Trabalho 1: Implementação e Análise dos Algoritmos de Busca no Labirinto

João Marcelo Gonçalves Lisboa Sergio Henrique Quedas Ramos

Sumário

1 Fundamentação Teórica

Algoritmos de busca são a espinha dorsal de muitos problemas em Inteligência Artificial, permitindo que um agente encontre uma sequência de ações para atingir um objetivo. Neste trabalho, exploramos duas categorias principais de algoritmos de busca clássicos: buscas não informadas e buscas informadas.

1.1 Buscas Não Informadas (Cegas)

Buscas não informadas, ou buscas cegas, operam sem qualquer conhecimento sobre a "direção"do objetivo. Elas exploram o espaço de estados de forma sistemática, baseando-se apenas na estrutura do grafo do problema.

1.1.1 Busca em Largura (BFS)

A Busca em Largura (Breadth-First Search) explora o grafo "camada por camada". Partindo do nó inicial, ela visita todos os seus vizinhos diretos, depois os vizinhos dos vizinhos, e assim por diante. Para gerenciar os nós a serem explorados, o BFS utiliza uma estrutura de dados de Fila (Queue), que opera no princípio "primeiro a entrar, primeiro a sair"(FIFO).

No contexto do labirinto, o BFS é **completo** (sempre encontra a solução se ela existir) e encontra o caminho com o menor número de passos, assumindo que todos os passos têm custo uniforme (neste caso, custo 1.0).

1.1.2 Busca em Profundidade (DFS)

A Busca em Profundidade (Depth-First Search) explora um ramo do grafo o mais profundamente possível antes de fazer o "backtracking"e explorar outro ramo. Ela utiliza uma estrutura de dados de Pilha (Stack), que opera no princípio "último a entrar, primeiro a sair"(LIFO).

O DFS é **completo** para espaços de estados finitos (como o nosso labirinto), mas **não é ótimo**. Ele tende a encontrar soluções rapidamente, mas estas são frequentemente caminhos longos e ineficientes, pois ele simplesmente segue o primeiro caminho válido que encontra até o fim.

1.1.3 Busca de Custo Uniforme (UCS)

A Busca de Custo Uniforme (Uniform Cost Search) é uma evolução do BFS que é otimizada para problemas onde os custos dos passos não são uniformes (embora em nosso labirinto o custo seja sempre 1.0). O UCS expande o nó com o menor custo de caminho acumulado ($g(n)$) a partir da origem. Para isso, utiliza uma Fila de Prioridade (Priority Queue).

O UCS é **completo** e **ótimo**. No nosso cenário de custo uniforme, seu comportamento é idêntico ao do BFS, embora possa ter uma sobrecarga computacional ligeiramente maior devido ao gerenciamento da fila de prioridade.

1.2 Buscas Informadas (Heurísticas)

Buscas informadas utilizam conhecimento adicional sobre o problema, na forma de uma função heurística ($h(n)$), para estimar a distância de um nó n até o objetivo. Isso permite que a busca explore de forma mais "inteligente", priorizando caminhos que parecem promissores.

1.2.1 Busca Gulosa (Greedy Best-First)

A Busca Gulosa (Greedy) tenta se aproximar do objetivo o mais rápido possível, ignorando o custo do caminho já percorrido. Ela expande o nó que tem o menor valor heurístico $h(n)$. A sua função de avaliação é:

$$f(n) = h(n)$$

Ela utiliza uma Fila de Prioridade ordenada por $h(n)$. Esta abordagem é rápida, mas "míope": pode ser atraída por um caminho que parece curto, mas que leva a um beco sem saída ou a um caminho total muito longo. Portanto, a Busca Gulosa **não é ótima** e, em grafos com ciclos, pode não ser completa.

1.2.2 Busca A*

A busca A* é amplamente considerada o algoritmo de busca de caminho mais eficiente e completo. Ele combina a eficiência da Busca Gulosa com a otimalidade do UCS. O A* avalia os nós com base em uma função que soma o custo real do caminho desde o início ($g(n)$) com o custo heurístico estimado até o objetivo ($h(n)$):

$$f(n) = g(n) + h(n)$$

O A* utiliza uma Fila de Prioridade (ordenada por $f(n)$). O A* é **completo** e **ótimo**, contanto que a função heurística $h(n)$ seja **admissível**, ou seja, que ela nunca superestime o custo real para alcançar o objetivo.

