MAC0425/5739 - Inteligência Artificial

Exercício-Programa 1 - Busca

Prazo limite de entrega: 23:59:59 11/9/2016

1 Introdução

Neste exercício-programa estudaremos a abordagem de resolução de problemas através de busca no espaço de estados, implementando um jogador inteligente de Pac-Man. Utilizaremos parte do material/código livremente disponível do curso UC Berkeley CS188 [1].

Os objetivos deste exercício-programa são:

(i) compreender a abordagem de resolução de problemas baseada em busca no espaço de estados;

(ii) estudar uma formulação de problema de busca para criar um jogador autônomo de Pac-Man;

(iii) implementar algoritmos de busca informada e não-informada e comparar seus desempenhos.

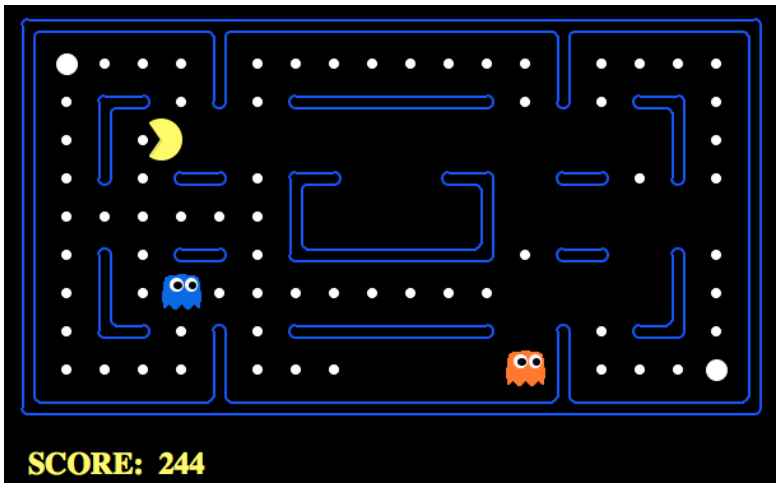

Figura 1: Pac-Man: Interface gráfica em Python

O jogo Pac-Man é um jogo eletrônico em que um jogador, representado por um boca que se abre e fecha, deve se mover em um labirinto repleto de comida e fantasmas que o perseguem. O objetivo é comer o máximo possível de comida sem ser alcançado pelos fantasmas, em ritmo progressivo de dificuldade. Existem muitas variações do jogo Pac-Man, para esse exercício-programa consideraremos alguns cenários nos quais utilizaremos algoritmos de busca para guiar nosso Pac-Man no labirinto a fim de atingir determinadas posições e coletar comida eficientemente. Além disso, através de técnicas de busca adversarial implementaremos a dinâmica de disputa com os fantasmas, como na versão clássica do jogo.

[1] `http://ai.berkeley.edu/project_overview.html`

1.1 Instalação

Para a realização deste EP será necessário ter instalado em sua máquina a versão 2.7 do Python [2]. Faça o download dos **arquivo ep1.zip** disponível na página web da disciplina. Descompacte o arquivo ep1.zip e rode na raiz do diretório o seguinte comando para testar a instalação:

```
$ cd busca/
$ python pacman.py
```

2 Pac-Man como problema de busca

Nesse exercício-programa você resolverá diversos problemas de busca. Independente da busca, a interface que implementa a formulação do problema é definida pela classe abstrata e seu conjunto de métodos abaixo (disponível no arquivo search.py na pasta **busca/**):

```
# arquivo search.py

class SearchProblem:

    def getStartState(self):
        """ Returns the start state for the search problem. """
        # ...

    def isGoalState(self, state):
        """ Returns True if and only if the state is a valid goal state. """
        # ...

    def getSuccessors(self, state):
        """ For a given state, this should return a list of triples
        (successor, action, stepCost), where 'successor' is a successor to
        the current state, 'action' is the action required to get there, and
        'stepCost' is the incremental cost of expanding to that successor. """
        # ...

    def getCostOfActions(self, actions):
        """ This method returns the total cost of a particular sequence
        of actions. The sequence must be composed of legal moves. """
        # ...
```

Note que embora a formulação de busca seja sempre a mesma para cada cenário de busca, a representação de estados varia de problema a problema. Por exemplo, veja a classe `PositionSearchProblem` no arquivo searchAgents.py e entenda a representação de estados utilizada para esse problema. Isso será importante pois você terá que implementar outra representação de estados para outro cenário de busca em um dos exercícios.

[2] `https://www.python.org/downloads/`

3 Implementação

Arquivos que você precisará editar:

- **busca/search.py** onde os algoritmos de busca serão implementados;
- **busca/searchAgents.py** onde os agentes baseados em busca serão implementados.
- **multiagentes/multiAgents.py** onde os agentes adversariais (com fantasmas) serão implementados.

Arquivos que você precisará ler e entender (em qualquer pasta):

- **pacman.py** arquivo principal para executar o jogo. Descreve a representação de estado accessível para acesso de informações dos agentes de busca.
- **util.py** estruturas de dados para auxiliar a codificação dos algoritmos de busca.

