Aplicação de técnicas de IA no domínio do Pacman - PARTE I

(Trabalho em triplas – entrega: ver DATAS)

O trabalho proposto é baseado no “Pacman Project” da Universidade de Berkeley, o qual visa aplicar conceitos de IA no domínio do Pacman.

Não divulgue respostas em hipótese alguma. As consequências podem ser severas.

O Pacman deve encontrar um caminho para atingir certas posições no seu mundo, composto inicialmente por um labirinto.

O código foi desenvolvido na linguagem Python e é constituído de vários arquivos Python, que você deverá ler, entender e completar. O arquivo compactado está disponível no moodle (search.zip).

As soluções que você encontrar não devem ser divulgadas em hipótese alguma, pois isto prejudicará o “Pacman Project” da Universidade de Berkeley e é passível de penalizações.

Você deve implementar os seguintes algoritmos de busca genéricos para melhorar o desempenho do Pacman:

1) Busca em Profundidade com retrocesso - DFS.
2) Busca em Extensão (Largura) - BFS.
3) Busca de Custo Uniforme - UCS
4) Busca A*.

Arquivo que você deve modificar:	
search.py	Seus algoritmos de busca.
Arquivos que você poderá se interessar:	
pacman.py	Arquivo principal.
game.py	Lógica que rege o mundo do Pacman: AgentState, Agent, Direction, and Grid.
util.py	Estruturas de dados úteis para algoritmos de busca.
searchAgents.py	Seus agentes de busca.
Você pode ignorar:	
graphicsDisplay.py	GUI
graphicsUtils.py	GUI
textDisplay.py	GUI
ghostAgents.py	Controle de fantasmas.
keyboardAgents.py	Interface de teclado.
layout.py	Leitura de cenários.

O que entregar: Você deverá submeter uma pasta compactada (<nome>.zip) contendo APENAS 2 arquivos: search.py modificado e o relatório.

ATENÇÃO:

NÃO altere nome de funções ou classes dos arquivos.

NÃO copie código, pois haverá um detector de plágio automático.

Bem-vindo ao mundo Pacman

Depois de baixar o código (search.zip), descomprima e vá para o diretório *search*, onde você poderá jogar Pacman com o comando:

`python pacman.py` (no "prompt" do Unix)

`pacman.py` (no "prompt" do Windows, o qual será utilizado no restante do enunciado)

Atenção: em caso de erro, muito provavelmente você está com o Python mal configurado. Em particular, o pacote de interface tk (tkinter) pode estar desconfigurado ou não instalado. O código está testado para versão 2.7 do Python.

O agente mais simples definido em searchAgents.py é um agente chamado `GoWestAgent`. Dependendo do labiranto, ele consegue executar sua tarefa, como por exemplo em:

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

Mas, se por acaso é preciso algo mais elaborado, não funciona:

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

Caso seu agente fique preso, você pode sair do jogo teclando CTRL-c. Note que `pacman.py` suporta um grande número de opções. Você tem acesso à lista e aos valores default digitando:

```
pacman.py -h
```

Achando um ponto de comida fixo

No arquivo `searchAgents.py`, você encontra a implementação de `SearchAgent`, que planeja um caminho no no labirinto e depois o executa passo-a-passo. Seu trabalho é implementar os algoritmos de busca definidos anteriormente.

Primeiro teste se `SearchAgent` está funcionando, rodando:

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

O comando diz a `SearchAgent` para usar `tinyMazeSearch` como algoritmo de busca (está implementado em `search.py`). Pacman deve navegar com sucesso.
Agora é hora de implementar suas funções de busca para ajudar o Pacman a planejar suas rotas! Não se esqueça que o nó de busca deve conter não só o estado mas também uma maneira de reconstruir o caminho (plano) que leva ao estado objetivo.

Todas as suas funções de busca devem retornar uma lista de *ações válidas* que levam o agente do estado inicial ao estado objetivo.

