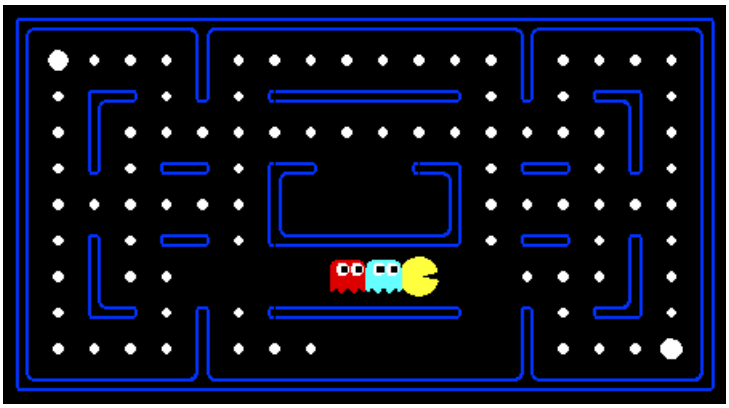
MAC0425/5739 - Inteligência Artificial

Exercício-Programa 2 - Jogos

Prazo limite de entrega: 13/05/18 às 23h55

1) Introdução

Neste projeto, você irá projetar agentes para a versão clássica do Pac-Man, incluindo fantasmas. Ao longo do caminho, você vai utilizar as buscas minimax e expectimax e projetar uma função de avaliação.



Pac-Man agora com fantasmas.
Minimax, Expectimax e Avaliação.

2) Código

A base de código não mudou muito em relação ao trabalho anterior mas, para evitar quaisquer problemas, pedimos, por favor, que faça uma nova instalação ao invés de utilizar os arquivos do primeiro trabalho.

O código para este projeto contém os seguintes arquivos (disponíveis no zip):

Arquivos que devem ser lidos:

multiAgents.py	Onde todos os seus agentes de busca multi-agente vão ficar.
pacman.py	O arquivo principal que roda jogos de Pacman. Esse arquivo também descreve o tipo *GameState*, que será amplamente usado nesse trabalho.
game.py	A lógica do mundo do Pacman. Esse arquivo descreve vários tipos auxiliares como *AgentState*, *Agent*, *Direction* e *Grid*.
util.py	Estruturas de dados úteis para implementar algoritmos de busca.

Arquivos que podem ser ignorados:

graphicsDisplay.py	Visualização gráfica do Pacman
graphicsUtils.py	Funções auxiliares para visualização gráfica do Pacman.
textDisplay.py	Visualização gráfica em ASCII para o Pacman.

ghostAgents.py	Agentes para controlar fantasmas.
keyboardAgents.py	Interfaces de controle do Pacman a partir do teclado.
layout.py	Código para ler arquivos de layout e guardar seu conteúdo.

O que deve ser entregue: Você irá preencher o arquivo `multiAgents.py` durante o trabalho. O trabalho é individual e cada aluno(a) deve entregar também um relatório respondendo as perguntas listadas abaixo. Você também pode enviar qualquer outro arquivo que tenha sido modificado por você (como `search.py`, etc.).

3) Parte Prática

Primeiramente, jogue um jogo do Pac-Man clássico:

```
$ python pacman.py
```

Agora, execute o código do agente reflexivo `ReflexAgent` que já está implementado em `multiAgents.py`:

```
$ python pacman.py -p ReflexAgent
```

Note que ele joga muito mal mesmo em layouts simples:

```
$ python pacman.py -p ReflexAgent -l testClassic
```

Inspecione o código dele (em `multiAgents.py`) e certifique-se de compreender o que ele está fazendo.

Passo 1 (2 pontos): melhore o código do `ReflexAgent` em `multiAgents.py` para que ele jogue decentemente. O código atual dá alguns exemplos de métodos úteis que consultam o estado do jogo (`GameState`) para obter informações. Um bom agente reflexivo deve considerar tanto as posições das comidas quanto as localizações dos fantasmas. O seu agente deve limpar facilmente o layout `testClassic`:

```
$ python pacman.py -p ReflexAgent -l testClassic
```

Experimente seu agente reflexivo no layout default `mediumClassic` com um ou dois fantasmas (e desligue a animação para acelerar a exibição):

```
$ python pacman.py --frameTime 0 -p ReflexAgent -k 1
```

```
$ python pacman.py --frameTime 0 -p ReflexAgent -k 2
```

(Pergunta 1): Como é o desempenho do seu agente? É provável que muitas vezes ele morra com 2 fantasmas no tabuleiro default, a não ser que a sua função de avaliação seja muito boa.

Obs1: você não pode colocar mais fantasmas do que o layout permite.

