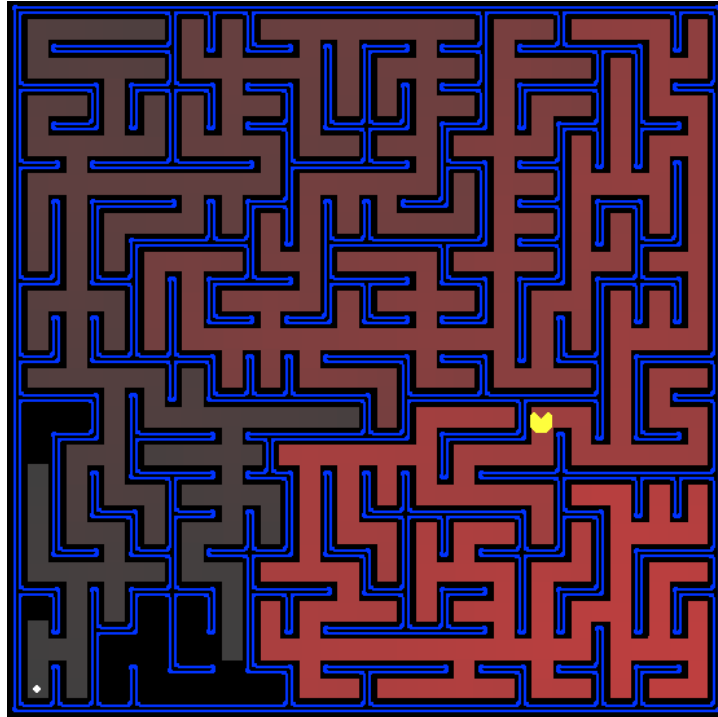


Trabalho 1: Busca no Pacman

Este trabalho é parte do Pacman Project desenvolvido na UC Berkeley disciplina CS188 – Artificial Intelligence. Tradução realizada pelo professor Eduardo Bezerra (ebezerra@cefet-rj.br, CEFET/RJ).



Introdução

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.

Este projeto inclui um autoavaliador (*autograder*) para você avaliar suas respostas em sua máquina. Isso pode ser executado com o comando a seguir:

```
python autograder.py
```

O código fornecido nesse trabalho consiste de diversos arquivos Python, alguns dos quais você terá que ler e entender para fazer o trabalho. O código está no Moodle, arquivo **search.zip**.

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 nesse trabalho.
game.py	A lógica 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.
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
autograder.py	Projeto autograder
testParser.py	Analisa arquivos de solução e teste autograder
testClasses.py	Classes de teste de classificação automática geral
test_cases	Diretório contendo os casos de teste para cada questão
searchTestClasses.py	Classes de teste de avaliação automática específicas do Projeto 1

Especificações da entrega

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

1. Os arquivos **search.py** e **searchAgents.py** alterados durante esse trabalho. Não entregue outros arquivos fonte além desses dois
2. Um relatório (necessariamente em formato PDF). Nesse relatório, explicita as respostas às perguntas listadas abaixo. O relatório deve apresentar análise e explicação dos resultados obtidos, assim como deve apresentar descrições das partes relevantes do código implementado.

Este trabalho é individual. Seu trabalho deve ser submetido pelo Moodle até o prazo final estabelecido na página do curso.

Bem-vindo ao Pacman

Depois de baixar o código (**search.zip**), descompactá-lo e entrar no diretório *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. Trata-se de 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
```

Repare que o *script* **pacman.py** tem opções de comando 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
```

Todos os comandos que aparecem aqui também estão no arquivo **commands.txt**, e podem ser copiados e colados para execução.

Pseudocódigo da busca em árvore

Para as implementações dos algoritmos de busca nas questões de Q1 até Q3, você implementará o seguinte pseudocódigo:

```
Algoritmo GRAPH_SEARCH:

frontier = {startNode}
expanded = {}
while frontier is not empty:
    node = frontier.pop()
    if isGoal(node):
        return path_to_node
    if node not in expanded:
        expanded.add(node)
        for each child of node's children:
            frontier.push(child)
return failed
```

Q1 (4 pontos) - Encontrando comida usando DFS

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** disponível no fim deste documento. 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 implementado na função denominada `tinyMazeSearch`, que está implementada 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 em sua implementação. Os algoritmos de busca em profundidade (DFS), busca em extensão (BFS), busca de custo uniforme (BFS) 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).

Dica: Dê uma olhada no código dos tipos `Stack` (pilha), `Queue` (fila) e `PriorityQueue` (fila com prioridade) que estão no arquivo **util.py**.

Implemente o algoritmo de busca em profundidade (DFS) na função `depthFirstSearch` do arquivo **search.py**. Para que a busca seja *completa*, implemente a versão do DFS que não expande estados já visitados.

