Inteligência Artificial - Busca no Labirinto (Não Informada e Informada)

Mesquita S. Arthur[1], Costa V. N. João[2]

[1]*8º Período de Engenharia de Computação, CEFET-MG Campus V*
Divinópolis, MG, Brasil
arthur.santana.mesquita@gmail.com
[2]*4º Período de Engenharia de Computação, CEFET-MG Campus V*
Divinópolis, MG, Brasil
joaovnevescosta@hotmail.com

Abstract. Este trabalho tem como objetivo implementar algoritmos de busca de forma a solucionar um labirinto fornecido através de um arquivo externo. A atividade foi realizada em Outubro de 2025 para a disciplina de Inteligência Artifical, no curso de Engenharia de Computação do CEFET-MG - Campus V.

Keywords: Algoritmos, IA, Busca

1 Introdução

Foram implementados e comparados dois algoritmos de busca não informada (BFS, DFS) e dois algoritmos de busca informada (Gulosa e A*) aplicados ao problema do labirinto existente no arquivo de texto `/data/maze.txt`. Para tal foram necessárias funções para ler e interpretar o arquivo de entrada, transformando-o em uma lista de adjacência, a implementação de 2 funções heurísticas a serem comparadas nos algoritmos de busca informada, além do monitoramento para diversas medidas de desempenho nos algoritmos e seu armazenamento em um arquivo externo. A linguagem selecionada para as implementações foi Python 3.12

2 Implementação

2.1 Estrutura Geral

Foram construídos **4 scripts em Python**:

- **Adj.py**
  Lê o arquivo `/data/maze.txt` e cria uma lista de adjacência que será utilizada para a execução do algoritmo.
- **Heuristics.py**
  Possui a implementação das funções heurísticas que serão utilizadas para definir os parâmetros das funções de busca (Distância Euclidiana e Distância de Manhattan)
- **Search.py**
  Contém as funções de busca a serem utilizadas (bfs, dfs, Busca Gulosa, A*)
- **Maze.py**
  Contém a função `main`, além de outras funções que geram os arquivos de saída. Deste arquivo, os outros são chamados automaticamente.

Para executar o programa em *python* é necessário a instalação das bibliotecas *matplotlib*, *colorama* e *memory_profiler* através do seguinte comando do terminal:

```
pip install colorama matplotlib memory_profiler
```

E o seguinte script realiza a execução dos algoritmos a partir da raiz do projeto:

```
python src/Maze.py
```

2.2 Maze.txt

Arquivo contendo o Labirinto utilizado nos testes.

```
##################################################
#S....#.......#.........#.#.....................##
#####.#.#.#####.###.###.#.#.#####.##############.##
#.#...#.#...#.....#...#.#.#.....#.#...#...#...#.##
#.#.###.###.#.#######.#.#.#####.#.#.#.#.#.#.#.#.##
#...#.....#...#.......#.#...#...#.#.#...#...#...##
#.#######.#####.#######.#.###.###.#.##############
#.....#...#...#.....#.....#...#...#.#...........##
#####.###.###.#####.#.#####.###.###.#.#########.##
#...#...#.#.....#...#.#.....#.#.#.#...#.......#.##
#.#.###.#.#.###.#.#####.#####.#.#.###########.#.##
#.#...#.#...#...#.#.....#.#...#.#.............#.##
#.###.#.#####.###.#.#####.#.#.#.#.#####.###.###.##
#...#...#...#.#.#.#...#.....#.#.#.#.....#.#.#...##
###.#####.#.#.#.#.###.#.#####.#.#.#.#####.#.#.####
#.#...#...#.#...#...#.#...#...#.#.#.#...#...#...##
#.###.#####.###.###.#.#####.###.#.#.#.#.#.#####.##
#...#.#...#.#...#.#...#.....#...#.#.#.#.#.#.....##
#.###.#.#.#.#.###.#####.#####.###.#.#.#.#.#.######
#.#...#.#...#.#.........#.....#...#.#.#...#.#...##
#.#.###.###.#.#####.#.###.#####.###.#######.###.##
#.#...#...#.#.......#.#...#...#...#.........#...##
#.###.#.#.###########.#.#####.###.###########.#.##
#...#.#.#.#.....#.....#.....#...#.#...........#.##
###.#.###.#.###.#.#########.###.#.###.#######.#.##
#...#...#...#.#...#.....#...#...#...#.#.......#.##
#.#####.#.###.#########.#.###.#####.###.#######.##
#.#.....#.#.............#.#.......#.....#.....#.##
#.#.#####.#####.#######.#.#######.#######.###.#.##
#.#.....#.....#.#.#...#.#.#.........#.......#.#.##
#.#####.#####.#.#.#.#.#.#.#.#.#######.#######.#.##
#.....#.....#...#...#.#.#.#.#.#.#.....#.....#.#.##
#####.#####.#########.#.#.#.###.#######.###.#.#.##
#.....#...#.........#...#.#.....#.....#.#...#.#.##
#.###.#.###########.###.#.###.#######.#.#.###.#.##
#.#.#.#...........#...#.#...#.........#.#...#.#.##
#.#.#.###.###.#######.#.###.#####.#####.###.#.#.##
#.#...#...#...#.......#.#.#.#...#.#.....#.....#.##
#.#.###.#.#####.#######.#.#.#.#.#.#.###########.##
#.#...#.#.#.....#.....#...#.#.#...#.#.....#...#.##
#.###.#.#.#.#####.###.###.#.#.#####.#.###.#.#.#.##
#...#...#.#...#.....#...#.#.#.#...#...#.#.#.#...##
#.#.###.#####.#####.###.#.#.#.###.#####.#.#.######
#.#.#...#...#.....#...#...#.#...#...#...#.#.....##
###.#####.#.#####.#######.#.###.#.#.#.#.#.#####.##
#...#.....#.....#.......#.#...#...#.#.#.#...#...##
#.###.########.#######.#####.#####.###.###.#.####
#............#.............#..........#.....##
##################################################.G#
##################################################
```