2 Metodologia Aplicada

O projeto foi estruturado em múltiplos arquivos Python, cada um com uma responsabilidade clara, para implementar e avaliar os algoritmos de busca em um labirinto carregado de um arquivo de texto.

- **Labirinto (‘labirinto.txt‘):** Define o ambiente. É um arquivo de texto simples onde 'S' representa o início, 'G' o objetivo, '' as paredes (obstáculos) e '.' os caminhos livres.
- **‘maze_representation.py‘:** Contém a classe `Maze`, responsável por carregar o arquivo `.txt`, identificar as posições de início e fim, e definir as regras do ambiente (quais ações são válidas, qual o resultado de uma ação, e o custo de um passo). Também define a classe `Node`, que armazena o estado, o nó pai (para reconstrução do caminho), o custo acumulado ($g(n)$) e o valor heurístico ($h(n)$).

- **‘data_structures.py‘:** Implementa as estruturas de dados fundamentais exigidas pelos algoritmos: `Stack` (para DFS), `Queue` (para BFS) e `PriorityQueue` (para UCS, Greedy e A*). A fila de prioridade foi implementada usando uma lista ordenada.
- **‘heuristics.py‘:** Define as funções heurísticas utilizadas pelas buscas informadas. Neste trabalho, implementamos a Distância de Manhattan e a Distância de Chebyshev.
- **‘search.py‘:** Contém a lógica central de todos os algoritmos de busca: `bfs_search`, `dfs_search`, `ucs_search`, `greedy_search` (usando Manhattan), `astar_search` (usando Manhattan), `greedy_chebyshev_search` e `astar_chebyshev_search`. Cada função retorna um dicionário contendo métricas detalhadas (sucesso, custo, caminho, nós expandidos, memória, tempo).
- **‘main.py‘:** É o ponto de entrada da aplicação. Ele instancia o labirinto, executa todos os algoritmos de busca através da função `run_all_searches`, exibe uma tabela comparativa no console e utiliza a função `plot_maze_search` (com `matplotlib`) para gerar e salvar visualizações gráficas do caminho encontrado e dos nós explorados por cada algoritmo. As imagens são salvas no diretório **data**/.

3 Análise Comparativa

A eficácia de um algoritmo de busca é medida por métricas como custo do caminho, número de nós expandidos (complexidade de tempo), uso de memória e se a solução é ótima.

3.1 Heurísticas Utilizadas

As buscas informadas dependem crucialmente da qualidade de suas heurísticas.

3.1.1 Distância de Manhattan

A Distância de Manhattan (usada em `greedy_search` e `astar_search`) calcula o custo de movimento em um grid permitindo apenas movimentos horizontais e verticais. A fórmula é:

$$h(n) = |n_{linha} - G_{linha}| + |n_{coluna} - G_{coluna}|$$

Esta heurística é perfeitamente adequada e admissível para o nosso labirinto, pois o custo de cada movimento (N, S, L, O) é 1, e ela calcula exatamente o menor número de passos na ausência de obstáculos.

3.1.2 Distância de Chebyshev

A Distância de Chebyshev (usada em `greedy_chebyshev_search` e `astar_chebyshev_search`) calcula a distância permitindo movimentos diagonais (como um rei no xadrez). A fórmula é:

$$h(n) = \max(|n_{linha} - G_{linha}|, |n_{coluna} - G_{coluna}|)$$

No nosso problema, onde movimentos diagonais não são permitidos, o custo real de um movimento "diagonal"(ex: Leste + Sul) é 2. A heurística de Chebyshev retorna 1 para este mesmo movimento. Como $h(n) \leq h^*(n)$ (o custo real), esta heurística também é admissível, mas ela subestima mais o custo real do que Manhattan.

3.2 Resultados e Discussão

- Os algoritmos foram executados no labirinto 8x8:

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

Entretanto qualquer tamanho é aceito

- A tabela 3.2 resume as métricas de desempenho coletadas diretamente da saída do terminal.

