Centro Federal de Educação Tecnológica de Minas Gerais
Engenharia de Computação
Inteligência Artificial

Trabalho 01:
Busca no Labirinto (Não Informada e Informada)

Guilherme Alvarenga de Azevedo
Maria Eduarda Teixeira Souza

Divinópolis - MG
Outubro de 2025

Sumário

Problema

Introdução

Este trabalho aborda a implementação e análise de algoritmos de busca aplicados à resolução de labirintos. Para isso, os labirintos são representados como matrizes convertidas em grafos, nos quais cada posição corresponde a um nó e cada movimento possível a uma aresta. Os algoritmos estudados exploram o espaço de estados de maneiras distintas, permitindo comparar desempenho, eficiência e comportamento frente a diferentes configurações e heurísticas.

Descrição

Encontrar o caminho correto de um labirinto, considerando cada posição como um nó e cada movimento permitido como uma transição, permite avaliar como diferentes algoritmos exploram o espaço de estados. Assim, neste trabalho será discutido:

Objetivo

O objetivo deste trabalho é implementar e comparar dois algoritmos de busca não informada e dois algoritmos de busca informada, aplicados ao labirinto da Figura 1.

O Labirinto

A Figura 1 é a representação básica, passada pelo professor para ser usada como base para o desenvolvimento deste trabalho:

Figura 1 – Labirinto para testes

Como pode ser visualizado, apesar dos espaços do labirinto serem 5×5, também deve-se incluir as "paredes", fazendo com que se torne 9×9 na representação em TXT denotada abaixo, que será transformada em grafo pelo programa.

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

Assim, foi possível plotar o labirinto em uma visualização melhor, reproduzida no *GUI do Projeto*, veja na Figura 2:

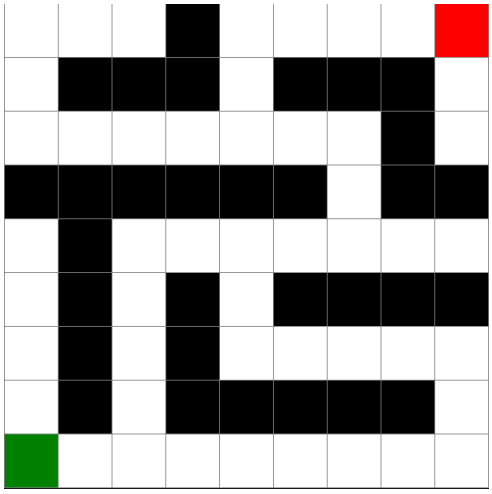


Figura 2 – Labirinto para testes - Visualização

A partir dessa estrutura, as análises serão realizadas a fim de determinar: quais algoritmos performam melhor em determinadas situações; quais os recursos computacionais necessários para se realizar uma busca, e como balanceá-los para atingir uma melhor solução; como a disposição de paredes do labirinto pode afetar os algoritmos; qual o impacto de diferentes heurísticas para algoritmos de busca informada; como todos esses parâmetros influenciam na capacidade dos algoritmos de encontrarem uma solução quando ela existe (completude) e de encontrar a que possui o menor custo possível (otimalidade).

Modelagem

Para resolver o problema do labirinto, é necessário definir formalmente os elementos do espaço de estados, as ações possíveis e os critérios de transição. Esta modelagem fornece a base para aplicar algoritmos de busca de forma consistente e eficiente, garantindo que cada nó, ação e custo seja tratado de maneira estruturada.

Representação como Grafo

Deve ser levado em consideração o fato de que o labirinto matricial do problema em questão é simplesmente modelado com uma estrutura em grafo, cujos nós são as posições (i, j), e cujas arestas são as transições (N, S, L, O), que representam ações de custo unitário. Cada posição do labirinto é avaliada para determinar seus vizinhos válidos, respeitando os limites da matriz e a presença de paredes, permitindo a conversão do labirinto para uma representação em grafo, como uma lista de adjacência do formato:

$$(i, j) : [(i \pm 1, j \pm 1)]$$

A modelagem do problema é feita através da classe `MazeProblem`, que herda de uma classe base abstrata `Problem`. Essa classe define formalmente o estado inicial, o teste de objetivo, as ações possíveis, os resultados das ações e os custos associados a cada transição. Assim, o labirinto deixa de ser apenas uma matriz de caracteres e passa a ser um espaço de estados estruturado, pronto para aplicação de algoritmos de busca.