Observação: Utilize as estruturas de dados `Stack`, `Queue`, `PriorityQueue` e `PriorityQueueWithFunction` disponíveis no arquivo **util.py** para implementação da fronteira de busca. Essas implementações são necessárias para manter a compatibilidade com o aquivo de testes (**autograder.py**). Listas, tuplas, tuplas nomeadas, conjuntos e dicionários do Python podem ser utilizados sem problemas caso necessário.

4 Parte prática

Você deverá implementar algumas funções nos arquivos **search.py** e **searchAgents.py**. Não esqueça de remover o código `util.raiseNotDefined()` ao final de cada função. Para os cenários 1 e 2 você deve desenvolver sua implementação nos arquivos do diretório **busca/** e para o cenário 3 nos arquivos do diretório **multiagentes/**.

Observação: rode cada comando de teste de dentro de cada respectivo diretório.

4.1 Cenário 1 - Encontrando um ponto fixo de comida

Código 1 - Busca em Profundidade (DFS)

Implemente a busca em profundidade (DFS) no arquivo search.py na função `depthFirstSearch`. Para que a busca seja completa, implemente busca em grafo, que evita a expansão de estados previamente visitados. Para testar seu algoritmo rode os comandos:

```
$ python pacman.py -l tinyMaze -p SearchAgent
$ python pacman.py -l mediumMaze -p SearchAgent
$ python pacman.py -l bigMaze -z .5 -p SearchAgent
```

Observação: note que o tabuleiro do jogo mostra os estados visitados pela busca por cor, isto é, quanto mais vermelho escuro mais cedo na busca o estado foi visitado. Além disso, para todas essas buscas seu algoritmo DFS deve encontrar uma solução rapidamente, isto é, se demorar mais que alguns milisegundos algo provavelmente está errado no seu código!

Código 2 - Busca em Largura (BFS)

Implemente a busca em largura (BFS) no arquivo search.py na função `breadthFirstSearch`. Novamente, implemente busca em grafo. Para testar seu algoritmo rode os comandos:

```
$ python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
$ python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
```

Observação: se o Pac-Man se mover muito devagar tente usar a opção `--frameTime 0` na linha de comando.

Código 3 - Busca de Aprofundamento Iterativo (IDS)

Implemente a busca de aprofundamento iterativo (IDS) na função `iterativeDeepeningSearch` no arquivo search.py. **Lembre-se que você deve fazer uma busca em grafo**, ou seja, memorizando os estados visitados. Para que a busca IDS em grafo continue ótima (como é no caso de busca em árvore), é necessária uma pequena modificação. Ao explorar um nó cujo estado já foi visitado, descartamos esse nó **apenas** se seu custo for maior que o menor custo de um nó já visitado associado ao mesmo estado. Para testar seu algoritmo rode os comandos:

```
$ python pacman.py -l mediumMaze -p SearchAgent -a fn=ids
$ python pacman.py -l bigMaze -p SearchAgent -a fn=ids -z .5
```

Observação: Você pode comparar seu algoritmo IDS com o algoritmo BFS para verificar a otimalidade da solução.

Código 4 - Variando a função custo

Implemente a busca de custo uniforme (UCS) no arquivo search.py na função `uniformCostSearch`. Novamente, implemente busca em grafo. Para testar seu algoritmo rode o comando:

```
$ python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
```

Código 5 - Busca heurística

Implemente busca em grafo A* no arquivo search.py na função `aStarSearch`. Para testar seu algoritmo rode o comando:

```
$ python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
```

4.2 Cenário 2 - Encontrando os cantos do labirinto

Código 6 - Formulação de problema de busca dos 4 cantos

Nesse exercício vamos implementar um novo problema de busca. Você deverá completar a implementação dos métodos da classe `CornersProblem` no arquivo searchAgents.py. Você deverá escolher

uma representação de estados que codifique **somente a informação necessária** para detectar se todos os 4 cantos do labirinto foram visitados. Para testar a formulação do problema rode o comando:

```
$ python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
$ python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
```

Observação: Não utilize um objeto `GameState` como estado de busca! Se utilizar, seu código provavelmente ficará errado e muito lento.

Código 7 - Heurística para o problema dos 4 cantos

Além da nova formulação de problema, você deverá implementar uma heurística não trivial e consistente na função `cornersHeuristic`. Para testar sua heurística rode o comando:

```
$ python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
```

Nesse exercício consideramos heurísticas triviais aquelas que retornam 0 para todos os estados ou calculam o valor real do custo (por meio de buscas sistemáticas). O primeiro tipo de heurística trivial não ajuda muito do ponto de vista da eficiência do algoritmo e o último tipo demora muito mais tempo que o ideal.

4.3 Cenário 3 - Fugindo de fantasmas

Nesse cenário passaremos a considerar os fantasmas como adversários e implementaremos o comportamento do Pac-Man durante uma partida.