Algoritmo	**Custo**	**Nós Expandidos**	**Memória Máx.**	**Tempo (s)**	**Ótimo?**
BFS	11.00	31	45	0.0001	SIM
DFS	21.00	36	56	0.0002	NÃO
UCS	11.00	31	45	0.0002	SIM
GREEDY (Manhattan)	11.00	11	29	0.0001	NÃO
A* (Manhattan)	11.00	15	26	0.0001	SIM
GREEDY (Chebyshev)	11.00	11	29	0.0001	NÃO
A* (Chebyshev)	11.00	20	31	0.0001	SIM

Tabela 1: Comparação de Métricas dos Algoritmos de Busca.

Pontos cruciais das análises dos resultados:

1. **Otimalidade de Custo:** O custo ótimo para este labirinto é **11.00**, encontrado por BFS, UCS e ambas as variações do A*. A busca DFS falhou catastroficamente em otimalidade, encontrando um caminho de custo **21.00** — quase o dobro — o que é confirmado pelo seu longo e sinuoso caminho de solução.

2. **A "Sorte"da Busca Gulosa:** É de extrema importância notar que ambas as buscas `GREEDY` (Manhattan e Chebyshev) também encontraram o caminho de custo 11.00. No entanto, elas estão corretamente marcadas como `Ótimo: NÃO`. Isso ocorre porque a otimalidade não é uma garantia para a Busca Gulosa; ela pode facilmente ser "enganada"por um beco sem saída. Neste labirinto específico, a heurística pura (sem $g(n)$) foi suficiente para guiá-la ao caminho ótimo.

3. **Eficiência (Nós Expandidos):** Aqui reside a principal diferença.

   - As buscas **Greedy** foram as mais eficientes, expandindo apenas **11 nós**. Elas simplesmente "correram"em direção ao objetivo.
   - **A* (Manhattan)** foi o algoritmo ótimo mais eficiente, expandindo apenas **15 nós**. Isso demonstra o poder da função $f(n) = g(n) + h(n)$, que evitou explorações desnecessárias.
   - **A* (Chebyshev)** foi menos eficiente (20 nós). Isso prova que Manhattan é uma heurística mais informada (mais precisa) para este problema, pois sua estimativa é mais próxima do custo real, direcionando a busca de forma mais eficaz.
   - **BFS e UCS** tiveram desempenho idêntico (31 nós), como esperado, pois o custo é uniforme. Elas exploraram significativamente mais que o A*, pois não tinham "senso de direção".
   - **DFS** foi o que mais expandiu nós (36), perdendo-se em ramos profundos antes de fazer o backtracking.

4. **Uso de Memória:**A métrica 'memoria_maxima' representa o pico da soma dos nós na **fronteira** (nós a explorar) e no **conjunto de explorados** (nós já visitados).

   - **Pior Caso (DFS): 56 unidades.** A Busca em Profundidade foi a que mais consumiu memória. Embora sua fronteira (a Pilha) seja geralmente pequena (proporcional à profundidade do caminho, $b \times m$), o conjunto `explored` precisa armazenar todos os nós visitados para evitar ciclos. Como o DFS se perdeu em caminhos longos e errados (custo 21), ele visitou e armazenou 36 nós distintos, levando ao maior pico de memória.
   - **Buscas Cegas (BFS e UCS): 45 unidades.** O BFS e o UCS apresentaram uso de memória idêntico e elevado. O gargalo aqui é a **fronteira** (a Fila). O BFS explora camada por camada, e sua fronteira precisa armazenar potencialmente todos os nós de uma camada antes de passar para a próxima. Isso é exponencialmente custoso em memória. O UCS, em labirintos de custo uniforme, comporta-se exatamente como o BFS.
   - **Buscas Informadas (A* e Greedy): 26-31 unidades.** Aqui vemos a maior vantagem das heurísticas.
     - **A* (Manhattan)** foi o mais eficiente, com pico de apenas **26 unidades**. Isso porque sua heurística é muito precisa ($h(n)$ é próximo de $h^*(n)$) e a função $f(n) = g(n)+h(n)$ o impede de se afastar muito do caminho ótimo. Ele mantém uma fronteira pequena e um conjunto de explorados focado.
     - **A* (Chebyshev)** usou mais memória (31 unidades). Como a heurística de Chebyshev é menos precisa (subestima mais o custo real), a busca se torna menos "confiante". Ela precisa explorar mais caminhos "talvez"bons, aumentando o tamanho da fronteira e do conjunto de explorados (20 nós expandidos vs. 15 do Manhattan) antes de achar o ótimo.
     - **Greedy** (ambos) tiveram um pico de **29 unidades**. Embora tenham expandido menos nós (11), seus picos de memória foram ligeiramente maiores que o A* (Manhattan). Isso pode ocorrer porque a busca gulosa (f=h) pode "mergulhar"em um caminho que parece bom, aumentando a fronteira, e só