Para representar a exploração do espaço de estados, cada posição do labirinto é encapsulada em um objeto `Node`. Cada `Node` armazena informações sobre seu estado, custo acumulado (g), heurística (h), custo total (f), ação que levou a ele e seu nó pai. Essa estrutura permite reconstruir o caminho da solução, calcular métricas de custo e orientar algoritmos informados, como A*, e não informados, como Dijkstra, garantindo consistência na expansão e avaliação dos nós durante a busca.

Estruturas de Dados

O primeiro passo consiste na leitura do labirinto a partir de um arquivo de texto. Cada linha do arquivo representa uma linha da matriz, e cada caractere é transformado em um elemento da matriz, permitindo a distinção entre espaços livres, paredes, ponto de início e ponto de objetivo. Esse processo gera uma estrutura bidimensional que mantém a disposição original do labirinto.

A partir dessa matriz, é gerado um grafo em que cada célula livre corresponde a um nó e cada vizinho acessível nas quatro direções (Norte, Sul, Leste e Oeste) constitui uma aresta com custo unitário. Deve ser levado em consideração o fato de que o labirinto matricial do problema em questão é simplesmente modelado com uma estrutura em grafo, cujos nós são as posições (i, j), e cujas arestas são as transições (N, S, L, O), que representam ações de custo unitário. A classe `Maze` encapsula essa lógica, oferecendo métodos para verificar limites, passabilidade, ações válidas e vizinhos, além de permitir a conversão em grafo de adjacência e impressão formatada.

Cada estado explorado é representado por um `Node`, que armazena o estado atual, a ação que o gerou, uma referência para o nó pai e os custos associados: g (custo acumulado), h (heurística) e f (custo total). Essa estrutura possibilita reconstruir o caminho completo a partir do nó objetivo e comparar nós na fronteira durante a expansão.

A classe `MazeProblem` conecta o grafo e os nós à lógica de busca, definindo o estado inicial, teste de objetivo, ações possíveis, resultados de ações e custos. Para algoritmos informados, também fornece o cálculo da heurística, permitindo a aplicação do A* e da Busca Gulosa.

No caso do algoritmo de Dijkstra, a busca é realizada de forma similar ao A*, mas a prioridade de expansão de cada nó é determinada exclusivamente pelo custo acumulado $g(n)$. A fronteira é gerenciada por uma fila de prioridade que seleciona sempre o nó com menor custo total percorrido até o momento, garantindo a exploração ordenada e a descoberta de caminhos de menor custo. O algoritmo bidirecional segue a mesma linha de raciocínio, porém realiza duas buscas simultâneas: uma partindo do nó inicial e outra partindo do nó objetivo. Cada fronteira é gerenciada separadamente e a busca termina quando um nó explorado por uma das direções já foi alcançado pela outra. Assim, é possível reduzir o número de nós expandidos, embora a memória utilizada seja levemente maior, já que duas fronteiras e dois conjuntos de nós alcançados são mantidos simultaneamente.

A cada expansão, vizinhos válidos são gerados como novos nós e adicionados à fronteira caso ainda não tenham sido explorados ou se um caminho mais curto para eles for encontrado. Dessa forma, toda a estrutura de dados — desde a leitura da matriz, geração do grafo, representação dos estados via `Node` e integração com a lógica de busca via `MazeProblem` — permite que os algoritmos explorem o espaço de estados de maneira organizada e eficiente, possibilitando análise precisa de tempo de execução, memória utilizada e qualidade das soluções encontradas.

Tamanho e Complexidade

A matriz utilizada permite uma árvore de busca máxima de tamanho $9 \times 9 = 81$ nós. Por isso, os resultados encontrados podem não ser muito diferentes entre si, já que para tamanhos pequenos todos os algoritmos podem performar bem, a depender da disposição de paredes - como será visto nas próximas seções.

Algoritmos e Heurísticas

Neste trabalho, são estudados algoritmos de busca em labirintos, divididos em não informados e informados. Entre os algoritmos não informados, serão abordados o *Dijkstra* e a busca bidirecional de custo uniforme, que exploram o espaço de estados sem informação adicional sobre a localização do objetivo. Já entre os algoritmos informados, serão analisados o A* e a Busca Gulosa, que utilizam funções heurísticas para guiar a exploração de maneira mais eficiente.

Buscas Não Informadas

Serão detalhados o algoritmo de *Dijkstra*, que prioriza nós de menor custo acumulado, e a busca bidirecional, que combina duas buscas simultâneas, uma do início e outra do objetivo, reduzindo o número de expansões necessárias.

Buscas Informadas

O A* combina custo acumulado e estimativa heurística para encontrar caminhos ótimos, enquanto a Busca Gulosa expande nós com base apenas na heurística, priorizando velocidade em detrimento da garantia de otimalidade.

