MAC0425/5739 - Inteligência Artificial

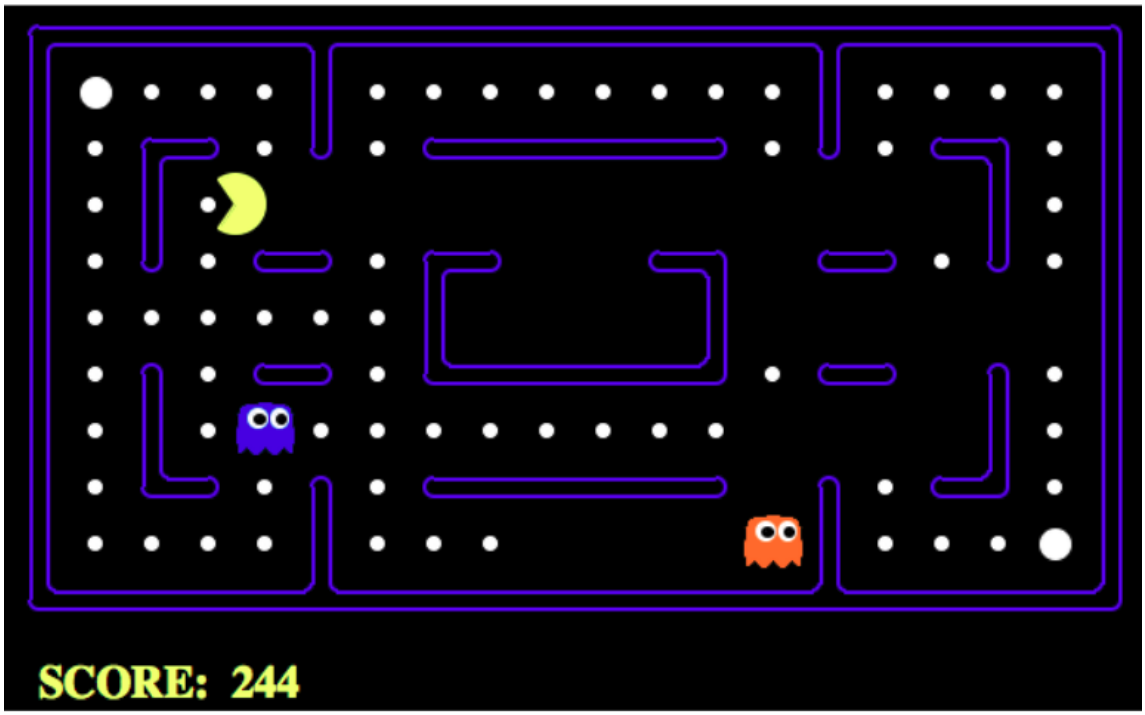
Exercício-Programa 1 - Busca

Prazo limite de entrega: 13/04/18 às 23h55

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.

Pac-Man é um jogo eletrônico em que um jogador, representado por uma boca que se abre e fecha, deve mover-se 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 este exercício-programa, consideraremos um cenário no qual utilizaremos algoritmos de busca para guiar nosso Pac-Man no labirinto a fim de atingir determinadas posições e coletar comida eficientemente. Técnicas de busca adversarial, porém, serão abordadas posteriormente.

Os objetivos deste exercício-programa são:

- compreender a abordagem de resolução de problemas baseada em busca no espaço de estados;
- estudar uma formulação de problema de busca para criar um jogador autônomo de Pac-Man;
- implementar algoritmos de busca informada e não-informada e comparar seus desempenhos.

Instalação:

Para a realização deste EP será necessário ter instalado em sua máquina a versão 2.7 do Python. Feito o download do arquivo ep1.zip disponível no PACA, descompacte o arquivo e rode na raiz do diretório o seguinte comando para testar a instalação:

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

O agente mais simples em *searchAgents.py* é o agente `GoWestAgent`, que sempre vai para oeste (um agente reflexivo trivial). Este agente pode ganhar às vezes:

```
$ python pacman.py --layout testMaze --pacman GoWestAgent
```

Mas as coisas se tornam mais difíceis quando virar é necessário:

```
$ python pacman.py --layout tinyMaze --pacman GoWestAgent
```

pacman.py tem opções que podem ser dadas em formato longo (por exemplo, `--layout`) ou em formato curto (por exemplo, `-l`). A lista de todas as opções pode ser vista executando:

```
$ python pacman.py -h
```

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

2) Códigos

O código deste trabalho consiste de diversos arquivos Python, alguns dos quais você terá que ler e entender para fazê-lo.