depois perceber (ao expandir) que não era o ideal, enquanto o A* (f=g+h) é mais "cauteloso"e mantém sua fronteira mais enxuta ao longo de toda a execução.

Em suma, a memória é o "calcanhar de Aquiles"das buscas cegas (BFS/UCS) em espaços de estados grandes. As buscas informadas, especialmente o A* com uma boa heurística (Manhattan), resolvem esse problema de forma eficaz, mantendo a fronteira de busca gerenciável e focada no objetivo.

5. **Análise do Tempo de Execução (Tempo (s)):** Observa-se na tabela que todos os tempos de execução foram extremamente baixos (na ordem de 0.0001s a 0.0002s) e muito similares entre si. Isso se deve ao fato do labirinto de teste ser pequeno (8x8). Em um cenário com um labirinto significativamente maior (ex: 100x100), as diferenças de eficiência algorítmica se tornariam mais evidentes. Algoritmos menos eficientes, como o BFS (que expandiu 31 nós), levariam um tempo de execução substancialmente maior em comparação com o A* (Manhattan) (que expandiu 15 nós). Portanto, em problemas de maior escala, a superioridade em 'Nós Expandidos' do A* se traduziria diretamente em um tempo de execução muito menor. Da mesma forma, as diferenças nas outras métricas, como 'Memória Máx', também se tornariam exponencialmente mais visíveis.

3.2.1 Visualização dos Caminhos

As figuras a seguir ilustram o comportamento de cada algoritmo. As áreas coloridas (em tons de azul) mostram a ordem de expansão dos nós (quanto mais escuro, mais tarde foi explorado), e a linha vermelha mostra o caminho da solução final.