Obs2: como características, tente o inverso de valores importantes (como a distância para comida) ao invés dos próprios valores.

Obs3: a função de avaliação que você está escrevendo está avaliando pares estado-ação; na próxima parte do trabalho (com busca competitiva), a função de avaliação avaliará estados.

Opções: Os fantasmas default são aleatórios; você também pode jogar com fantasmas mais espertos usando `-g DirectionalGhost`. Se a aleatoriedade está impedindo você de perceber se o seu agente está melhorando ou não, você pode usar `-f` para executar com uma semente aleatória fixa (mesmas escolhas aleatórias a cada jogo). Você também pode jogar vários jogos em seguida com `-n`. A parte gráfica pode ser desligada com `-q` para executar muitos jogos rapidamente.

Avaliação: nós vamos executar o seu gente no layout `openClassic` 10 vezes. Você receberá 0 pontos se o seu agente causar *time out* ou não vencer em nenhuma das tentativas. Você vai receber 0,5 ponto se o seu agente vencer pelo menos 5 vezes ou 1 ponto se o seu agente vencer todas as 10 partidas. O outro 1 ponto será baseado no *score* médio do seu agente. Se for maior do que 500, você vai receber 0,5 ponto, ou 1 ponto caso seja maior do que 1000. Você pode testar o seu agente sob essas condições executando o comando:

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

Para executar sem o recurso gráfico, utilize:

```
$ python autograder.py -q q1 --no-graphics
```

Porém, não gaste muito tempo nesta questão, visto que o ponto alto do projeto vem mais à frente.

Passo 2 (3 pontos): agora você vai escrever um agente de busca competitiva na classe `MinimaxAgent` em `multiAgents.py`. O seu agente minimax deve funcionar com qualquer número de fantasmas, então você terá que escrever um algoritmo que seja um pouco mais geral do que o que aparece no livro. Em particular, a sua árvore minimax terá múltiplas camadas min (uma para cada fantasma) para cada camada max.

Seu código deve também expandir a árvore de jogo até uma profundidade arbitrária. A utilidade das folhas da árvore minimax deve ser obtida com a função `self.evaluationFunction`, que tem como default a `scoreEvaluationFunction`. A classe `MinimaxAgent` estende a classe `MultiAgentAgent`, que dá acesso às variáveis `self.depth` (profundidade da árvore) e `self.evaluationFunction` (função de avaliação). Verifique se o seu código minimax faz referência a essas duas variáveis quando necessário já que elas são preenchidas de acordo com a linha de comando.

Importante: uma busca de profundidade 1 considera uma jogada do Pac-Man e todas as respostas do fantasmas; profundidade 2 considera o Pac-Man e cada fantasma se movendo duas vezes; e assim por diante.

Avaliação: nós vamos conferir o seu código para determinar se ele explora o número correto de *game states*. Esta é a única forma confiável de detectar alguns bugs muito sutis na implementação do minimax. Como consequência, o autograder vai ser muito exigente em relação a

quantas vezes você invoca `GameState.generateSuccessor`. Se você chamar mais ou menos vezes do que o necessário, o autograder vai reclamar. Para testar e debugar seu código, execute:

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

Para executar sem o recurso gráfico, utilize:

```
$ python autograder.py -q q2 --no-graphics
```

Dicas e Observações:

- A implementação correta de minimax vai levar o Pacman a perder o jogo em alguns testes. Isto não é um problema: como é o comportamento correto, o código vai passar nos testes.
- A função de avaliação desta parte já está feita (`self.evaluationFunction`). Você não deve alterar essa função, mas reconhecer que agora estamos avaliando *estados* ao invés de ações, como fizemos para o agente reflexivo. Agentes de busca avaliam estados futuros enquanto agentes reflexivos avaliam as ações do estado atual.
- Os valores minimax do estado inicial no layout `minimaxClassic` são 9, 8, 7, -492 para profundidades 1, 2, 3 e 4 respectivamente. Note que o seu agente minimax vai vencer muitas vezes (665/1000 jogos para nós), apesar do prognóstico sombrio do minimax de profundidade 4:

  ```
  $ python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4
  ```