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 nas iterações iniciais). 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). Essa é uma solução ótima? Senão, discuta o que a busca em profundidade está fazendo de errado?

Agora, verifique a corretude de sua implementação por meio do comando a seguir:

```
python autograder.py -q q1
```

Q2 (4 pontos) - BFS

Agora 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 os comandos a seguir:

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

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

Dica: se a execução dos comandos acima for muito lenta em sua máquina, acrescente a opção `--frameTime 0`.

Se o seu código foi escrito de maneira correta, ele deve funcionar também para o quebra-cabeças de 8 peças sem modificações. Teste isso por meio do comando a seguir:

```
python eightpuzzle.py
```

Quantas ações compõem a solução encontrada pelo BFS?

Agora, verifique a corretude de sua implementação por meio do comando a seguir:

```
python autograder.py -q q2
```

Q3 (4 pontos): A* search

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 recebem dois parâmetros: um estado do problema de busca (o parâmetro principal), e o próprio objeto `Problem` (para consulta). 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 até 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 do que a busca de custo uniforme (549 vs. 620 nós de busca expandidos na nossa implementação, mas a aplicação de desempates pode produzir valores um pouco diferentes). O que acontece em `openMaze` para as várias estratégias de busca?

Agora, verifique a corretude de sua implementação por meio do comando a seguir:

```
python autograder.py -q q3
```

Q4 (3 pontos): Encontrando todos os cantos

O verdadeiro poder do A* só ficará aparente com um problema de busca mais desafiador. Agora, é hora de formular um novo problema e projetar uma heurística para ele.

Nos labirintos de canto, existem quatro pílulas, uma em cada canto. O novo problema de busca consiste em encontrar o caminho mais curto através do labirinto que toca todos os quatro cantos (independente de se o labirinto realmente tem comida em um canto ou não). Observe que, em alguns labirintos como o `tinyCorners`, o caminho mais curto nem sempre vai para a comida mais próxima primeiro! Dica: o caminho mais curto por `tinyCorners` leva 28 etapas.

Observação: certifique-se de responder à Q2 antes de trabalhar na Q4, porque esta última se baseia em sua resposta à Q2.

Implemente o problema de busca `CornersProblem` em `searchAgents.py`. Você precisará escolher uma representação de estado que codifique todas as informações necessárias para detectar se todos os quatro cantos foram alcançados. Use os comandos a seguir para testar seu agente de busca:

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

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

Para receber todos os pontos desta parte do trabalho, você precisa definir uma representação de estado abstrato que não codifique informações irrelevantes (como a posição dos fantasmas, onde há comida extra, etc.). Em particular, não use um `Pacman GameState` como um estado de busca. Seu código ficará muito lento (e também errado) se você fizer isso.

Dica 1: as únicas partes do estado do jogo que você precisa fazer referência em sua implementação são a posição inicial do Pacman e a localização dos quatro cantos.

Dica 2: ao codificar a expansão, certifique-se de adicionar cada nó filho à lista de filhos com custo `getActionCost` e próximo estado `getNextState`. Observe que você também precisará codificar a função `getNextState`.

Dica 3: você deve armazenar estados do formato da tupla `((x, y), ____)`. Você precisará decidir quais informações armazenar no espaço em branco.

Como base para comparação, nossa implementação de `breadthFirstSearch` expande pouco menos de 2.000 nós de busca no `mediumCorners`. No entanto, as heurísticas (usadas com a busca A*) podem reduzir a quantidade de busca necessária.

Agora, verifique a corretude de sua implementação por meio do comando a seguir:

```
python autograder.py -q q4
```

Q5 (3 pontos): Heurística para o Problema dos Cantos

Observação: certifique-se de completar a Q3 antes de trabalhar na Q5, porque a Q5 se baseia em sua resposta para a Q3.

Implemente uma heurística consistente e não trivial para `CornersProblem` em `cornersHeuristic`.

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

Nota: AStarCornersAgent é um atalho para

```
-p SearchAgent -a  
fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic
```

Admissibilidade vs. Consistência: Lembre-se de que as heurísticas são apenas funções que pegam estados de busca e retornam números que estimam o custo até o (estado) objetivo mais próximo. Heurísticas mais eficazes retornarão valores mais próximos dos custos reais para o objetivo. Para serem admissíveis, os valores heurísticos devem ser limites inferiores no custo do caminho mais curto real para o objetivo mais próximo (e não negativos). Para ser consistente, ele deve adicionalmente sustentar que se uma ação tem custo c , então realizar essa ação só pode causar uma queda na heurística de no máximo c .