Heurísticas

Serão utilizadas três heurísticas diferentes: a Distância de Manhattan, adequada para movimentos ortogonais; a Distância Euclidiana, que considera o movimento em linha reta; e uma heurística inadmissível, criada para analisar o comportamento dos algoritmos quando a função superestima o custo real.

Critério de Análise e Avaliação

Para avaliar o desempenho dos algoritmos utilizados, foram coletadas diversas métricas que fornecem informações sobre tempo, memória, eficiência e qualidade das soluções encontradas.

O tempo médio (`avg time`) de execução indica quanto tempo, em milissegundos, o algoritmo leva para encontrar a solução, permitindo comparar a rapidez entre diferentes algoritmos.

A memória média (`avg memory`) utilizada mede o consumo de memória virtual durante a execução, fornecendo uma visão do impacto do algoritmo sobre os recursos do sistema.

O número médio de nós (`avg nodes`) explorados representa quantos nós do labirinto foram visitados até encontrar a solução, mostrando a eficiência da busca.

A memória atual média (`avg current`) e o pico de memória (`avg peak`) indicam, respectivamente, a memória ocupada ao final da execução e o ponto máximo de uso durante o processo, ajudando a identificar possíveis picos de consumo.

O número de soluções encontradas (`found count`) indica quantas vezes o algoritmo conseguiu chegar ao objetivo em múltiplas execuções, verificando consistência e confiabilidade.

Por fim, o custo médio (`avg cost`) da solução fornece o valor médio do caminho encontrado, essencial para avaliar se o algoritmo encontra soluções ótimas ou próximas do ótimo.

Organização e Execução do Projeto

No repositório do *Github* [1] encontram-se as implementações de todos os algoritmos utilizados. Os quatro algoritmos de busca foram traduzidos dos pseudocódigos encontrados em [2] e [3] e foram organizados de forma a permitir comparações: visuais, numéricas, gráficas e de utilização de recursos. Isso foi feito através do uso da linguagem *Python* e de algumas de suas bibliotecas.

Organização do diretório

O projeto está organizado de forma a separar claramente os dados, o código-fonte e os arquivos de configuração.

- **data/** - contém os arquivos de entrada (`input/`) como labirintos, e os resultados gerados (`output/`), que incluem gráficos (`graphics/`), métricas (`metrics/`) e GIFs de visualização (`visualization/`), separados por algoritmos informados e não informados.
- **src/** - código-fonte do projeto.
  - **core/** - implementação central do labirinto, definição de nós, problemas e heurísticas.
  - **informed/** e **uninformed/** - algoritmos de busca informados (A*, Gulosa) e não informados (Dijkstra, Bidirecional) com geração de visualizações.
  - **comparisons/** - scripts para gerar gráficos comparativos a partir das métricas.
  - **search/** - funções auxiliares de medição de tempo/memória e visualização do labirinto.
  - **tools/** - scripts principais de execução (`main.py`) e interface gráfica (`run_gui.py`).
- Arquivos de configuração e documentação, como `pyproject.toml`, `requirements.txt`, `README.md` e `relatorio.pdf`.

Visualização

O projeto conta com uma interface gráfica (GUI) desenvolvida para facilitar a visualização do comportamento dos algoritmos implementados. Nela, é possível acompanhar passo a passo a execução de cada algoritmo, tornando mais intuitiva a compreensão das funcionalidades do sistema.

Além da GUI, o projeto também permite a execução via `main.py`, proporcionando a mesma visualização, porém no terminal. Nesse modo, as execuções geram arquivos GIF na pasta `data/output/visualization`, que representam a evolução dos algoritmos. Esses GIFs também são gerados quando se utiliza a GUI, mas, neste caso, são exibidos diretamente na interface.

Complementarmente, todas as métricas capturadas durante as execuções são salvas em arquivos JSON, e a partir dessas informações são gerados gráficos comparativos entre os algoritmos. Todos os gráficos apresentados ao longo do relatório são produzidos a partir dessa etapa de visualização.

Instalação e Execução

Para utilizar o projeto, siga os passos abaixo:

1. Clone o repositório e entre na pasta do projeto:

```
git clone https://github.com/dudatsouza/ia-trabalhos.git
cd ia-trabalhos/trabalho1
```

2. Crie e ative um ambiente virtual (recomendado) - garanta que já possui o Python, no mínimo na versão 3.11.9:

```
python3 -m venv venv
source venv/bin/activate   # Linux/macOS
venv\Scripts\activate      # Windows
```

3. Instale as dependências com Poetry - garanta que possui o Poetry instalado:

```
poetry install
```

4. Alternativamente, instale as dependências com pip:

```
pip install -r requirements.txt
```

5. Caso o `tkinter` não esteja instalado (necessário para GUI):

```
sudo apt-get install python3-tk   # Linux
```

   Windows e macOS já incluem o `tkinter`.

