Relatório do Projeto 01 - Busca em Labirinto

Luís Fernando Almeida - 8102
SIN 323 - Inteligência Artificial

3 de dezembro de 2024

Conteúdo

1 Objetivo

O objetivo deste projeto é desenvolver um agente capaz de resolver um labirinto utilizando algoritmos de busca baseados em grafos, como a Busca em Largura, a Busca em Profundidade e a Busca A*. O agente deve ser projetado de forma modular, permitindo alterações nas características do ambiente, sensores e atuadores, além de realizar a verificação de estados visitados. A meta principal é que o agente encontre o caminho do estado inicial até o estado final do labirinto utilizando o menor número possível de movimentos, conforme especificado pelas regras do problema.

2 Tecnologias utilizadas

Para o desenvolvimento desse projeto, foram empregadas as seguintes tecnologias:

- ***Python***: A linguagem de programação *Python* foi escolhida devido à sua vasta quantidade e variedade de *modules* (bibliotecas) que facilitaram todo o processo de desenvolvimento. Por exemplo, a bibliioteca *Pygame* foi utilizada para criar a interface do labirinto com o usuário, permitindo que o usuário visualize o labirinto, o agente e o caminho, fazendo alterações, caso necessário.
- **LaTeX**: O sistema de preparação de documentos LaTeX foi utilizado para confeccionar esse relatório.
- ***GitHub***: A plataforma de hospedagem de código fonte *GitHub* foi utilizada para versionar e disponibilizar o código fonte desse projeto. O repositório estava privado até a data de entrega deste relatório, sendo essa dia 03 de dezembro de 2024.

3 Estrutura do projeto

O projeto está estruturado da seguinte forma:

- ***agent.py***:
  1. Define a classe *Agent*.
  2. Define os métodos utilizados pelo agente, sendo eles:
     (a) *move*, que é um trecho inutilizado de código, visto que a lógica de movimento do agente foi substituída de comandos de movimento para simplesmente teleportar para as coordenadas encontradas pelos algoritmos de busca.
     (b) *draw*, que lida com a representação do agente na tela.

   (c) *get_neighbors*, que é o método pelo qual o agente analisa seus arredores e define quais movimentos são possíveis.

3. Define e implementa os algoritmos de busca utilizados pelo agente, sendo eles:
   (a) Busca em Largura (*Breadth First Search*).
   (b) Busca em Profundidade (*Depth First Search*).
   (c) Busca A* (*A-Star Search*).

   A lógica utilizada por esses algoritmos será abordada na seção 4.

- ***main.py***:

1. O arquivo *main.py* define variáveis importantes para o funcionamento do programa, tais como o tamanho da tela, os pontos de início e objetivo do agente, além de chamar todas as funções responsáveis pela lógica da execução e pela renderização na tela.

- ***maze_layouts.py***:

1. O arquivo *maze_layouts.py* é apenas um arquivo auxiliar que contém algumas definições de matrizes que são utilizadas como labirintos no programa.

- ***maze.py***:

1. Define os métodos utilizados pelo labirinto (cenário), sendo eles:
   (a) *draw*, que lida com a representação do labirinto.
   (b) *draw_path*, que lida com a representação do taminho tomado pelo agente.
   (c) *draw_start_goal*, que lida com a representação do ponto inicial e final do agente.

- ***menu.py***:

1. Define os métodos que realizam a interação com o usuário, sendo eles:
   (a) *draw_menu*, que exibe o menu inicial para o usuário.
   (b) *show_menu*, que exibe o menu inicial para o usuário e permite alterações desejadas, tais como:
      i. Qual algoritmo utilizar.
      ii. Qual caminho será destacado.
      iii. Qual a velocidade do agente.
   (c) *save_report*, que gera um arquivo *.txt* com informações sobre o algoritmo que foi executado.

- **Pasta *extras***:

    1. Contém o arquivo *maze_editor.py*, que é uma utilidade para gerar labirintos para serem utilizados no programa principal, e contém a pasta *mazes*, que salva os arquivos gerados pelo *maze_editor.py*.

- **Pasta *txt***:

    1. Armazena os relatórios gerados pela função *save_report*.

4 Lógica dos algoritmos

Nesta seção, vamos detalhar a lógica e os passos principais dos algoritmos de busca implementados: busca em largura (*Breadth-First Search*), busca em profundidade (*Depth-First Search*) e A* (*A-Star Search*).

