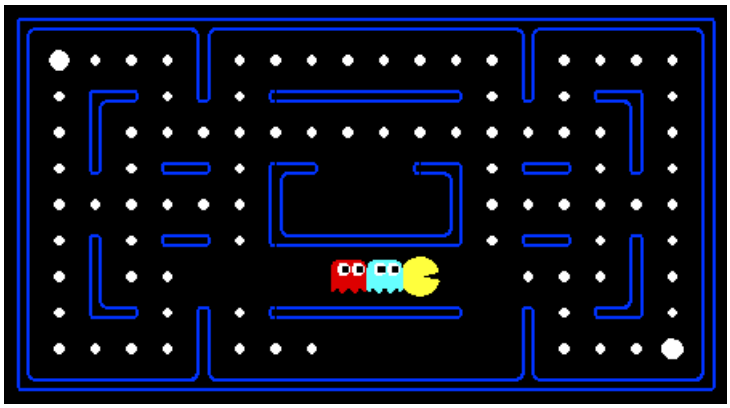
MAC0425/5739 - Inteligência Artificial

Exercício-Programa 2 - Jogos

Prazo limite de entrega: 12/06/23 às 23h55



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

1) Introdução

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

2) Código

A base de código não mudou muito em relação ao trabalho anterior, mas por favor **faça uma nova instalação**, ao invés de utilizar os arquivos do primeiro trabalho.

O código para este projeto contém os seguintes arquivos (disponíveis em ep2.zip):

Arquivos que devem ser lidos:

`pacman.py`	O arquivo principal que executa jogos Pac-Man. Este arquivo também descreve um tipo `GameState` para o Pac-Man, que você vai usar bastante neste trabalho.
`game.py`	A lógica por trás de como o mundo de Pac-Man funciona. Este arquivo descreve vários tipos de suporte como `AgentState`, `Agent`, `Direction` e `Grid`.
`util.py`	Estruturas de dados úteis para a implementação de algoritmos de busca.

Arquivo que deve ser editado:

`multiAgents.py`	Onde todos os seus agentes de busca multi-agente vão ficar.

Arquivos que você pode ignorar:

`graphicsDisplay.py`	Gráficos para o Pac-Man
`graphicsUtils.py`	Suporte gráfico para o Pac-Man
`textDisplay.py`	Gráficos ASCII para o Pac-Man
`ghostAgents.py`	Agentes de controle de fantasmas
`keyboardAgents.py`	Interfaces de teclado para controle do Pac-Man
`layout.py`	Código para leitura de arquivos de layout e armazenamento de seu conteúdo
`autograder.py`	Autoavaliador de projeto
`testParser.py`	Analisa arquivos de teste e solução do autograder
`testClasses.py`	Classes gerais de teste de autoavaliação
`test_cases/`	Diretório contendo os casos de teste para cada pergunta
`multiagentTestClasses.py`	Classes de teste específicas do Projeto 2

O que entregar: Cada dupla deve entregar o arquivo `multiAgents.py` que será modificado durante o trabalho, juntamente com um relatório contendo as respostas às perguntas abaixo.

Importante: Por favor, não altere os nomes de nenhuma das funções ou classes fornecidas dentro do código, ou você causará problemas no autograder. A correção da sua implementação, e não apenas o julgamento do autograder, é que determinará sua nota final. Se necessário, seu código será revisado e as tarefas avaliadas individualmente para garantir que você receba o devido crédito. Seu trabalho também passará por verificação de plágio. Se você copiar o código de outra dupla e enviá-lo com pequenas alterações, será detectado, e haverá consequências.