6. Execute o programa:

   - Com GUI (<modulo> = `run_gui`) ou Sem GUI (<modulo> = `main`):
   - Linux/macOS:

```
PYTHONPATH='src' python3 -m tools.<modulo>
```

     ou

```
poetry run python src/tools/<modulo>.py
```

   - Windows:

```
set PYTHONPATH=src && python -m tools.<modulo>
```

     ou

```
poetry run python src/tools/<modulo>.py
```

O projeto depende das bibliotecas `numpy`, `matplotlib`, `memory-profiler`, `networkx` e `tkinter`. Todas as informações sobre as dependências também estão presentes no arquivo `requirements.txt`.

Buscas Não Informadas

Algoritmos Utilizados

Os algoritmos de busca não informada escolhidos para serem estudados nesta seção foram o de custo uniforme (ou *Dijkstra*) e o BFS (*Best-First-Search*) bidirecional, também com custo uniforme. Assim, foi possível analisar o impacto de se utilizar mais memória para armazenar outro problema idêntico, porém na direção contrária, e combiná-lo ao original para diminuir o tempo de execução e/ou nós expandidos.

Dijkstra

O Algoritmo de Dijkstra implementado é uma especialização da busca de melhor custo (*Best-First Search*). Ele utiliza como estruturas de dados uma fila de prioridade mínima (*min-heap*) para a fronteira, garantindo que o nó com o menor custo acumulado ($g(n)$) seja sempre escolhido para expansão, e uma tabela de dispersão (*hash table*) para armazenar os nós já alcançados, otimizando a verificação de estados visitados e evitando ciclos.

A função de avaliação do algoritmo é definida como $f(n) = g(n)$, onde $g(n)$ representa o custo real do caminho desde o nó inicial até o nó n. O processo se inicia inserindo o nó inicial na fronteira. A cada passo, o nó de menor custo é removido da fronteira. Se for o nó objetivo, a busca termina. Caso contrário, seus vizinhos são explorados; se um vizinho ainda não foi alcançado ou se um caminho mais curto para ele for encontrado, ele é adicionado à fronteira.

BFS Bidirecional

A busca bidirecional de custo uniforme opera executando duas buscas de *Dijkstra* simultaneamente: uma partindo do nó inicial (busca para frente) e outra partindo do nó final (busca para trás). Para a busca para trás, uma nova instância do problema é criada, na qual as posições de início e fim são invertidas.

Esta abordagem utiliza duas filas de prioridade mínimas, uma para cada fronteira de busca, e duas tabelas de dispersão para registrar os nós alcançados em cada direção. O algoritmo alterna a expansão entre as duas fronteiras, sempre selecionando o nó com o menor custo acumulado ($g(n)$) de qualquer uma das duas. A busca termina com sucesso quando um nó expandido por uma das buscas já se encontra na lista de nós alcançados da busca oposta. Nesse momento, os caminhos das duas direções são concatenados para formar a solução final.

Comparação

Análise Assintótica

A Tabela 1 abaixo teve suas informações retiradas de RUSSELL (2021).

Tabela 1 – Comparação entre os Algoritmos Não Informados - Análise Assintótica

Critério	Custo Uniforme/Dijkstra	BFS Bidirecional de Custo Uniforme
Completude	Sim	Sim
Otimalidade	Sim	Sim
Tempo	$O\left(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor}\right)$	$O\left(b^{\frac{d}{2}}\right)$
Espaço	$O\left(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor}\right)$	$O\left(b^{\frac{d}{2}}\right)$

($\boldsymbol{b}$: fator de ramificação; $\boldsymbol{d}$: profundidade da solução mais rasa ou $\boldsymbol{m}$ quando não há solução; $\boldsymbol{m}$: profundidade máxima da árvore de busca; $\boldsymbol{C^*}$: custo da solução ótima; $\boldsymbol{\epsilon}$: limite inferior de custos das ações)