4.1 Busca em Largura (*BFS*)

O algoritmo de busca em largura explora o labirinto nível por nível, garantindo que o primeiro caminho encontrado seja sempre o mais curto. Ele funciona da seguinte forma:

1. **Inicializações:**
    - Cria uma fila (`queue`) contendo apenas o nó inicial (`start`).
    - Inicializa um conjunto de nós visitados (`visited`) com o nó inicial.
    - Mantém um dicionário (`came_from`) para reconstruir o caminho ao final.
    - Cria uma lista (`all_visited`) para registrar todos os nós visitados.
    - Define uma variável `steps` para contar o número de passos realizados.
2. **Exploração:**
    - Enquanto houver nós na fila, remova o primeiro nó (`current`) e registre-o como visitado.
    - Verifique se o nó atual é o objetivo (`goal`). Caso positivo:
        - Reconstrua o caminho percorrendo o dicionário `came_from`.
        - Retorne o caminho, o número de passos e todos os nós visitados.
    - Caso contrário, obtenha os vizinhos do nó atual e adicione à fila os que ainda não foram visitados, registrando a origem no dicionário `came_from`.
3. **Resultado:**
    - Se a fila esvaziar sem encontrar o objetivo, o algoritmo retorna `None`, indicando que o objetivo é inalcançável.

4.2 Busca em Profundidade (*DFS*)

A busca em profundidade explora o labirinto priorizando um caminho até o fim antes de retroceder. O algoritmo utiliza uma estrutura de pilha para simular o comportamento de recursão. Os passos são os seguintes:

1. **Inicializações:**
   - Cria uma pilha (`stack`) contendo apenas o nó inicial (`start`).
   - Inicializa um conjunto de nós visitados (`visited`) com o nó inicial.
   - Mantém um dicionário (`came_from`) para reconstruir o caminho ao final.
   - Cria uma lista (`all_visited`) para registrar todos os nós visitados.
   - Define uma variável `steps` para contar o número de passos realizados.
2. **Exploração:**
   - Enquanto houver nós na pilha, remova o último nó (`current`) e registre-o como visitado.
   - Verifique se o nó atual é o objetivo (`goal`). Caso positivo:
     - Reconstrua o caminho percorrendo o dicionário `came_from`.
     - Retorne o caminho, o número de passos e todos os nós visitados.
   - Caso contrário, obtenha os vizinhos do nó atual e adicione à pilha os que ainda não foram visitados, registrando a origem no dicionário `came_from`.
3. **Resultado:**
   - Se a pilha esvaziar sem encontrar o objetivo, o algoritmo retorna `None`, indicando que o objetivo é inalcançável.

4.3 Algoritmo A* (*A-Star*)

O A* é um algoritmo heurístico que utiliza uma função de custo estimada para guiar a busca de forma eficiente. A função de custo escolhida foi a Distância Euclidiana, sendo composta por dois valores:

- `g_score`: Custo real do nó inicial até o nó atual.
- `f_score`: Custo estimado do nó inicial até o objetivo, somando `g_score` e a heurística.

Os passos do algoritmo são:

1. **Inicializações:**

- Cria uma fila de prioridade (`open_set`) contendo o nó inicial com prioridade 0.
- Inicializa os dicionários `g_score` e `f_score` com valores iniciais para o nó inicial.
- Mantém um dicionário (`came_from`) para reconstruir o caminho ao final.
- Cria uma lista (`all_visited`) para registrar todos os nós visitados.
- Define uma variável `steps` para contar o número de passos realizados.

2. **Exploração:**
   - Enquanto a fila de prioridade não estiver vazia, remova o nó com menor `f_score`.
   - Verifique se o nó atual é o objetivo (`goal`). Caso positivo:
     - Reconstrua o caminho percorrendo o dicionário `came_from`.
     - Retorne o caminho, o número de passos e todos os nós visitados.
   - Caso contrário, obtenha os vizinhos do nó atual. Para cada vizinho:
     - Calcule o custo `tentative_g_score`.
     - Se o custo for menor que o registrado, atualize os valores `g_score` e `f_score`, e adicione o vizinho à fila de prioridade.
3. **Resultado:**
   - Se a fila esvaziar sem encontrar o objetivo, o algoritmo retorna `None`, indicando que o objetivo é inalcançável.

5 Análise de desempenho

Para conduzir os testes, foi analisado a mesma estrutura de labirinto para os três algoritmos de busca.

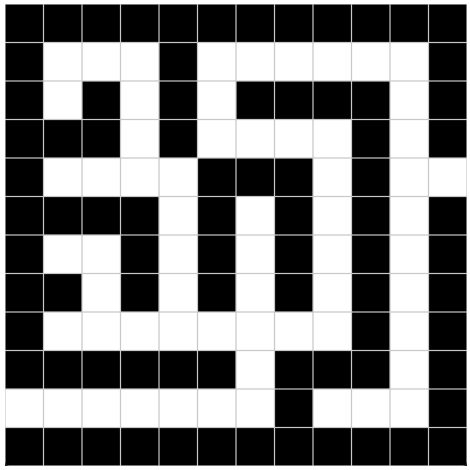

Figura 1: Labirinto definido para testes.

O agente começa na posição (5, 12) (Vermelho), e tem como objetivo chegar na posição (11, 1) (Verde).