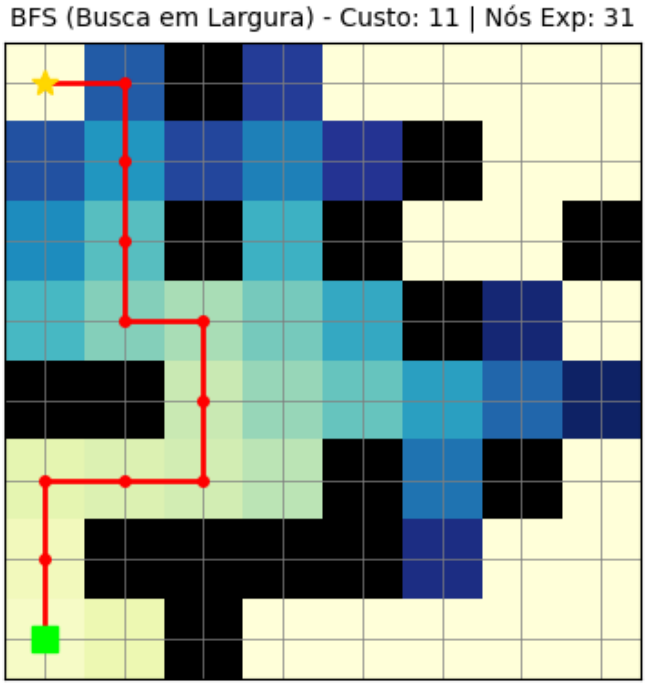

Figura 1: Caminho e Nós Expandidos - BFS

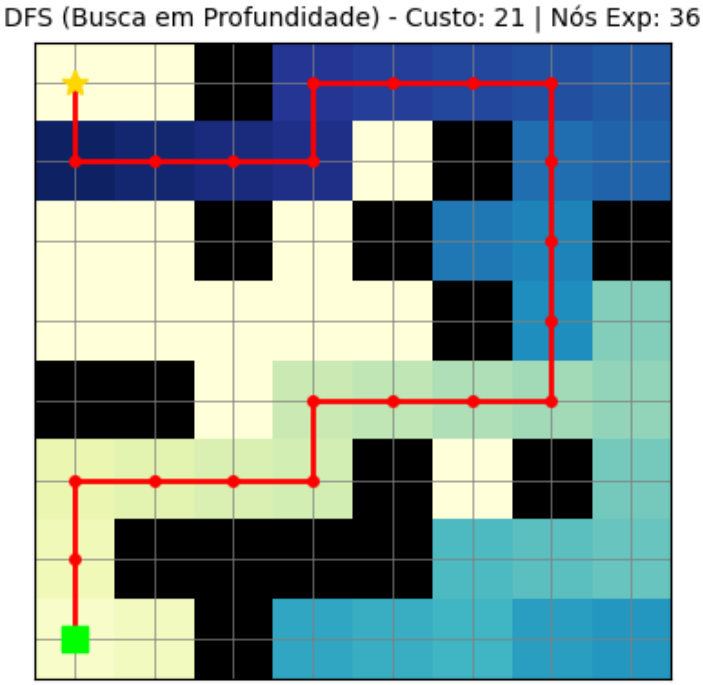

Figura 2: Caminho e Nós Expandidos - DFS

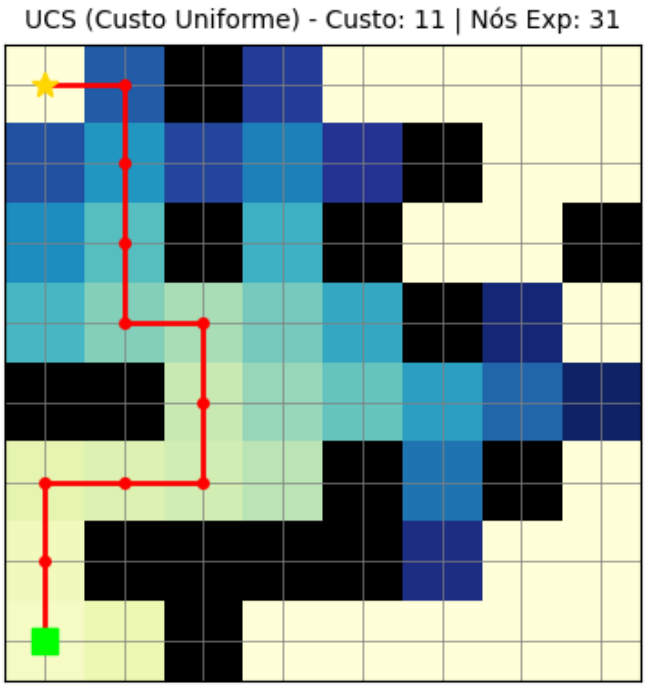

Figura 3: Caminho e Nós Expandidos - UCS

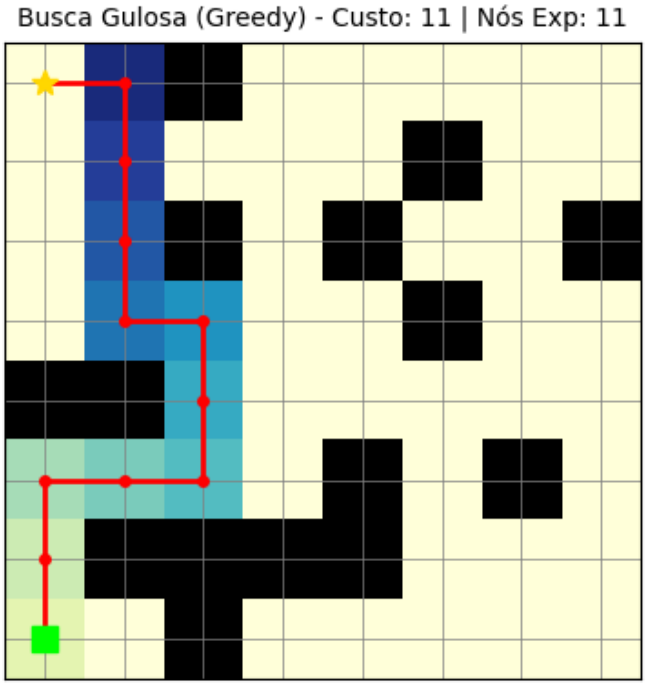

Figura 4: Caminho e Nós Expandidos - Greedy (Manhattan)

