Atividade Prática 02
Algoritmos de Busca

Universidade Tecnológica Federal do Paraná (UTFPR), campus Apucarana
Curso de Engenharia de Computação
Sistemas Inteligentes 1 - SICO7A
Prof. Dr. Rafael Gomes Mantovani

Instruções: Este trabalho é parte do `Pacman Project`[a] desenvolvido na UC Berkeley na disciplina "*CS188 – Artificial Intelligence*". A tradução foi baseado no trabalho de buscas da prof. Bianca Zadrozny (UFF) e prof. Eduardo Bezerra (CEFET/RJ).

[a] http://ai.berkeley.edu/project_overview.html

1 Descrição da atividade

Neste trabalho, o agente Pacman tem que encontrar caminhos no labirinto, tanto para chegar a um destino quanto para coletar comida eficientemente. O objetivo do trabalho será programar algoritmos de busca e aplicá-los ao cenário (mundo) do Pacman. O código fornecido nesse trabalho consiste de diversos arquivos Python, alguns dos quais você terá que ler e entender para fazer o trabalho. A figura 1 apresenta um tpo de resolução de problema baseado em buscas para o Pacman. O código está no Moodle, arquivo `search.zip`. Na Tabela 1 estão descritos todos os arquivos contidos no zip.

Você (equipe) deve entregar um único arquivo compactado contendo os seguintes itens:

- os arquivos **search.py** e **searchAgents.py** alterados depois da codificação. Não entregue outros arquivos além desses dois;
- um relatório (em PDF) apresentando análise e comentários/explicações sobre os resultados obtidos. Descreva e explique também partes relevantes do código implementado.
- O trabalho deve ser submetido pelo Moodle até o prazo final estabelecido na página do curso.

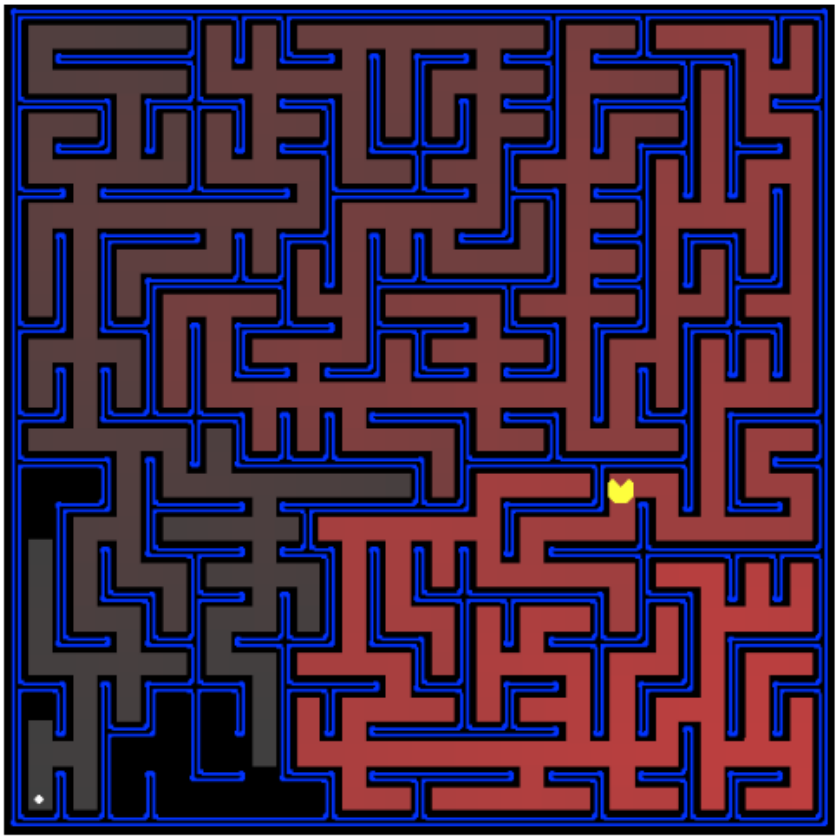

Figura 1: Pacman em ação. O caminho de busca encontrado pelo agente (Pacman) no labirinto é representado em cores. Estados visitados mais de uma vez apresentam coloração mais avermelhada.

