featureExtractors.py`. Os vetores de features são objetos `util.Counter` (como um dicionário) de objetos contendo os pares não-nulos de features e valores; todas as features omitidas têm valor zero.

A função-Q aproximada assume 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 característica particular $f_i(s, a)$. Em seu código, você deve implementar o vetor de pesos como um dicionário de features (que serão retornadas pelos extratores de features) para valores de peso. Você atualizará seus vetores de peso de forma semelhante à atualização dos valores-Q (consultar aulas):

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

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

Note que o termo *difference* é o mesmo que no Q-learning normal, e r é a recompensa recebida.

Por padrão, o `ApproximateQAgent` usa o `IdentityExtractor`, que atribui uma feature distinta para cada par (estado, ação). Com esse 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` ao invés de acessar valores-Q diretamente, para que, ao fazer um override de `getQValue` em seu agente aproximado, os novos valores-q aproximados sejam usados para computar ações.

Quando tiver certeza de que sua implementação aproximada trabalha corretamente com as features identidade, execute seu agente Q-learning aproximado com nosso extrator de features 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 cometido erros, seu agente Q-learning aproximado deve ganhar quase todas as vezes com essas features simples, mesmo com apenas 50 jogos de treinamento.

Avaliação: Nós executaremos seu agente Q-learning aproximado e verificaremos que ele aprende os mesmos Q-values e pesos para features da nossa implementação de referência, quando são apresentados ao mesmo conjunto de exemplos. Para avaliar sua implementação, execute o autoavaliador:

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

Parabéns! Você tem um agente Pacman que aprende!

Entrega e Apresentação

Os 3 arquivos que você modificará (ver início deste documento) devem ser entregues em um único arquivo .zip até o prazo estabelecido no Classroom. O código deve estar bem comentado de forma que sua implementação seja perfeitamente compreendida. **Você também deve preparar uma apresentação (slides) para mostrar sua solução, questão por questão, na apresentação ao professor, a ser marcada após a entrega. Esses slides também devem ser colocados no mesmo .zip.**