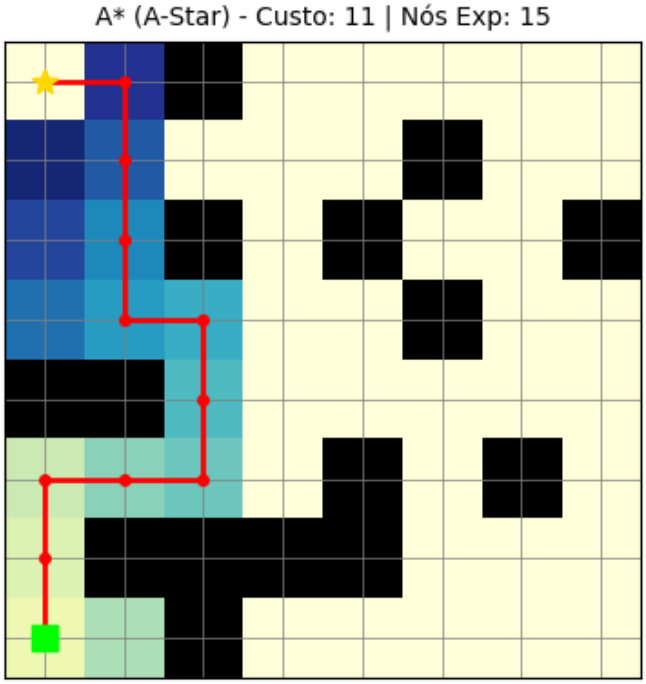

Figura 5: Caminho e Nós Expandidos - A* (Manhattan)

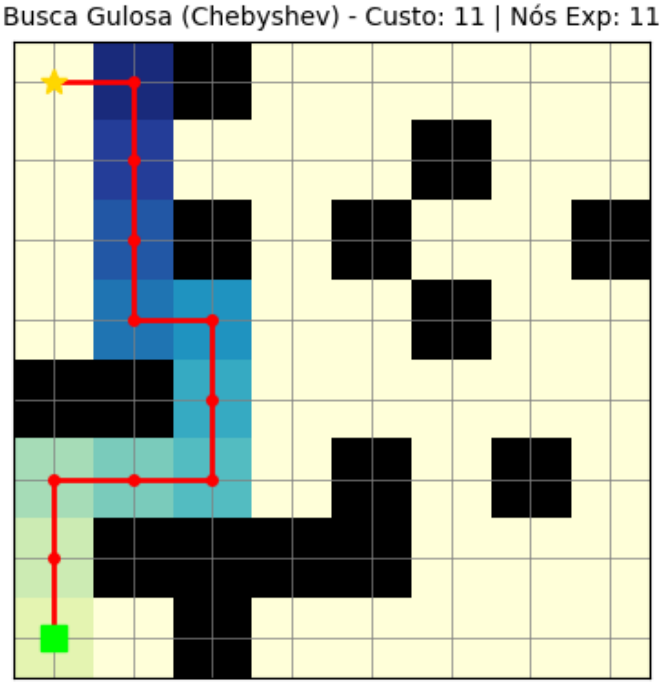

Figura 6: Caminho e Nós Expandidos - Greedy (Chebyshev)