2 Instruções Gerais

Depois de baixar o código (*search.zip*), descompactá-lo e entrar na pasta *search*, você pode jogar um jogo de Pacman digitando a seguinte linha de comando:

```
python pacman.py
```

O agente mais simples em **searchAgents.py** é `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
```

A aplicação tem opções que podem ser dadas em formato longo (por exemplo, –layout) ou

Tabela 1: Arquivos contidos no pacote de implementação do Pacman em Python.

Arquivos que devem ser editados	
`search.py`	onde ficam os algoritmos de busca
`searchAgent.py`	onde ficam os agentes baseados em 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 neste 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	
`graphicalDisplay.py`	Visualização gráfica do Pacman
`graphicUtils.py`	Funções auxiliares para visualização 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

em formato curto (-l). A lista de todas as opções pode ser vista executando:

```
python pacman.py --help
```

Todos os comandos que aparecem aqui também estão descritos em *commands.txt*, e podem ser copiados e colados no terminal.

2.1 Glossário de Objetos

Este é um glossário dos objetivos 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.
- **FoodSearchProblem (searchAgents.py)**: Um tipo específico de `SearchProblem` — corresponde a procurar um caminho para comer toda a comida em um 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 escritas pelo grupo. A função de busca dada `tinyMazeSearch` é uma função muito ruim que só funciona para o labirinto `tinyMaze`;

- **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. `SearchProblem` 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.

2.2 Encontrando 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**. 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.
Para implementar os seus algoritmos de busca para o Pacman, use os pseudocódigos dos algoritmos de busca que estão no livro-texto. Lembre-se de 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, busca de custo uniforme 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). Dê uma olhada no código dos tipos *Stack* (pilha), *Queue* (fila) e *PriorityQueue* (fila com prioridade) que estão no arquivo **util.py**.

3 Atividades

3.1 Busca em Profundidade (*Depth First Search* - DFS)

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. Teste seu código executando:

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

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). **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).
Algumas perguntas:

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?

2. Essa é uma solução ótima? Senão, o que a busca em profundidade está fazendo de errado?

3.2 Busca em Amplitude (*Breadth First Space* - BFS)

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
```

A busca BFS encontra a solução ótima? Senão, verifique a sua implementação. Se o seu código foi escrito de maneira correta, ele deve funcionar também para o quebra-cabeças de 8 peças sem modificações.

```
python eightpuzzle.py
```

Quantas ações compõem a solução encontrada pelo BFS?

3.3 Busca com Custo Uniforme (*Uniform Cost Search* - UCS)

A busca BFS vai encontrar o caminho com o menor número de ações até o objetivo. Porém, podemos querer encontrar caminhos que sejam melhores de acordo com outros critérios.

Considere o labirinto *mediumDottedMaze* e o labirinto *mediumScaryMaze*. Mudando a função de custo, podemos fazer o Pacman encontrar caminhos diferentes. Por exemplo, podemos ter custos maiores para passar por áreas com fantasmas e custos menores para passar em áreas com comida, e um agente Pacman racional deve poder ajustar o seu comportamento. Implemente o algoritmo de busca de custo uniforme (checando estados repetidos) na função `uniformCostSearch` do arquivo **search.py**. Teste seu código executando os comandos a seguir, onde os agentes têm diferentes funções de custo (os agentes e as funções são dados):

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```

3.4 Busca A*

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
```

A busca A* deve achar a solução ótima um pouco mais rapidamente que a busca de custo uniforme (549 vs. 621 nós de busca expandidos na nossa implementação). O que acontece em openMaze para as várias estratégias de busca?

3.5 Coletando comida

Agora iremos atacar um problema mais difícil: fazer o Pacman comer toda a comida no menor número de passos possível. Para isso, usaremos uma nova definição de problema de busca que formaliza esse problema: ***FoodSearchProblem*** no arquivo **searchAgents.py** (já implementado). Uma solução é um caminho que coleta toda a comida no mundo do Pacman. A solução não será modificada se houver fantasmas no caminho; ela só depende do posicionamento das paredes, da comida e do Pacman. Se os seus algoritmos de busca estiverem corretos, A* com uma heurística nula (equivalente a busca de custo uniforme) deve encontrar uma solução para o problema **testSearch** sem nenhuma mudança no código (custo total de 7).

```
python pacman.py -l testSearch -p AStarFoodSearchAgent
```

Nota: `AStarFoodSearchAgent` é um atalho para:

```
-p SearchAgent -a fn=astar,prob=FoodSearchProblem,
                 heuristic=foodHeuristic
```

Porém, a busca de custo uniforme fica lenta até para problemas simples como *tinySearch*. Implemente uma heurística admissível *foodHeuristic* no arquivo **searchAgents.py** para o problema `FoodSearchProblem`. Teste seu agente no problema *trickySearch*:

```
python pacman.py -l trickySearch -p AStarFoodSearchAgent
```

4 Links

- http://ai.berkeley.edu/project_overview.html
- https://github.com/nomaanakhtar/Berkeley-AI-Pacman-Search
- http://ai.berkeley.edu/search.html
- http://ai.berkeley.edu/course_schedule.html

Referências

[1] LUGER, George F. Inteligência artificial. 6. ed. São Paulo, SP: Pearson Education do Brasil, 2013. xvii, 614 p. ISBN 9788581435503.

[2] RUSSELL, Stuart J.; NORVIG, Peter. Inteligência artificial. Rio de Janeiro, RJ: Elsevier, 2013. 988 p. ISBN 9788535237016.