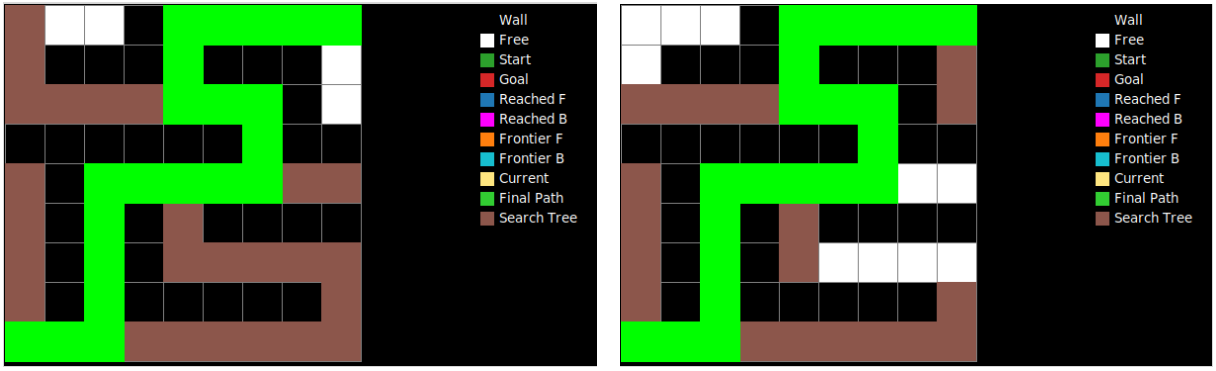
Segundo Russel, ambos são "completos se o valor de $\boldsymbol{b}$ é finito e o espaço de estados tem uma solução ou é finito". Para a completude do algoritmo de *Dijkstra*, também é levado em consideração que todas as "ações se mantêm com custo $\geq \epsilon > 0$". Como o algoritmo bidirecional utilizado é de "custo uniforme em ambas as direções", determina-se que possui mais uma condição necessária para ser completo e ótimo. Para ser ótimo, também, "o custo de todas as ações deve ser idêntico".

A complexidade de tempo e espaço do algoritmo de *Dijkstra* são limitadas pelo custo da solução ótima dividido pelo menor custo de cada ação. Portanto, quanto maior a diferença entre ambos, maior o tempo gasto e a memória utilizada pelo algoritmo. Já as complexidades do BFS bidirecional de custo uniforme, são limitadas pela metade da profundidade da solução ótima. Isso ocorre porque será necessário explorar e armazenar a metade do caminho da solução ótima de "cada lado" da busca - ou em cada direção de busca. Isso pode aumentar a quantidade de memória gasta, porém, diminui relativamente os nós expandidos, ou até o tempo de execução.

Soluções Encontradas

Como o custo é uniforme, e no problema em questão toda ação possui custo igual a 1, os algoritmos utilizados poderiam encontrar diferentes soluções ótimas, a depender do labirinto buscado e da ordem de busca (N, S, L, O). Porém, no caso aqui testado, a solução ótima possui custo 20.

As Figuras 3(a) e 3(b) foram prints retiradas do programa encontrado em [1], ao final do GIF de visualização. Elas exemplificam as soluções encontradas e as árvores de busca de ambos os algoritmos utilizados.



((a)) Árvore de Busca do Algoritmo de *Dijkstra*

((b)) Árvore de Busca do Algoritmo de Busca Bidirecional de Custo Uniforme

Figura 3 – Árvores de Busca dos Algoritmos Não Informados

Como é possível visualizar, a árvore de busca do algoritmo de busca bidirecional é menor que a do algoritmo de *Dijkstra* (menos quadrados marrons, segundo a legenda). Porém, encontraram a mesma solução, já que além de ser a de menor custo, é a única possível. Além disso, caso não exista solução (Figura 4), ambos os algoritmos são capazes de determinar essa condição.

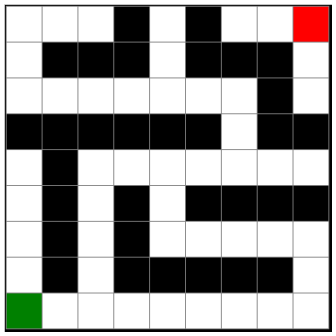


Figura 4 – Labirinto sem Solução Testado

Comparação de Desempenho Real

A Tabela 2 abaixo foi preenchida com as métricas obtidas no programa encontrado em [1], quando executado com o labirinto da Figura 1. Para tal, ambos os algoritmos foram executados 15 vezes e tiveram suas médias calculadas para cada critério.