Código 8 - Agente reativo

Melhore o agente reativo (`ReflexAgent`) em `multiAgents.py` para jogar de um jeito aceitável. O agente deve considerar as posições da comida e dos fantasmas no tabuleiro para jogar melhor. Seu agente deve ganhar fácilmente o tabuleiro `testClassic` (primeiro comando).

```
$ python pacman.py -p ReflexAgent -l testClassic
$ python pacman.py --frameTime 0 -p ReflexAgent -k 1
$ python pacman.py --frameTime 0 -p ReflexAgent -k 2
```

Observação: Você precisa modificar apenas a função `evaluationFunction` da classe `ReflexAgent`.

Código 9 - Minimax

Nesse caso você precisa implementar um agente adversarial usando o algoritmo Minimax em `multiAgents.py`. O agente minimax deve funcionar com qualquer número de fantasmas, em particular deve ter várias camadas min (uma por cada fantasma) para cada camada max.

```
$ python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=2
$ python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4
```

Importante: Usar as variáveis self.depth e self.evaluationFunction na sua implementação.

Observação: Cada busca deve considerar um movimento do Pac-Man e dos fantasmas. Ou seja, uma busca com profundidade 2 é mover Pac-Man e cada um dos fantasmas duas vezes.

Código 10 - AlphaBeta

Além de implementar o algoritmo Minimax, você terá que implementar a poda alpha-beta (`AlphaBetaAgent`) para explorar a árvore minimax eficientemente.

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

5 Relatório

Após o desenvolvimento da parte prática, você deverá testar seus algoritmos e redigir um relatório claro e sucinto. Assim, você deverá:

(a) compilar em tabelas as estatísticas das buscas implementadas em termos de nós expandidos e comprimento de plano;

(b) discutir os méritos e desvantagens de cada método, no contexto dos dados obtidos;

(c) responder as questões teóricas;

(d) sugerir possíveis melhorias ao sistema e relatar dificuldades.

5.1 Questões

Questão 1 - Busca em Profundidade A ordem de exploração do espaço de estados seguiu conforme esperado? O Pac-Man de fato se move para todos os estados explorados em sua deliberação para encontrar uma solução para a meta?

Questão 2 - Busca em Largura A sua implementação de busca em largura encontra uma solução de custo mínimo? Por quê?

Questão 3 - Busca de Custo Uniforme Execute os comandos abaixo e explique sucintamente o comportamento os agentes `StayEastSearchAgent` e `StayWestSearchAgent` em termos da função custo utilizada por cada agente.

```
$ python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
$ python pacman.py -l mediumDottedMaze -p StayWestSearchAgent
```

Questão 4 - Busca de Aprofundamento Iterativo Por que a busca IDS não é ótima em busca em grafo (sem a modificação)? Por que se torna ótima adaptando a maneira de descartar nós da busca em grafo? Por que para os problemas de busca estudados neste EP não é aconselhável implementar o algoritmo IDS com busca em árvore?

Questão 5 - Busca Heurística Você deve ter percebido que o algoritmo A* encontra uma solução mais rapidamente que outras buscas. Por quê? Qual a razão para se implementar uma heurística consistente para sua implementação da busca A*?

Questão 6 - Agente adversariais Por que o agente reativo tem mais problemas para ganhar que o agente minimax? Que mudanças poderia fazer na função de avaliação (`evaluationFunction`) para melhorar o comportamento do agente reativo?

6 Entrega

Você deve entregar um arquivo `ep1-SeuNomeVaiAqui.zip` contendo **APENAS** os arquivos:

(1) **search.py**, **searchAgents.py** e **multiAgents.py** com as implementações da parte prática;

(2) **relatório** em formato PDF com as questões e comparação e discussão dos resultados (máximo de 2 páginas).

Não esqueça de identificar cada arquivo com seu nome e número USP! No código coloque um cabeçalho em forma de comentário.

7 Critério de avaliação

O critério de avaliação dependerá parcialmente dos resultados dos testes automatizados do autograder.py. Dessa forma você terá como avaliar por si só parte da nota que receberá para a parte prática. **Note que para o algoritmo IDS não há testes automatizados no autograder**. Avaliaremos esse algoritmo separadamente. Para rodar os testes automatizados, execute o seguinte comando de dentre de cada diretório **busca/** e **multiagentes/**:

```
$ python autograder.py
```

Com relação ao relatório, avaliaremos principalmente sua forma de interpretar comparativamente os desempenhos de cada busca. Não é necessário detalhar a estratégia de cada busca ou qual estrutura de dados você utilizou para cada busca, mas deve ficar claro que você compreendeu os resultados obtidos conforme esperado dado as características de cada busca.

Parte prática (36 pontos)

- Cenário 1: autograder (12 pontos)
- Cenário 2: autograder (6 pontos)
- Cenário 3: autograder (14 pontos)
- Código 3 (IDS) (4 pontos)

Relatório (24 pontos)

- Questões: (10 pontos)
- Comparação e Discussão: (14 pontos)