Como no EP anterior, este projeto inclui um auto-avaliador para que você possa avaliar suas próprias respostas. Ele pode ser executado com o comando:

```
$ python autograder.py
```

3) Parte Prática

Primeiro, jogue um jogo do Pac-Man Clássico, usando as setas para se movimentar:

```
$ 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 (3 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.

Nota: Você não pode colocar mais fantasmas do que o layout permite.

Nota: Lembre que `newFood` possui a função `asList()`

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

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

Nota: Vocês podem achar útil observar os conteúdos de vários objetos, imprimindo sua representação como string. Por exemplo, você pode imprimir `newGhostStates` com `print(newGhostStates)`.

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 um 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 1 ponto se o seu agente vencer pelo menos 5 vezes ou 2 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.

Vocês podem testar esta parte do código com:

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

Ou sem a parte gráfica:

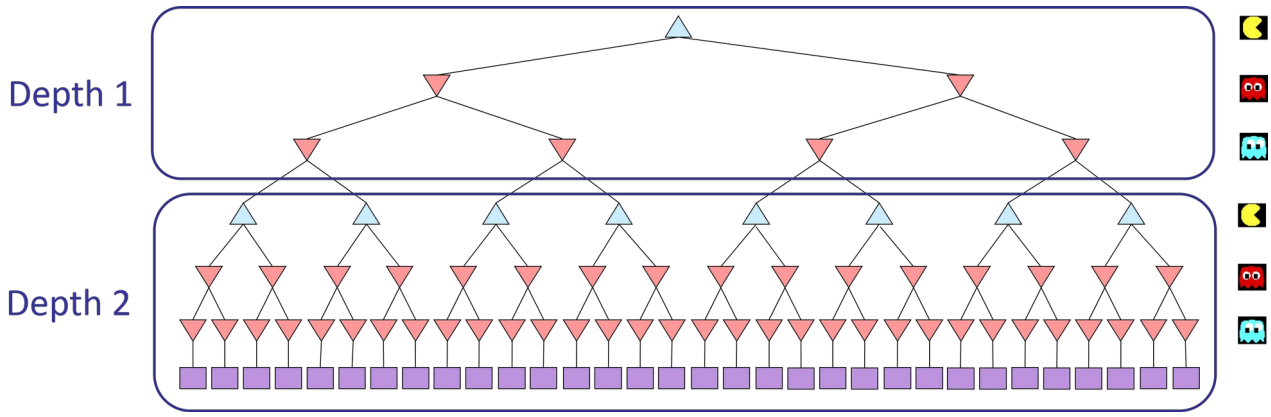
```
$ python autograder.py -q q1 --no-graphics
```

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

Passo 2 (4 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` extende a classe `MultiAgentSearchAgent`, 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 dos fantasmas, profundidade 2 considera o Pac-Man e cada fantasma se movendo duas vezes e assim por diante. Veja abaixo um exemplo para dois fantasmas.



Dicas e Observações

- Implemente o algoritmo de forma recursiva, com funções auxiliares.
- A implementação correta do minimax vai fazer o Pacman perder o jogo em alguns testes. Isso não é um problema, ele vai passar nos testes.
- A função de avaliação desta parte já está feita (`self.evaluationFunction`). Vocês não devem 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, apesar do prognóstico sombrio do minimax de profundidade 4.

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

- 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 para `getAction` ou gerados por `GameState.generateSuccessor`.
- 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
  ```