Tabela 2 – Comparação entre os Algoritmos Não Informados - Desempenho Real

Critério	**Custo Uniforme/Dijkstra**	**BFS Bidirecional de Custo Uniforme**
Tempo de Execução (ms)	0,774	0,472
Nós Explorados	45,0	39,0
[1]Memória Máxima Gasta (KB)	36,402	37,521
[1]Memória Gasta ao Final da Execução (KB)	2,780	3,898
Soluções Encontradas	15/15	15/15

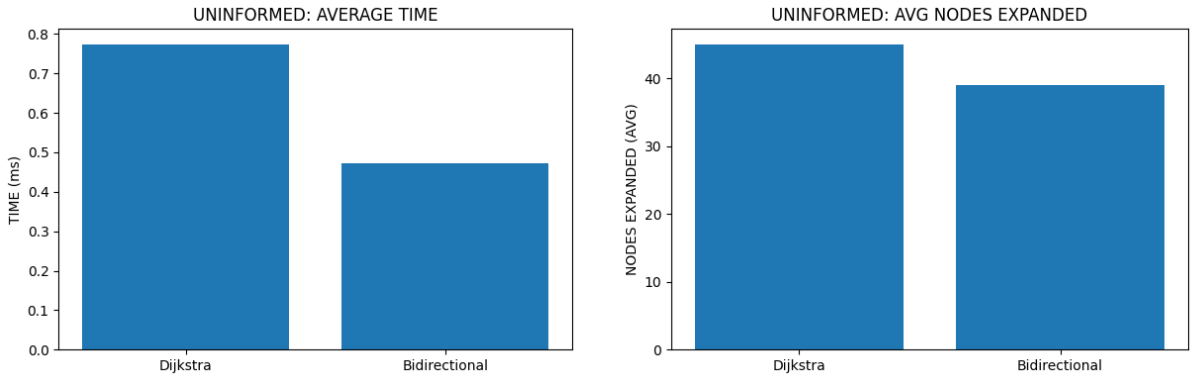
[1]Medidos com *tracemalloc*, diz respeito à memória virtual alocada e controlada pelo interpretador da linguagem *Python*, não necessariamente com endereços físicos.

Como mostrado na Tabela 2, o algoritmo bidirecional de fato leva um pouco menos de tempo para executar, em concordância, também, com a quantidade de nós explorados, que é

menor em relação ao algoritmo de *Dijkstra*. Além disso, a memória utilizada pelo algoritmo bidirecional durante a execução também teve picos de utilização maiores, o que está de acordo com a teoria estudada. Vale ressaltar que o problema testado é um caso leve, portanto, quanto maior o tamanho do espaço de estados e custo da solução ótima, maiores serão essas diferenças entre os algoritmos - assintoticamente.

Comparação Gráfica do Desempenho

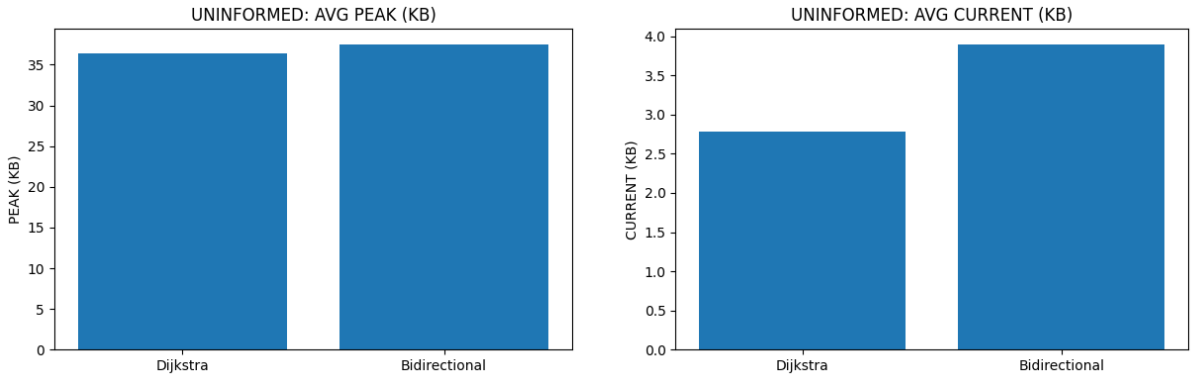
As Figuras 5(a) e 5(b) mostram a comparação entre tempo e nós expandidos dos dois algoritmos e a relação entre essas duas grandezas. Provando, assim, pela análise gráfica, que o algoritmo bidirecional é levemente mais eficiente em questão de tempo de execução e de nós expandidos para esse caso.



((a)) Comparação de Tempo de Execução ((b)) Comparação de Nós Explorados

Figura 5 – Comparação de Tempo/Nós Explorados

Já as Figuras 6(a) e 6(b) mostram a comparação entre os algoritmos para seus respectivos: pico de memória utilizada durante a execução e memória utilizada ao final da execução. Em ambos os casos, percebe-se uma maior utilização para o algoritmo bidirecional.