2.3 Adj.py

`read_maze(file)` é responsável por ler o arquivo `maze.txt` e transformá-lo em uma matriz retangular contendo em cada posição o caractere respectivo àquela posição no labirinto, sendo que:

- S — início
- G — objetivo
- # — parede
- `.` — livre

Assume-se para tal que o labirinto apresentado possui formato retangular, quando consideradas as paredes, apenas esse conjunto de caracteres válidos estará presente (outros caracteres serão considerados como '.'), e por fim, que apenas 1 inicio e 1 objetivo estão presentes no labirinto (Caso haja mais de um, apenas o último será considerado).

Em seguida `build_adjacency_list(maze)` utiliza a matriz da primeira função para gerar uma lista de adjacência onde cada posição livre do labirinto pode possuir 4 tuplas de offset de coordenadas, sendo estas relativas às quatro direções cardinais (Norte, Sul, Leste ou Oeste). Para tal ela passa por cada posição do labirinto, verificando se é livre e quais posições adjacentes também são livres, retornando no final a lista de adjacência completa na forma de um *Dicionário*

Sempre ao encontrar os caracteres especiais 'S' e 'G' a função `find_positions` modifica as variáveis `start` e `goal` para a posição atual na matriz.

`generate_maze_matrix` e `mark_path_on_maze` são responsáveis por reconstruir e modificar a matriz novamente com o caminho encontrado pelos algoritmos de busca. Sendo que um novo caractere é adicionado:

- $*$ — caminho percorrido

2.4 Heuristics.py

Um arquivo pequeno contendo as funções de geração de heurística para cada método selecionado:

- `manhattan_distance`: Para todas as posições da Lista de Adjacência calcula o módulo da subtração entre a posição da matriz do nó atual e do nó objetivo tanto verticalmente quanto horizontalmente e soma-se o resultado, conforme a função abaixo.

$$d_{\text{Manhattan}}(k, g) = |X_k - X_g| + |Y_k - Y_g|$$

- `manhattan_distance`: Para todas as posições da Lista de Adjacência calcula o quadrado da subtração entre a posição da matriz do nó atual e do nó objetivo tanto verticalmente quanto horizontalmente, soma-se o resultado e calcula-se a raiz quadrada, conforme a função abaixo.

$$d_{\text{Euclidiana}}(k, g) = \sqrt{(X_k - X_g)^2 + (Y_k - Y_g)^2}$$

Para ambas as heurísticas o resultado final é um Dicionário cuja chave também é a tupla contendo as coordenadas da matriz relativas à posição no labirinto.