Arquivo que deve ser editado:

busca/search.py	Onde ficam os algoritmos de busca.

Arquivos que devem ser lidos:

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.
busca/searchAgents.py	Onde ficam os agentes baseados em 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.

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: o arquivos `search.py`, que será modificado no trabalho conforme especificado anteriormente. O trabalho é individual e cada aluno(a) deve entregar também um relatório respondendo as perguntas listadas abaixo.

3) Parte prática

Você deverá implementar algumas funções no arquivo *`search.py`*. Não esqueça de remover o código *`util.raiseNotDefined()`* ao final de cada função implementada.

Nossa meta será encontrar comida em um ponto fixo usando algoritmos de busca.

No arquivo *`searchAgents.py`*, você irá encontrar o programa de um agente de busca (*`SearchAgent`*), que planeja um caminho no mundo do Pacman e executa o caminho passo-a-passo. Os algoritmos de busca para planejar o caminho não estão implementados -- este será o seu trabalho. Para entender o que está descrito a seguir, pode ser necessário olhar o glossário de objetos ao final deste enunciado. Primeiro, verifique que o agente de busca *`SearchAgent`* está funcionando corretamente, rodando:

```
$ python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
```

O comando acima faz o agente *`SearchAgent`* usar o algoritmo de busca *`tinyMazeSearch`*, que está implementado em *`search.py`*. O Pacman deve navegar o labirinto corretamente.

Agora chegou a hora de implementar os seus algoritmos de busca para o Pacman! Os pseudocódigos dos algoritmos de busca estão no livro-texto. Lembre-se que um nó da busca deve conter não só o estado mas também toda a informação necessária para reconstruir o caminho (sequência de ações) até aquele estado.

Importante: Todas as funções de busca devem retornar uma lista de *ações* que irão levar o agente do início até o objetivo. Essas ações devem ser legais (direções válidas, sem passar pelas paredes).

Dica: Os algoritmos de busca são muito parecidos. Os algoritmos de busca em profundidade, busca em extensão e A* diferem somente na ordem em que os nós são retirados da borda. Então o ideal é tentar implementar a busca em profundidade corretamente e depois será mais fácil implementar as outras. Uma possível implementação é criar um algoritmo de busca genérico que possa ser configurado com uma estratégia para retirar nós da borda. (Porém, implementar dessa forma não é necessário).