Todos os algoritmos são bastante parecidos. DFS, BFS e A* são diferentes somente na forma de gerenciar a fronteira. Então, concentre-se primeiro em fazer o DFS funcionar e o resto ficará mais fácil.

Dê uma olhada nos tipos `Stack`, `Queue` e `PriorityQueue` disponíveis em `util.py`

Questão 1 (3 pontos)

Implemente o algoritmo de busca em profundidade (DFS) na função `depthFirstSearch` de `search.py`. Para que seu algoritmo seja *completo*, certifique-se que você implementou busca em grafo (aquela que evita expandir nós já visitados). Seu código deve achar rapidamente soluções para:

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

pacman.py -l mediumMaze -p SearchAgent

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

O tabuleiro do Pacman vai mostrar os estados explorados de forma sobreposta, na ordem que foram explorados (quanto mais vermelho, mais velho...).

Se você usar `Stack` como sua estrutura de dados, a solução encontrada para `mediumMaze` deve ter chegado a um custo total (comprimento) de 130 e ter expandido 146 nós. Para `bigMaze` deve ter chegado a um custo total de 210 e ter expandido 390 nós.

Questão 2 (2 pontos)

Implemente o algoritmo de busca em largura (BFS) na função `breadthFirstSearch` de `search.py`. Novamente, implemente busca em grafo e teste seu código como na questão anterior.

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

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

Você achou a solução de menor custo? Se não, tem algo errado com sua implementação.

Se o Pacman estiver se movendo muito lentamente, tente `--frameTime 0`.
Se você usar `Queue` como sua estrutura de dados, a solução encontrada para `mediumMaze` deve ter chegado a um custo total (comprimento) de 68 e ter expandido 269 nós. Para `bigMaze` deve ter chegado a um custo total de 210 e ter expandido 620 nós.

Questão 3 (2 pontos)

Apesar da busca em largura achar o menor caminho em número de passos, pode-se querer caminhos que sejam "melhores" em outros sentidos. Considere os labirintos mediumDottedMaze e mediumScaryMaze. Ao alterar a função de custo, podemos encorajar o Pacman a achar outros caminhos. Por exemplo, pode-se estimular áreas ricas em comida e evitar áreas assombradas.

Implemente o algoritmo de busca de custo uniforme na função uniformCostSearch do search.py. Você deverá notar comportamento bem sucedido nos seguintes cenários (agentes e funções de custo já estão implementadas para você, dê uma olhada no que elas fazem):

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

pacman.py -l mediumDottedMaze -p StayEastSearchAgent

pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```

Se você usar `PriorityQueue` como sua estrutura de dados, a solução encontrada para `mediumMaze` deve ter chegado a um custo total (comprimento) de 68 e ter expandido 269 nós. Para `bigMaze` deve ter chegado a um custo total de 210 e ter expandido 620 nós.

Questão 4 (2 pontos)

Implemente a busca em grafo A* na função `aStarSearch` de `search.py`. A* recebe uma função heurística como argumento. As heurísticas, por sua vez, têm dois argumentos: o estado do problema de busca e o problema propriamente dito. A função heurística `nullHeuristic` em `search.py` é um exemplo trivial.
Teste sua implementação do A* no problema original achando o caminho através do labirinto para uma posição fixa usando a heurística da distância de Manhattan (já disponível em `manhattanHeuristic` do `searchAgents.py`).

```
pacman.py -l mediumMaze -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

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

Se você usar `PriorityQueue` como sua estrutura de dados, a solução encontrada para `mediumMaze` deve ter chegado a um custo total (comprimento) de 68 e ter expandido 221 nós. Para `bigMaze` deve ter chegado a um custo total de 210 e ter expandido 549 nós.

Questão 5 (1 ponto)

O que acontece no `openMaze` para as várias estratégias? Apresente um relatório comparando-as. Explique os resultados.