Lembre-se de que a admissibilidade não é suficiente para garantir a correção na busca em grafo (*graph search*) - você precisa de uma condição mais forte de consistência. No entanto, as heurísticas admissíveis geralmente também são consistentes, especialmente se derivadas de relaxamentos de problemas. Portanto, geralmente é mais fácil começar fazendo um brainstorming de heurísticas admissíveis. Depois de ter uma heurística admissível que funcione bem, você pode verificar se ela também é consistente. A única maneira de garantir consistência é com uma prova. A consistência pode ser verificada para uma heurística verificando se para cada nó que você expandir, seus nós filhos são iguais ou menores em valor f . Se essa condição for violada para qualquer nó, sua heurística é inconsistente. Além disso, se UCS (A* com a heurística nula) e A* retornarem caminhos de comprimentos diferentes, sua heurística será inconsistente. Atente para isso!

Heurísticas não triviais: As heurísticas triviais são aquelas que retornam zero em todos os lugares (UCS) e a heurística que calcula o custo de conclusão verdadeiro. O primeiro não vai lhe poupar tempo, enquanto o último vai expirar o autograder. Você quer uma heurística que reduza o tempo total de computação, embora para esta atribuição o autograder verificará apenas as contagens de nós (além de impor um limite de tempo razoável).

Avaliação: sua heurística deve ser uma heurística consistente não trivial não negativa para receber qualquer ponto. Certifique-se de que sua heurística retorne 0 em cada estado de meta e nunca retorne um valor negativo. Dependendo de quantos nós sua heurística se expande, você será avaliado:

Quantidade de nós expandidos	Pontos
Mais do que 2000	0/3
No máximo 2000	1/3
No máximo 1600	2/3
No máximo 1200	3/3

Atenção: se sua heurística for inconsistente, você não receberá os pontos desta parte do trabalho.

Agora, verifique a corretude de sua implementação por meio do comando a seguir:

```
python autograder.py -q q5
```

Q6 (4 pontos): Comendo todas as pílulas

Agora você irá considerar e resolver um problema de busca 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`.

Você deve notar que o UCS começa a ficar mais lento, mesmo para o aparentemente simples `tinySearch`. Como referência, nossa implementação leva 2,5 segundos para encontrar um caminho de comprimento 27 após expandir 5057 nós de busca.

Observação: certifique-se de completar a Q3 antes de trabalhar na Q6, porque a Q6 se baseia em sua resposta para a Q3.

Complete a função `foodHeuristic` em `searchAgents.py` com uma heurística consistente para `FoodSearchProblem`. Em seguida, teste seu agente no problema `trickySearch`:

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

Como referência, nossa implementação com UCS encontra a solução ótima em cerca de 13 segundos, explorando aproximadamente 16.000 nós.

Qualquer heurística consistente não trivial não negativa receberá 1 ponto. Certifique-se de que sua heurística retorne 0 em cada estado de meta e nunca retorne um valor negativo. Dependendo de quantos nós sua heurística se expande, você obterá pontos adicionais:

Quantidade de nós expandidos	Pontos
mais de 15.000	1/4
no máximo 15.000	2/4
no máximo 12.000	3/4
no máximo 9.000	4/4 (crédito total; médio)
no máximo 7.000	5/4 (crédito extra opcional; difícil)

Agora, verifique a corretude de sua implementação por meio do comando a seguir:

```
python autograder.py -q q6
```

Q7 (3 pontos): Busca subótima

Às vezes, mesmo com A* e uma boa heurística, é difícil encontrar o caminho ótimo através de todos os pontos. Nesses casos, ainda gostaríamos de encontrar um caminho razoavelmente bom, rapidamente. Nesta seção, você escreverá um agente que sempre come avidamente o ponto mais próximo. `ClosestDotSearchAgent` já está implementado para você em `searchAgents.py`, mas está faltando uma função-chave que encontra um caminho para o ponto mais próximo.

Implemente a função `findPathToClosestDot` em `searchAgents.py`. Nosso agente resolve este labirinto (abaixo do ideal!) em menos de um segundo com um custo de caminho de 350:

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

Dica: a maneira mais rápida de concluir `findPathToClosestDot` é preencher `AnyFoodSearchProblem`, que está faltando seu teste de objetivo. Em seguida, resolva esse problema com uma função de pesquisa apropriada. A solução deve ser muito curta!

Seu agente `ClosestDotSearchAgent` nem sempre encontrará o caminho mais curto possível pelo labirinto. Certifique-se de entender o porquê e tente criar um pequeno exemplo em que ir repetidamente para o ponto mais próximo não resulte em encontrar o caminho mais curto para comer todos os pontos.

Agora, verifique a corretude de sua implementação por meio do comando a seguir:

```
python autograder.py -q q7
```

Glossário

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.

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

`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.