Universidade Federal Fluminense
Instituto de Computação / Departamento de Ciência da Computação.
Inteligência Artificial

Trabalho 1 - Implementação de Algoritmos de Busca

Instruções

- Apresentação do código em sala de aula no dia 20.06
- Algum integrante do grupo deve enviar o código do trabalho e um relatório contendo as respostas para as perguntas (quando cabível) para o email alinepaes@ic.uff.br, até o dia 19.06-20hrs. Assunto: trabalho1-IA, constando os nomes dos integrantes do grupo
- Cada integrante do grupo deve enviar um relatório para o mesmo email acima (assunto: trabalho1-IA-relatorio-de-XXXX), constando sua porcentagem de participação no trabalho e a porcentagem de participação dos demais integrantes do grupo. Exemplo (supondo grupo de 3): se você fez metade do trabalho e cada um dos seus colegas contribuiu com 25%, reporte qual foi a sua contribuição correspondente aos 50% e no que seus colegas contribuíram com 25% cada.

Parte 1 (1.5 pts)
Escreva um programa que recebe como entrada duas páginas web e encontra um caminho de links de uma para a outra. Sua implementação deve considerar qual é a estratégia de busca mais adequada para esse caso e implementar somente ela.

Parte 2: original de School of Computer Science / Carnegie Mellon University / USA, projeto desenvolvido originalmente no Pacman Project, Universidade de Berkeley.

Introdução

- O objetivo do trabalho será programar algoritmos de busca aplicados ao cenário do Pacman. O Pacman deve encontrar caminhos no labirinto, tanto para chegar a um destino quando para coletar comida eficientemente. O código do trabalho contém vários arquivos Python, alguns devem ser modificados e outros que pode ser ignorados. O código está indo junto com este texto, no arquivo compactado.

- Arquivos que devem ser editados:

search.py - onde ficam os algoritmos de busca

searchAgents.py - Contém a classe searchAgent, 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.

- Arquivos que você pode querer olhar

pacman.py - O arquivo principal que roda os jogos de Pacman. Esse arquivo descreve o tipo GameState, que será usado nesse trabalho.
game.py - A lógica por trás do mundo do Pacman. Este arquivo descreve vários tipos auxiliares como AgentState, Agent, Direction e Grid.
util.py - Estruturas de dados úteis para implementar algoritmos de busca.

- Os demais arquivos correspondem a parte gráfica, interfaces de controle a partir do teclado e outros, que você pode ignorar.

Bem vindo ao Pacman

- Depois de baixar o código, descompactá-lo e entrar no diretório *search*, você pode jogar um jogo de Pacman digitando a seguinte linha de comando:

  python pacman.py

OBS: se aparecerem erros relativos a biblioteca TK, veja aqui como instalá-la.

- Todos os comandos para jogar no Pacman estão no arquivo commands.txt. Por exemplo, o Pacman sempre vai para oeste com o comando:

  python pacman.py --layout testMaze --pacman GoWestAgent
- Todas as opções do ambiente podem ser vistas ao digitar:

  python pacman.py -h
- Por exemplo, você pode trocar o layout do agente com o comando:

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

Encontrando comida em um ponto fixo usando algoritmos de busca

- No arquivo searchAgents.py, você encontrará 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 -- that's you job!. Primeiro, teste se o agente de busca está funcionando corretamente, executando:

  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 e nos slides de aula. Lembre-se

que um nó da busca deve conter não só o estado mas também toda a informação necessária para reconstruir o caminho 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 similares. Os algoritmos de busca em profundidade, busca em largura, busca de custo uniforme e A* diferem somente na ordem em que os nós são retirados da fronteira. Então o ideal é tentar implementar a busca em largura 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 fronteira. (Mas não é necessário implementar dessa forma). Verifique o código dos tipo Stack (pilha), Queue (fila) e PriorityQueue (fila com prioridade) que estão no arquivo util.py.

Passo 1 (1.5 pts) Implemente o algoritmo de busca em profundidade (DFS) na função *depthFirstSearch* do arquivo search.py. Para que a busca seja *completa*, implemente a versão graphSearch, para não expandir estados repetidos. 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 mostrará os estados explorados e a ordem em que eles foram explorados (vermelho mais forte significa que o estado foi explorado antes). **Responda**:

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? Essa é uma solução ótima? Senão, o que a busca em profundidade está fazendo de errado? Existe alguma diferença no tamanho retornado se os estados sucessores forem colocados na pilha na ordem dada pela função getSuccessors ou na ordem reversa?

Passo 2 (1 pt) Implemente o algoritmo de busca em largura (BFS), versão graphSearch na função *breadthFirstSearch* do arquivo search.py. 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

Dica: Se o Pacman se mover muito lentamente, tente a opção --frameTime 0. **Responda:**

A busca BFS encontra a solução ótima? Senão, verifique a sua implementação.

<u>Variando a função de custo</u>

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 labirintomediumScaryMaze. Mudando a função de custo, podemos encorajar o Pacman a encontrar caminhos diferentes. Por exemplo, podemos ter custos maiores para passar por

áreas com fantasmas e custos menores para passar em áreas com comida. O Pacman racional deve poder ajustar o seu comportamento de acordo com essas possibilidades.

Passo 3 (2 pts) Implemente o algoritmo de busca de custo uniforme (versão dada em sala de aula e na 3a edição do livro texto) na função *uniformCostSearch* do arquivo search.py. Teste seu código executando os comandos a seguir, onde as diferentes funções de custo são dadas:

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

Busca A*

Passo 4 (2 pts) Implemente a busca A*, de acordo com o livro-texto, na função *aStarSearch* do arquivo search.py. A busca A* recebe uma heurística como parâmetro. Heurísticas nessa implementação 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* 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

Responda:

Qual a diferença no tempo para encontrar uma solução e nos estados explorados da busca de custo uniforme e na A*?

Busca subótima

Algumas vezes, mesmo que a busca A* tenha uma boa heurística, encontrar a solução ótima é um problema difícil. Nesses casos, ainda podemos encontrar um caminho razoalvelmente bom, rapidamente. Você implementará um agente que sempre come a comida que está mais perto, o *ClosestDotSearchAgent,* que deverá estar em *searchAgents.py*. A função chave, que encontra a comida mas próxima, está faltando -- that's also you job.

Passo 5 (2 pts)

Implemente a função *findPathToClosestDot* em *searchAgents.py*. O agente resolve este labirinto de forma subótima em menos de 1 segundo. Teste com

python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5