((a)) Comparação de Pico de Utilização de Memória ((b)) Comparação de Utilização de Memória ao Final

Figura 6 – Comparação de Memória Gasta Pico/Final

O gráfico da Figura 7 demonstra essa relação entre tempo e memória em conjunto. Ele evidencia que o algoritmo de *Dijkstra* gasta um pouco mais de tempo para executar e consome um pouco menos de memória que o bidirecional.

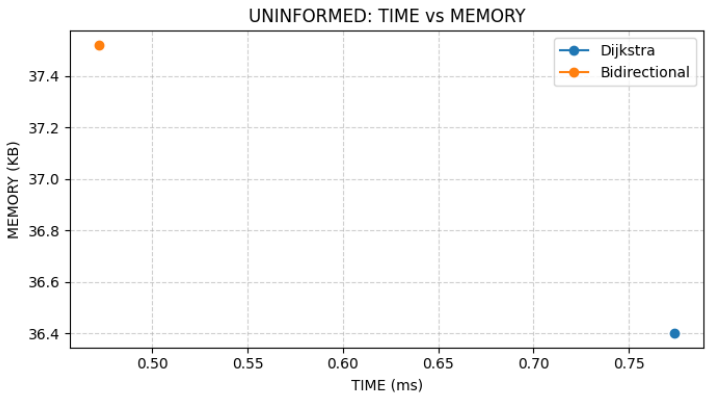


Figura 7 – Gráfico de Tempo vs Memória

Portanto, dentro do ambiente de testes utilizado, e com os parâmetros controlados do problema do labirinto, tende-se a fortalecer a ideia de que o algoritmo de busca bidirecional com custo uniforme é mais eficiente que o algoritmo de *Dijkstra*. Porém, vale ressaltar que há diversas variáveis que influenciam no tempo de execução e uso de memória, tais como os processos rodando no computador, o SO utilizado, e o *hardware* em que foram testados. Assim, para casos como esses, não é possível chegar a tantas conclusões apenas com casos de teste simples, que demonstraram variações entre as execuções até mesmo em uma só máquina.

Buscas Informadas

Algoritmos Utilizados

Os algoritmos de busca informada selecionados para este estudo foram a Busca Gulosa (*Greedy Best-First Search*) e o A*. Ambos utilizam heurísticas para guiar a busca em direção ao objetivo, porém o fazem de maneiras distintas, o que impacta diretamente a otimalidade e o desempenho de cada um. Foram utilizadas três heurísticas distintas para avaliar o comportamento dos algoritmos: a distância de Manhattan, a distância Euclidiana e uma heurística inadmissível.

Busca Gulosa

A Busca Gulosa é uma especialização da busca de melhor custo (*Best-First Search*) que seleciona o próximo nó a ser expandido baseando-se unicamente no valor da função heurística, $h(n)$. A função de avaliação do algoritmo é, portanto, definida como $f(n) = h(n)$, onde $h(n)$ é uma estimativa do custo do nó n até o nó objetivo.

A implementação utiliza uma fila de prioridade mínima (*min-heap*) para gerenciar a fronteira, garantindo que o nó que aparenta estar mais próximo do objetivo seja sempre o escolhido para expansão. Adicionalmente, uma tabela de dispersão (*hash table*) armazena os nós já alcançados (*reached*) para evitar a exploração de estados repetidos e a ocorrência de ciclos. A estratégia "gulosa"do algoritmo visa encontrar uma solução rapidamente, potencialmente sacrificando a otimalidade do caminho encontrado.

Busca $A*$

O algoritmo A* também é uma forma de busca de melhor custo que aprimora a estratégia da Busca Gulosa. Sua função de avaliação considera tanto o custo real do caminho percorrido desde o início até o nó atual, $g(n)$, quanto o custo estimado (heurístico) do nó atual até o objetivo, $h(n)$. A função é definida como $f(n) = g(n) + h(n)$.

Assim como na Busca Gulosa, a implementação do A* emprega uma fila de prioridade mínima para a fronteira e uma tabela de dispersão para os nós explorados. Ao combinar o custo já percorrido com a estimativa futura, o A* equilibra a exploração de caminhos que são promissores tanto em termos de custo acumulado quanto de proximidade com o objetivo. Se a heurística $h(n)$ for admissível — ou seja, nunca superestimar o custo real para alcançar o objetivo —, o A* garante encontrar a solução de menor custo.