- Para aumentar a profundidade da busca, retire a ação `Directions.STOP` da lista de ações possíveis do Pac-Man. A busca com profundidade 2 deve ser rápida, mas com profundidade 3 ou 4 deve ser lenta.
- O Pac-Man é sempre o agente 0 e os agentes se movem em ordem crescente de índice do agente.
- Todos os estados do minimax devem ser `GameStates`, sejam passados por `getAction` ou gerados por `GameState.generateSuccessor`. Neste trabalho você não vai abstrair para estados simplificados.
- Em tabuleiros maiores como `openClassic` e `mediumClassic` (o default), o Pac-Man será bom em evitar a morte, mas não será capaz de ganhar facilmente. Muitas vezes ele vai vagar sem ter progresso. Ele pode até vagar próximo a um ponto sem comê-lo porque ele não sabe pra onde iria depois de comer o ponto. Não se preocupe se você perceber esse comportamento, no passo 5 isso será corrigido.
- Quando Pac-Man acredita que sua morte é inevitável, ele vai tentar terminar o jogo o mais rapidamente possível por causa da penalidade constante de viver. Às vezes, esta é a coisa errada a fazer com fantasmas aleatórios, mas os agentes minimax sempre supõem o pior:

  ```
  $ python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3
  ```

(Pergunta 2): Por que o Pac-Man corre para o fantasma mais próximo neste caso?

(Pergunta 3): Por que o agente reativo tem mais problemas para ganhar que o agente minimax?

(Pergunta 4): Que mudanças poderiam ser feitas na função de avaliação (evaluationFunction) para melhorar o comportamento do agente reativo?

Passo 3 (2 pontos): faça um novo agente em `AlphaBetaAgent` que use a poda alfa-beta para explorar mais eficientemente a árvore minimax. Novamente, o algoritmo deve ser um pouco mais geral do que o pseudo-código no livro, então parte do desafio é estender a lógica da poda beta-alfa adequadamente ao caso de múltiplos agentes minimizadores. Você deverá ver um aumento de velocidade (a busca com poda com profundidade 3 deve rodar tão rápido quanto a busca sem poda com profundidade 2). Idealmente, a profundidade 3 em `smallClassic` deve rodar em poucos segundos por jogada ou mais rápido.

```
$ python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic
```

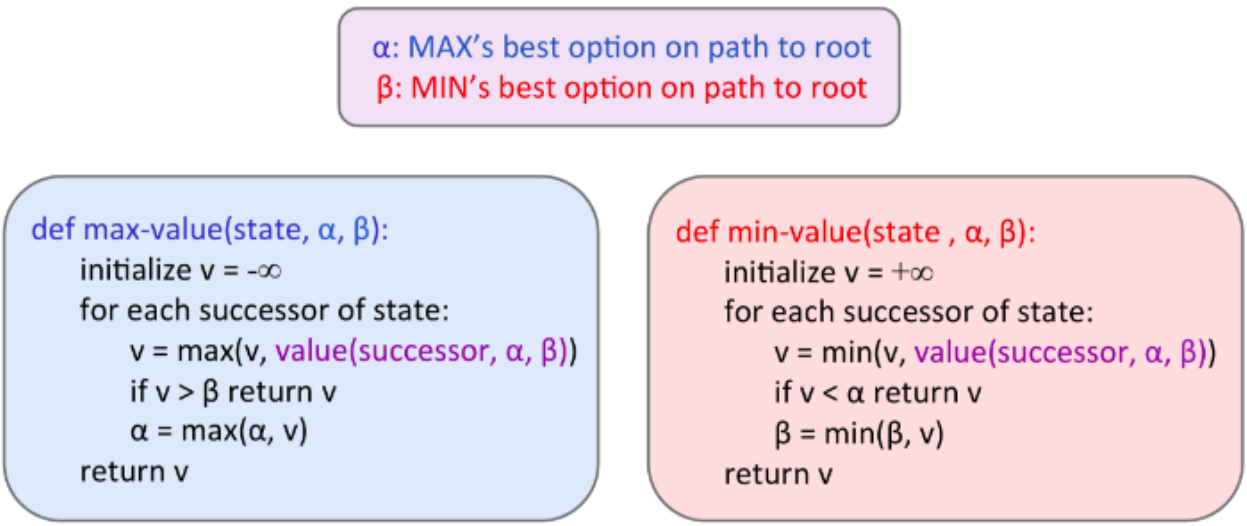
Os valores minimax do `AlphaBetaAgent` devem ser idênticos aos do `MinimaxAgent`, embora as ações selecionadas possam variar por causa de desempates diferentes. Novamente, os valores minimax do estado inicial no layout `minimaxClassic` são 9, 8, 7 e -492 para profundidades 1, 2, 3 e 4, respectivamente.

Avaliação: como nós avaliamos seu código para determinar se ele explora o número correto de estados, é importante que você execute a poda alfa-beta sem reordenar os filhos. Em outras palavras, os estados sucessores deveriam ser sempre processados na ordem devolvida por `GameState.getLegalActions`.

De novo: não invoque `GameState.generateSuccessor` mais vezes do que o necessário.

Importante: você **NÃO** deve podar de modo a forçar a execução para corresponder ao conjunto de estados explorado pelo autograder.