2.5 Search.py

Contém as funções de busca implementadas na forma de funções cuja entrada é constituída pela Lista de Adjacência `adj`, a posição inicial `start` e o objetivo `goal` gerados em `Adj.py`, além do dicionário de heurística gerado em `Heuristics.py`.

Ao início da execução de cada algoritmo é criado um dicionário `tracker` que armazenará o nó pai de cada expansão.

Os métodos de busca implementados foram:

- **Breadth-First Search, `bfs`:** Cria-se uma fila inicialmente contendo apenas o nó `start`. Para cada nó na fila verifica-se se ele é `goal`, caso seja encerra-se a expansão, caso contrário, todos os seus vizinhos não visitados (`tracker` vazio para aquela posição) são adicionados à fila e o nó é adicionado a posição do dicionário `tracker` relativo a posição de seus vizinhos. Garante o **caminho mais curto** graças a estrutura da **fila**.

- **Depth-First Search, `dfs`:** Cria-se uma pílha inicialmente contendo apenas o nó `start`. Para cada nó na pilha verifica-se se ele é `goal`, caso seja encerra-se a expansão, caso contrário, todos os seus vizinhos não visitados (`tracker` vazio para aquela posição) são adicionados à pilha e o nó é adicionado a posição do

dicionário `tracker` relativo a posição de seus vizinhos. Pode levar a um resultado mais rápido por seguir um mesmo caminho até o **final**, graças a estrutura de **pilha**, mas dependendo do labirinto pode causar um **aumento** no tempo de execução.

- **Busca Gulosa, `greedy_search`:** Cria-se um vetor `frontier` inicialmente contendo apenas o nó `start`. A cada iteração e para cada nó no vetor verifica-se **apenas** o valor da **heurística** naquela posição. Caso seja **0** chegou-se ao objetivo, e encerra-se a expansão, caso contrário, todos os seus vizinhos não visitados (`tracker` vazio para aquela posição) são adicionados à fronteira e o nó é adicionado a posição do dicionário `tracker` relativo a posição de seus vizinhos. Em seguida o nó com menor valor de heurística é escolhido e analisado. Dessa forma, garante-se que sempre seja escolhido o nó que aparenta estar mais próximo do objetivo. Porém nem sempre essa estratégia leva ao caminho mais curto.

- **Busca A*, `A_star_search`:** Cria-se um vetor `frontier` inicialmente contendo apenas o nó `start`. A cada iteração e para cada nó no vetor verifica-se a **soma** entre o valor da heurística e o custo atual naquela posição. Caso o nó seja **goal**, chegou-se ao objetivo, e encerra-se a expansão, caso contrário, todos os seus vizinhos tem seu `new_cost` calculado e são adicionados à fronteira. Em seguida o nó com menor soma é escolhido e analisado com o nó atual nó sendo adicionado a posição do dicionário `tracker` relativo a posição de seus vizinhos. Dessa forma, garante-se que sempre seja escolhido o nó que possibilita o caminho mais curto até o objetivo, analisando-se todos os caminhos promissores a cada momento.

  Ao chegar-se ao nó `goal`, para todos os métodos de busca o caminho inverso é percorrido utilizando `tracker` e armazenando-o no vetor `path` até chegar a posição inicial. Caso não se encontre o objetivo, este não estará contido no `tracker`, o programa perceberá isso e retornará um vetor vazio.

2.6 Maze.py

O script principal do programa, responsável pela chamada de cada função de busca, e pela aferição, impressão e armazenamento dos resultados de execução.

Na função `main` são definidos o caminho para o arquivo com o labirinto, os algoritmos a serem executados, inicializa-se um dicionário onde serão armazenados os tempos de execução usando a biblioteca `time`. Após a criação da lista de adjacência outras variáveis são inicializadas para cada `match case` presente em `alg`.

Fazendo uso da biblioteca `memory_profile`, inicia-se um wrapper para cada algoritmo que mede o uso de memória da instância de execução durante a solução do problema.

Após cada execução, o programa salva o labirinto solucionado em um arquivo `.txt`, e um gráfico de uso de memória como `.png` por meio da biblioteca `matplotlib`, além de chamar a função `pretty_print_maze`, que faz o uso da biblioteca colorama para imprimir a matriz modificada com o caminho percorrido em verde para facilitar a visualização e comparação entre algoritmos.

Por fim o tempo de execução, expansões e nós gerados são impressos no terminal e passa-se a execução do próximo algoritmo.

