Projeto 1: Busca Não Informada na Resolução de Labirintos

Autor: Marcos Antonio Teles de Castilhos

Disciplina: FGA0221 - Inteligência Artificial

Professor: Fabiano Araujo Soares, Dr.

1. Introdução

Este projeto implementa e compara visualmente dois algoritmos fundamentais de busca não informada (cega):

Busca em Largura (BFS - Breadth-First Search) e a **Busca em Profundidade (DFS - Depth-First Search)**.

O objetivo é encontrar um caminho entre um ponto inicial ('S') e um ponto final ('E') em um labirinto bidimensional definido textualmente.

A característica principal da busca não informada é que ela explora o espaço de estados sem qualquer conhecimento prévio sobre a proximidade do objetivo, utilizando apenas a definição do problema. O projeto inclui uma interface gráfica que anima o processo de exploração de cada algoritmo, permitindo uma análise clara de suas estratégias e características.

2. Formulação do Problema

O problema de encontrar um caminho em um labirinto foi modelado como um problema de busca, seguindo a formulação padrão:

- **Estados:** Cada estado é representado por uma tupla **(linha, coluna)** indicando uma posição (célula) dentro do labirinto.
- **Estado Inicial:** A posição da célula marcada com 'S' no labirinto. Identificado pela função **encontrar_posicao**.
- **Ações (Função Sucessora):** A partir de um estado **(linha, coluna),** as ações possíveis são mover-se para uma célula adjacente (cima, baixo, esquerda,

direita) que não seja uma parede ('#') e esteja dentro dos limites do labirinto. Implementado na função obter_vizinhos. A lógica usa dr (delta-linha) e dc (delta-coluna) para calcular as coordenadas vizinhas.

- **Teste de Objetivo:** A busca termina com sucesso se o estado atual for a posição da célula marcada com 'E'. Verificado dentro do loop principal das funções de busca.
- **Custo do Caminho:** Assume-se um custo uniforme de 1 para cada movimento entre células adjacentes.

3. Algoritmos Implementados

O script permite escolher entre dois algoritmos clássicos de busca não informada:

3.1. Busca em Largura (BFS)

- **Estratégia:** Explora o labirinto de forma sistemática, nível por nível. Garante que todos os nós a uma profundidade **d** sejam visitados antes de qualquer nó na profundidade **d+1**.
- **Estrutura de Dados:** Utiliza uma **Fila (Queue)** para gerenciar a fronteira de nós a serem explorados. No código, isso é implementado usando **collections.deque** e a operação **popleft()**, que remove o elemento mais antigo da fila (FIFO - First-In, First-Out).
- **Propriedades:**
    - **Completo:** Sim, sempre encontrará uma solução se ela existir.
    - **Ótimo:** Sim, encontra a solução com o menor número de passos (caminho mais curto), pois o custo de cada passo é uniforme.
    - **Complexidade:** Requer espaço de memória exponencial no pior caso, pois precisa armazenar todos os nós da fronteira.
- **Visualização:** A animação mostra a exploração se expandindo em "ondas" a partir do ponto inicial.

3.2. Busca em Profundidade (DFS)

- **Estratégia:** Explora um caminho o mais fundo possível antes de retroceder (backtrack) para tentar alternativas.
- **Estrutura de Dados:** Utiliza uma **Pilha (Stack)** para gerenciar a fronteira. No código, isso é implementado usando uma lista Python padrão e a operação

pop(), que remove o elemento mais recentemente adicionado (LIFO - Last-In, First-Out).

- **Propriedades:**
    - **Completo:** Não é completo em grafos infinitos, mas na nossa implementação com um labirinto finito e um conjunto de **visitados**, torna-se completo na prática.
    - **Ótimo:** Não, não garante encontrar o caminho mais curto. Ele retorna a primeira solução encontrada.
    - **Complexidade:** Requer muito menos memória que o BFS, apenas o necessário para armazenar o caminho atual e os nós na pilha.
- **Visualização:** A animação mostra a exploração "mergulhando" em um corredor até atingir um beco sem saída ou o objetivo, e depois retrocedendo.

4. Implementação e Visualização Gráfica

- **Função Principal: buscar_no_labirinto_nao_informado** contém a lógica central, alternando entre BFS e DFS com base na escolha do usuário.
- **Graph Search:** Ambos os algoritmos utilizam um dicionário **visitados** para armazenar as células já exploradas e sua profundidade (**g_cost**). Isso evita ciclos e a re-exploração de estados, tornando a busca mais eficiente (característica de uma *Graph Search*).
- **Visualização (matplotlib):**
    - **preparar_visualizacao_grafica**: Configura a janela e o esquema de cores.
    - **visualizar_passo_grafico**: Converte o estado atual da busca (fronteira, visitados, caminho atual) em uma matriz numérica e a desenha usando cores. A legenda é:
        - **Preto:** Parede (#)
        - **Branco:** Caminho Livre ( )
        - **Verde:** Início (S)
        - **Vermelho:** Fim (E)
        - **Laranja:** Fronteira (nós na fila/pilha esperando para serem explorados)
        - **Cinza Claro:** Visitado (nós já explorados)

        - **Azul Claro:** Caminho Atual (o caminho específico sendo expandido no momento)
    - **Texto g=...:** Indica o custo (número de passos) para chegar àquela célula a partir do início ('S').

5. Como Usar o Programa

1. **Pré-requisitos:** Certifique-se de ter o Python 3 instalado e a biblioteca matplotlib. Se necessário, instale com:
2. pip install matplotlib
3. **Execução:** Abra um terminal, navegue até o diretório onde o arquivo .py foi salvo e execute o comando:

   python busca-n-informada.py

4. **Escolha do Algoritmo:** O programa solicitará que você digite **bfs** ou **dfs** e pressione Enter.
5. **Visualização:** Uma janela gráfica será aberta, mostrando a animação passo a passo da busca escolhida. Observe a diferença na estratégia de exploração entre BFS e DFS.
6. **Resultado:** Após a animação (ou se você fechar a janela), o programa imprimirá o labirinto com o caminho final encontrado (*) no terminal, juntamente com o número de passos.

6. Conclusão

Este projeto demonstra com sucesso a implementação e o comportamento dos algoritmos BFS e DFS. A visualização gráfica permite uma comparação direta, evidenciando a exploração sistemática e a garantia de otimalidade (em passos) do BFS, em contraste com a eficiência de memória e a natureza não ótima do DFS. O uso do conjunto de **visitados** implementa uma *Graph Search*, essencial para a eficiência em problemas com ciclos ou estados repetidos, como um labirinto.