O pseudo-código abaixo representa o algoritmo que você deve implementar para esta questão:



Para testar e debugar seu código, execute:

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

Isto irá mostrar o que o seu algoritmo faz com um número de pequenas árvores, assim como um jogo de Pacman. Para executar sem o recurso gráfico, utilize:

```
$ python autograder.py -q q3 –no-graphics
```

(Pergunta 5): Faça uma comparação entre os agentes Minimax e AlphaBeta em termos de tempo e número de nós explorados para profundidades 2, 3 e 4.

Passo 4 (2 pontos): minimax e alfa-beta são muito bons, mas eles assumem que você está jogando contra um adversário que toma decisões ótimas. Como é fácil ver, fantasmas aleatórios não são agentes minimax ótimos. Sendo assim, utilizar a busca minimax pode não ser apropriado. Neste passo, você vai implementar o `ExpectimaxAgent`. Nele, o seu agente deixará de escolher o mínimo entre as possíveis ações dos fantasmas e calculará o valor esperado supondo que os fantasmas escolhem as ações dentre as ações legais (`getLegalAction`) aleatoriamente de maneira uniforme (`RandomGhost`).

Assim como visto nos problemas de busca e de satisfação de restrições, a beleza destes algoritmos está na aplicabilidade genérica. Para acelerar o seu próprio desenvolvimento, nós fornecemos alguns casos de teste baseados em árvores genéricas. Você pode debugar sua implementação em árvores pequenas usando o comando:

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

É recomendável debugar com estes casos de teste pequenos e gerenciáveis pois ajuda a achar bugs rapidamente. Certifique-se de que quando você computar suas médias você use *floats*. A divisão inteira em Python trunca o número, ou seja, a divisão 1 / 2 = 0, por exemplo. Com *floats*, 1.0 / 2.0 = 0.5.

Para ver como o `ExpectimaxAgent` se comporta no Pacman, execute:

```
$ python pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=3
```

Agora você deve observar uma atitude mais cavalheiresca quando o Pac-Man se aproxima dos fantasmas. Em particular, se o Pac-Man percebe que ele pode ficar preso, mas tem a possibilidade de pegar mais algumas peças de comida, ele vai pelo menos tentar. Investigue os resultados destes dois cenários:

```
$ python pacman.py -p AlphaBetaAgent -l trappedClassic -a depth=3
-q -n 10

$ python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3
-q -n 10
```

Você deve notar que o seu agente `ExpectimaxAgent` ganha aproximadamente metade das vezes, enquanto o `AlphaBetaAgent` sempre perde.

(Pergunta 6): Por que o comportamento do expectimax é diferente do minimax?

Passo 5 (1 ponto): escreva uma função de avaliação melhor para o Pac-Man dentro da função `betterEvaluationFunction`. A função de avaliação deve avaliar os estados, ao invés de ações como a função de avaliação do agente reflexivo faz. Você pode usar todas as ferramentas à sua disposição para a avaliação, incluindo seu código do primeiro trabalho. Com a busca de profundidade 2, sua função de avaliação deveria limpar o layout `smallClassic` com um fantasma aleatório mais da metade do tempo e ainda executar a uma velocidade razoável (para obter crédito total, o Pac-Man deve atingir em média cerca de 1000 pontos quando estiver ganhando).

```
$ python pacman.py -l smallClassic -p ExpectimaxAgent -a evalFn=better -q -n 10
```

Avaliação: o autograder vai executar o seu agente no layout `smallClassic` 10 vezes. Nós vamos pontuar a sua função de avaliação da seguinte forma:
- se você vencer pelo menos uma vez sem causar *time out*, você recebe 0,2; qualquer agente que não satisfizer este critério recebe 0;
- se você vencer pelo menos 5 vezes recebe mais 0,2; ao invés, se vencer todas as dez vezes recebe 0,4;
- se você atingir um *score* de pelo menos 500 recebe 0,2; ou 0,4 se atingir 1000 (incluindo scores em partidas perdidas);
- os pontos por *score* só serão computados se você vencer pelo menos 5 vezes.

Você pode testar o seu agente executando o comando:

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

(Pergunta 7): Inclua uma explicação sobre a sua função de avaliação.

Dicas e Observações:

- Da mesma forma que na função de avaliação do agente reflexivo, você deve usar o inverso de valores importantes (como a distância para a comida) ao invés dos próprios valores
- Uma maneira de escrever sua função de avaliação é usar uma combinação linear de características. Ou seja, calcular valores para características do estado que você acha que são importantes, e então combinar as características, multiplicando os valores por pesos diferentes e somando os resultados. Você pode decidir os pesos com base na importância da característica.