Avaliação: nós vamos conferir o seu código para determinar se ele explora o número correto de estados. 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
```

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 (3 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 alfa-beta adequadamente ao caso de múltiplos agentes minimizadores. Vocês deverão ver um aumento de velocidade (a busca com poda com profundidade 3 talvez rode 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.

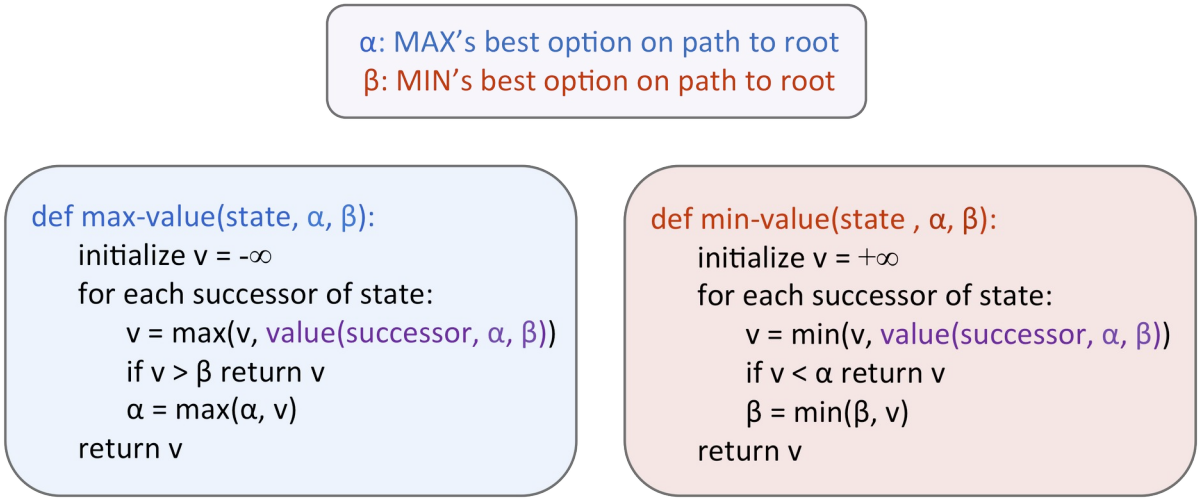
```
$ python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic
```

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 o auto-avaliador checa o código para verificar se ele explora o número correto de estados, é importante rodar a poda alfa-beta sem reordenar os nós. Isto é, os estados sucessores devem sempre ser processados na ordem devolvida por `GameState.getLegalActions`. Novamente, não chame `GameState.generateSuccessor` mais que o necessário.

Vocês NÃO devem podar em caso de igualdade, para que os estados explorados sejam iguais aos do auto-avaliador. A implementação deve seguir o algoritmo da figura abaixo.

Alpha-Beta Implementation



Mais uma vez, a implementação correta da poda alfa-beta vai fazer o Pacman perder o jogo em alguns testes. Isso não é um problema, ele vai passar nos testes.

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.

Sessão Bônus!

Passo 4 (1 ponto)

Minimax e alpha-beta são ótimos, mas ambos assumem que você está jogando contra um adversário que toma decisões ótimas. Como qualquer pessoa que já ganhou o jogo da velha pode dizer, nem sempre é esse o caso. Nesta questão, você implementará o `ExpectimaxAgent`, que é útil para modelar o comportamento probabilístico de agentes que podem fazer escolhas sub-ótimas.

Assim como a busca, a beleza desses algoritmos é sua aplicabilidade geral. Para agilizar seu próprio desenvolvimento, fornecemos alguns casos de teste baseados em árvores genéricas. Você pode depurar sua implementação em pequenas árvores de jogo usando o comando:

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

A depuração nesses casos de teste pequenos e gerenciáveis é recomendada e ajudará você a encontrar bugs rapidamente.

Depois que seu algoritmo estiver funcionando em pequenas árvores, você poderá observar sua atuação no Pacman. Fantasmas aleatórios obviamente não são agentes minimax ótimos e, portanto, modelá-los com minimax pode não ser apropriado. O `ExpectimaxAgent` não considerará mais o mínimo de todas as ações dos fantasmas, mas a esperança de acordo com o modelo do seu agente de

como os fantasmas agem. Para simplificar seu código, suponha que você estará jogando apenas contra um adversário que escolhe entre suas `getLegalActions` de maneira uniforme e aleatória.

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 abordagem mais corajosa próximo a fantasmas. Em particular, se Pacman perceber que pode estar preso, mas pode escapar para pegar mais alguns pontos de comida, ele pelo menos tentará. Investigue os resultados destes dois cenários:

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

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

Você deve descobrir que seu `ExpectimaxAgent` ganha cerca de metade das vezes, enquanto seu `AlphaBetaAgent` sempre perde. A implementação correta do expectimax fará com que o Pacman perca alguns dos testes. Isso não é problema: como é um comportamento correto, passará nos testes.

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

Entrega

Uma das pessoas da dupla deve entregar um arquivo `ep2-SeusNomesVaoAqui.zip` contendo **APENAS** os arquivos:

- `multiAgents.py` com as implementações da parte prática;
- relatório em formato PDF com as questões apresentadas para discussão dos resultados (máximo de 3 páginas + 1 página para a parte bônus).

Não esqueça de identificar cada arquivo com seu nome e número USP! Para os códigos, coloque um cabeçalho em forma de comentário.