3 Analise de Resultados

Após a execução do código, foi gerada a seguinte tabela contendo os valores de execução para cada algoritmo e cada heurística no caso das buscas informadas

Verifica-se que o algoritmo DFS não teve um tempo contabilizável na unidade de medida escolhida, demonstrando desempenho vastamente superior aos outros algoritmos. Uma explicação possível é o pouco uso de leitura e escrita de memória nos algoritmos de busca não-informada, aliado a um labirinto que contribuiu para a DFS por possuir poucos caminhos longos sem saída que seriam percorridos pelo algoritmo antes do caminho correto.

O uso de memória do DFS não foi o menor, apesar de os nós gerados/expandidos o serem, com o menor uso sendo do BFS, provavelmente devido à estrutura de fila, que constantemente garantia que elementos fossem removidos a uma razão similar a que eram adicionados (em um labirinto apertado como o apresentado a média de posições livres em cada posição se aproxima de 2, com um deles sendo o nó atual, que não precisa ser revisitado).

Já dentre os algoritmos de busca informada. O algoritmo guloso de Distância Euclidiana teve tempo de execução **4 vezes menor** que o algoritmo mais lento - Distância de Manhattan A* - mostrando a grande diferença do algoritmo guloso em seu **cenário ideal** e a sua superioridade para a estrutura de labirinto apresentada. Isso se faz ainda mais evidente quando se analisa o total de nós gerados/expandidos entre os algoritmos A* e suas contrapartidas gulosas. O algoritmo guloso conseguiu eliminar aproximadamente **1/3** das expansões de nó utilizadas no

A* em ambos os casos.

Como esperado, a Distância Euclidiana A*, sendo o algoritmo com mais variáveis envolvidas em sua execução, e o segundo lugar em número de nós expandidos, foi o algoritmo com maior uso de memória, 50% maior uso que o BFS, e 8% mais que a média dos outros algoritmos.

Algoritmo	Tempo (μs)	Nós Gerados	Nós Expandidos	Memória Usada (MiB)	Completude	Optimalidade
BFS	1000.64	915	913	62	Completo (se $custo = 1$)	Ótimo (se passos tem ocom mesmo custo)
DFS	0*	450	445	82	Não completo	Não ótimo
Manhattan (Guloso)	2001.29	555	549	86	Completo	Não ótimo
Manhattan (A*)	2007.96	768	767	88	Completo	Ótimo
D. Euclidiana (Guloso)	504.25	503	496	89	Completo	Não ótimo
D. Euclidiana (A*)	2001.76	807	805	92	Completo	Ótimo

Table 1. Comparação de algoritmos de busca

4 Conclusão

Conclui-se portanto, a importância de uma boa escolha de algoritmo para cada problema. No caso da estruturação do labirinto - caminhos estreitos, com poucas encruzilhadas e muitas paredes - o caminho encontrado pelos algoritmos **gulosos**, em especial a **Distância Euclidiana** foi igualmente satisfatório, e o **ganho de desempenho** foi **severo**. Pode-se porém, inferir que tais vantagens desapareceriam em um ambiente maior, com uma quantidade exponencialmente maior de caminhos errados e encruzilhadas cujo caminho mais curto leve a becos sem saída.

Assim não pode-se ignorar as vantagens apresentadas pelos outros algoritmos, como a garantia de menor caminho do A*, o baixo uso total de memória do BFS, e especialmente a pouca movimentação de memória do DFS, que, em labirintos pequenos, já apresentou um ganho de performance quando colocado em hardware real.

5 Créditos

- **João Vitor Neves Costa:** Implementação `Maze.py`, `Search.py`, escrita seções *1, 2.5, 2.6, 4, 6*
- **Arthur Santana de Mesquita:** Implementação `Heuristics.py`, `Adj.py`, análise de desempenho, escrita seções *2.1, 2.3, 2.4, 3*

6 Referências

References

[1] Ai — search algorithms — greedy best-first search. URL https://www.codecademy.com/resources/docs/ai/search-algorithms/greedy-best-first-search.

[2] Rajesh Kumar. The a* algorithm: A complete guide, 11 2024. URL https://www.datacamp.com/tutorial/a-star-algorithm.

[3] Stuart J Russell and Peter Norvig. *Inteligência Artificial - Uma Abordagem Moderna*. GEN LTC, 08 2022.