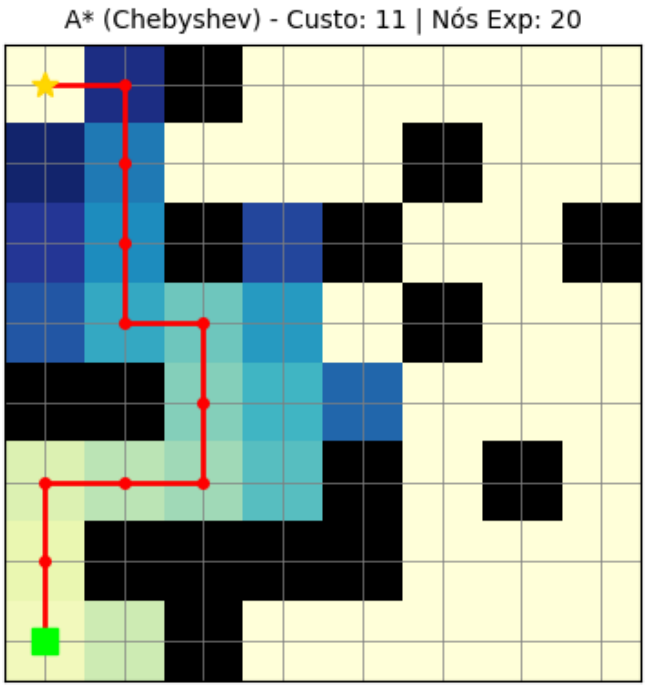

Figura 7: Caminho e Nós Expandidos - A* (Chebyshev)

4 Divisão de Tarefas

4.1 Papeis dos autores:

- **João Marcelo Gonçalves Lisboa**:Ficou responsável pela implementação dos algoritmos.
- **Sergio Henrique Quedas Ramos**: Ficou responsável pela documentação (devido a problemas médicos).

4.2 Uso de IA:

Foi utilizado o Google Gemini. Apenas para auxilio na revisãoda escrita.

4.3 Declaração:

Confirmamos que o código entregue foi desenvolvido pela equipe, respeitando as políticas da disciplina. Dessa forma para a axulio da implementação dos algoritmos foram utilizados as referências presentes na section 6.

5 Conclusão

Neste trabalho, implementamos e comparamos sete algoritmos de busca distintos para a resolução de um labirinto 8x8. Os resultados práticos (Tabela 3.2) validaram a teoria: as buscas A*, BFS e UCS encontraram o caminho ótimo de custo **11.00**, enquanto o DFS encontrou um caminho subótimo de custo **21.00**.

A análise revelou um claro trade-off entre as métricas:

- **DFS:** Falhou em otimalidade de custo (21.00) e foi o pior em consumo de memória (56 unidades) devido ao grande número de nós explorados em caminhos incorretos.
- **BFS/UCS:** Garantiram a otimalidade, mas ao custo de uma alta exploração de nós (31) e, consequentemente, alto uso de memória (45 unidades), pois exploram "cegamente"em todas as direções.
- **Greedy:** Foram os mais rápidos em termos de nós expandidos (11), mas não oferecem garantia de otimalidade (apesar de terem "acertado"neste labirinto específico).
- **A* (A-Estrela):** Provou ser o algoritmo superior. O **A* (Manhattan)** foi o vencedor geral, garantindo o caminho ótimo (custo 11.00) com o menor consumo de memória entre os algoritmos ótimos (26 unidades) e o menor número de nós expandidos (15).

A análise comparativa entre as heurísticas também foi conclusiva: a Distância de Manhattan, por ser uma estimativa mais precisa (mais informada) do custo real em um grid de 4 direções, superou a Distância de Chebyshev, resultando em menos nós expandidos (15 vs. 20) e menor pico de memória (26 vs. 31) para o A*.

Conclui-se que para problemas de caminho em grid, o A* com a heurística de Manhattan oferece a melhor combinação de otimalidade garantida e eficiência computacional, tanto em tempo (nós expandidos) quanto em espaço (memória).

6 Referências

Referências

[1] Russell, Stuart J., e Peter Norvig. *Artificial Intelligence: A Modern Approach*. 4ª ed., Pearson, 2020.

[2] Russell, S. J., e Norvig, P. *AIMA-Python: Python implementation of algorithms from "Artificial Intelligence: A Modern Approach"*. GitHub, 2023. Disponível em: https://github.com/aimacode/aima-python.

[3] Patel, Amit J. "Introduction to A*". *Red Blob Games*, 2023. Disponível em: https://www.redblobgames.com/pathfinding/a-star/introduction.html.

[4] *Trabalho-IA: Implementação e Análise dos Algoritmos de Busca no Labirinto*. GitHub Repository. Disponível em: https://github.com/joaojmgl/Trabalho-IA.git (Repositório do código-fonte desenvolvido para este trabalho).