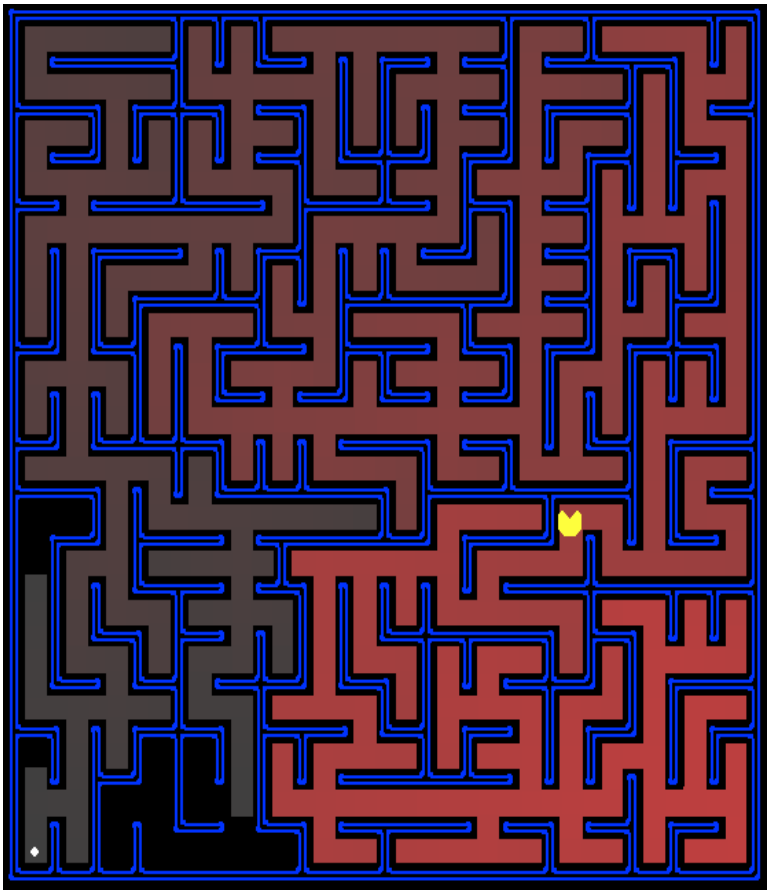
➔ **Passo 1 (2 pontos):** implemente o algoritmo de busca em profundidade (DFS) na função `depthFirstSearch` do arquivo `search.py`. Para que a busca seja completa, implemente a versão de DFS que não expande estados repetidos (seção 3.5 do livro). Teste seu código executando:

```
$ python pacman.py -l tinyMaze -p SearchAgent

$ python pacman.py -l mediumMaze -p SearchAgent

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

A saída do Pacman irá mostrar os estados explorados e a ordem em que eles foram explorados (vermelho mais forte significa que o estado foi explorado antes).



➔ **(Pergunta 1):** a ordem de exploração foi de acordo com o esperado? O Pacman realmente passa por todos os estados explorados no seu caminho para o objetivo?

Dica: Se você usar a pilha `Stack` como estrutura de dados, a solução encontrada pelo algoritmo DFS para o `mediumMaze` deve ter comprimento 130 (se os sucessores forem colocados na pilha na ordem dada por `getSuccessors`; pode ter comprimento 246 se forem colocados na ordem reversa).

➔ **(Pergunta 2):** essa é uma solução ótima? Senão, o que a busca em profundidade está fazendo de errado?

➔ **Passo 2 (2 pontos):** Implemente o algoritmo de busca em extensão (BFS) na função `breadthFirstSearch` do arquivo `search.py`. De novo, implemente a versão que não expande estados que já foram visitados. Teste seu código executando:

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

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

➔ **(Pergunta 3):** a busca BFS encontra a solução ótima? Por quê? Se não, verifique a sua implementação.

➔ **Passo 3 (2 pontos):** 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.

➔ **(Pergunta 4):** 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?

➔ **Passo 4 (2 pontos):** implemente a busca A* (com checagem de estados repetidos) na função `aStarSearch` do arquivo `search.py`. A busca A* recebe uma heurística como parâmetro. Heurísticas têm dois parâmetros: um estado do problema de busca (o parâmetro principal) e o próprio problema. A heurística implementada na função `nullHeuristic` do arquivo `search.py` é

um exemplo trivial. Teste sua implementação de A* no problema original de encontrar um caminho através de um labirinto para uma posição fixa usando a heurística de distância Manhattan (implementada na função *`manhattanHeuristic`* do arquivo *`searchAgents.py`*).

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

➔ **(Pergunta 5):** 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*?

➔ **(Pergunta 6):** o que acontece em *`openMaze`* para as várias estratégias de busca?

4) Entrega

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

- `search.py` com as implementações da parte prática;
- relatório em formato PDF com as questões apresentadas para discussão dos resultados (máximo de 2 páginas).

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

5) Avaliação

O critério de avaliação dependerá parcialmente dos resultados dos testes automatizados do `autograder.py`. Desta 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 dentro do diretório *`busca/`*:

```
$ python autograder.py
```

Em relação ao relatório, que vale 2 pontos, 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.

6) Glossário de Objetos

Este é um glossário dos objetos principais na base de código relacionada a problemas de busca:

- `SearchProblem (search.py)`
  Um *SearchProblem* é um objeto abstrato que representa o espaço de estados, função sucessora, custos, e estado objetivo de um problema. Você vai interagir com objetos do tipo *SearchProblem* somente através dos métodos definidos no topo de *`search.py`*
- `PositionSearchProblem (searchAgents.py)`
  Um tipo específico de *SearchProblem* --- corresponde a procurar por uma única comida no labirinto.
- Função de Busca
  Uma função de busca é uma função que recebe como entrada uma instância de *SearchProblem*, roda algum algoritmo, e retorna a sequência de ações que levam ao objetivo. Exemplos de função de busca são *`depthFirstSearch`* e *`breadthFirstSearch`*, que deverão ser implementadas. A função de busca dada *`tinyMazeSearch`* é uma função muito ruim que só funciona para o labirinto *`tinyMaze.`*
- `SearchAgent`
  `SearchAgent` é uma classe que implementa um agente (um objeto que interage com o mundo) e faz seu planejamento de acordo com uma função de busca. *`SearchAgent`* primeiro usa uma função de busca para encontrar uma sequência de ações que levem ao estado objetivo, e depois executa as ações uma por vez.