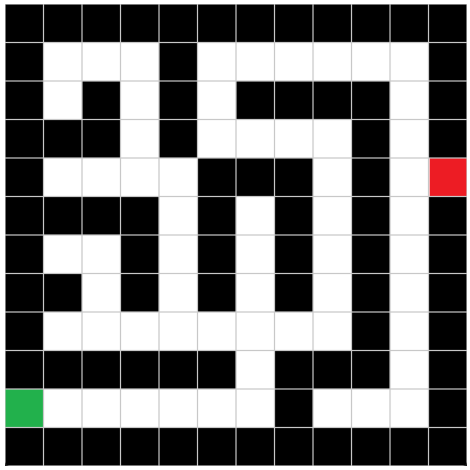

Figura 2: Labirinto com a posição inicial e o objetivo destacados.

Com essas informações definidas, podemos começar os testes utilizando os 3 algoritmos de busca definidos: Busca em Largura, Busca em Profundidade e Busca A*.

5.1 Busca em Largura (*BFS*)

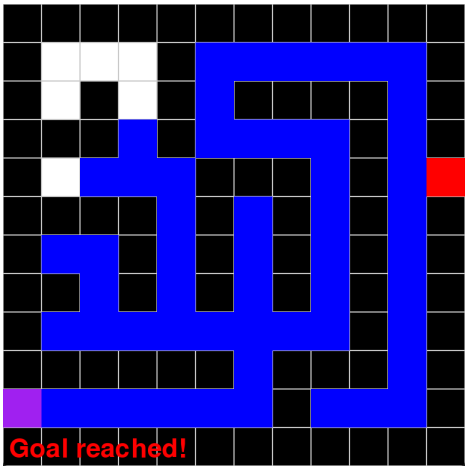

Figura 3: Caminho explorado pela busca em largura.

A busca em largura executou um total de 55 passos até encontrar o caminho.

5.2 Busca em Profundidade (*DFS*)

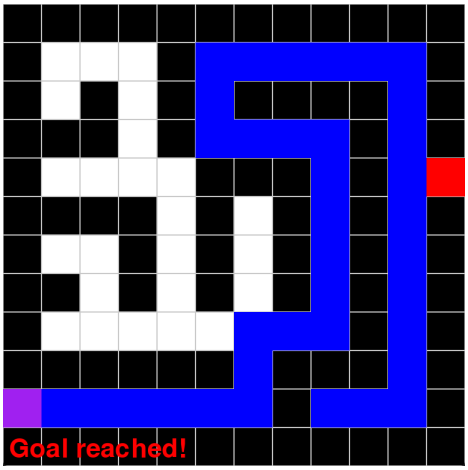

Figura 4: Caminho explorado pela busca em profundidade.

A busca em profundidade executou um total de 37 passos até encontrar o caminho.

5.3 Algoritmo A* (*A-Star*)

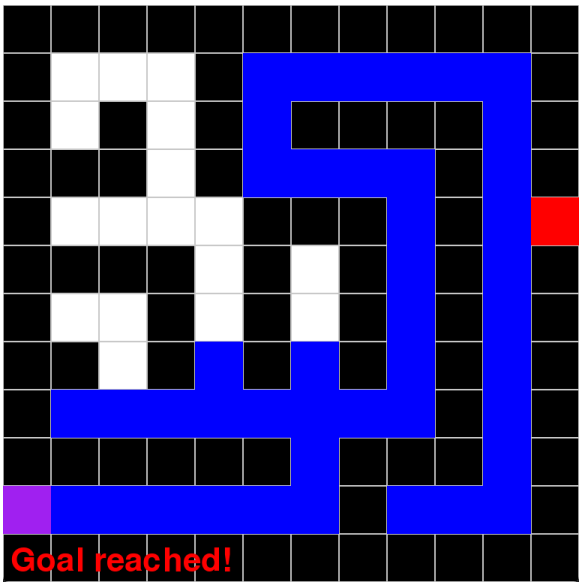

Figura 5: Caminho explorado pelo algoritmo A*.

O algoritmo A* executou um total de 44 passos até encontrar o caminho.

6 Conclusão

Com base nos testes realizados nos três algoritmos de busca (Busca em Largura, Busca em Profundidade e A*), foi possível observar as diferenças de desempenho e as características de cada abordagem.

Busca em Largura (*BFS*): A *BFS* apresentou um número maior de passos explorados (55). Esse comportamento é esperado, uma vez que a busca em largura explora todos os nós em uma camada antes de avançar para a próxima, o que a torna completa e ótima, mas, em alguns casos, mais lenta.

Busca em Profundidade (*DFS*): A *DFS* apresentou um desempenho mais rápido (37 passos), explorando menos nós que a *BFS*. Contudo, ela não garante o caminho mais curto, dependendo do ponto de partida e da estrutura do labirinto. Este resultado destaca a eficiência da *DFS* em alguns casos, mas também não podemos esquecer de sua limitação quanto à otimização da solução.

Algoritmo A* (*A-Star*): O A* realizou um total de 44 passos até encontrar o caminho. Isso demonstra o equilíbrio entre a exploração e a heurística, que guia a busca em direção ao objetivo de forma mais eficiente que a *BFS*, mas mantendo a garantia de encontrar o caminho mais curto. O uso de heurísticas torna o A* uma solução robusta para este tipo de problema.