Heurísticas Utilizadas

As heurísticas utilizadas neste estudo têm o objetivo de estimar o custo restante até o objetivo, influenciando diretamente o comportamento e a eficiência dos algoritmos de busca. Foram

implementadas três heurísticas distintas: a distância de Manhattan, a distância Euclidiana e uma heurística inadmissível. Cada uma delas é apresentada a seguir, com sua respectiva definição e implementação.

Distância Manhattan

A heurística de Distância Manhattan calcula o custo estimado entre dois pontos considerando apenas movimentos horizontais e verticais (4-direções). É amplamente utilizada em grades onde o movimento diagonal não é permitido, pois reflete com precisão o custo real de deslocamento em tais ambientes.

```
# MANHATTAN DISTANCE HEURISTIC (4-DIRECTIONAL MOVEMENT)
def h_manhattan_distance(a: Pos, b: Pos) -> float:
    return abs(a[0] - b[0]) + abs(a[1] - b[1])
```

No contexto do algoritmo, o valor dessa heurística é pré-calculado para todas as posições do ambiente antes da execução da busca, sendo armazenado em uma tabela de coordenadas. Assim, durante a expansão de cada nó, o valor de $h(n)$ é apenas recuperado dessa tabela, sem necessidade de novo cálculo. Esse método reduz o custo computacional e garante consistência entre as estimativas de diferentes nós.

Distância Euclidiana

A heurística de Distância Euclidiana utiliza a distância em linha reta entre dois pontos, sendo apropriada quando o movimento diagonal é permitido. É uma aproximação mais suave e contínua, que tende a gerar trajetórias mais realistas em espaços onde o deslocamento pode ocorrer em qualquer direção.

```
# EUCLIDEAN DISTANCE HEURISTIC (STRAIGHT-LINE DISTANCE)
def h_euclidean_distance(a: Pos, b: Pos) -> float:
    return ((a[0] - b[0]) ** 2 + (a[1] - b[1]) ** 2) ** 0.5
```

Assim como na distância de Manhattan, os valores dessa heurística também são pré-computados e armazenados em uma tabela. Durante a execução do A*, o algoritmo consulta diretamente esses valores, o que torna o processo de expansão mais rápido e eficiente, sem necessidade de realizar cálculos repetidos.

Heurística Inadmissível

A heurística Inadmissível foi definida como o a distância vertical multiplicada por um fator 9, resultando em valores superestimados. Seu propósito é avaliar o comportamento dos algoritmos quando a função heurística viola a condição de admissibilidade, podendo gerar caminhos subótimos.

```
# INADMISSIBLE HEURISTIC (9 TIMES THE DIFFERENCE OF LINE COORDINATES)
def h_inadmissible(a: Pos, b: Pos) -> float:
    return abs(a[0] - a[1]) * 9
```

Apesar de apresentar um cálculo diferente, essa heurística segue a mesma abordagem das demais: todos os valores são determinados previamente e armazenados em uma estrutura de tabela. Dessa forma, o A* acessa diretamente o valor de cada estado sem precisar recalcular a função heurística a cada expansão.

Comparação

Análise Assintótica

A Tabela 3 abaixo teve suas informações retiradas de RUSSELL (2021).

Tabela 3 – Comparação entre os Algoritmos Informados

Critério	Busca Gulosa/Greedy BFS	A*
Completude	Sim	Sim
Otimalidade	Não	Sim
Tempo	$O(\|V\|)$ ou $O(bm)$	$O\left(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor}\right)$
Espaço	$O(\|V\|)$ ou $O(bm)$	$O\left(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor}\right)$

($\boldsymbol{V}$: vértices do problema; $\boldsymbol{b}$: fator de ramificação; $\boldsymbol{d}$: profundidade da solução mais rasa ou $\boldsymbol{m}$ quando não há solução; $\boldsymbol{m}$: profundidade máxima da árvore de busca; $\boldsymbol{C^*}$: custo da solução ótima; $\boldsymbol{\epsilon}$: limite inferior de custos das ações)

Segundo Russel, o algoritmo A* será ótimo em custo caso a heurística utilizada seja admissível. Caso contrário, não é possível afirmar nada sobre a sua otimalidade. Já sobre sua completude, é possível afirmar que ele sempre encontra a solução quando ela existe. Por outro lado, para a busca gulosa, sua completude depende se o espaço de estados é finito ou não, sendo incompleto caso este seja infinito. Em contrapartida, ele não é ótimo em relação ao custo, já que leva em conta apenas a heurística de cada estado, o que pode desconsiderar características da solução ótima, em troca de encontrar uma outra solução mais rapidamente.

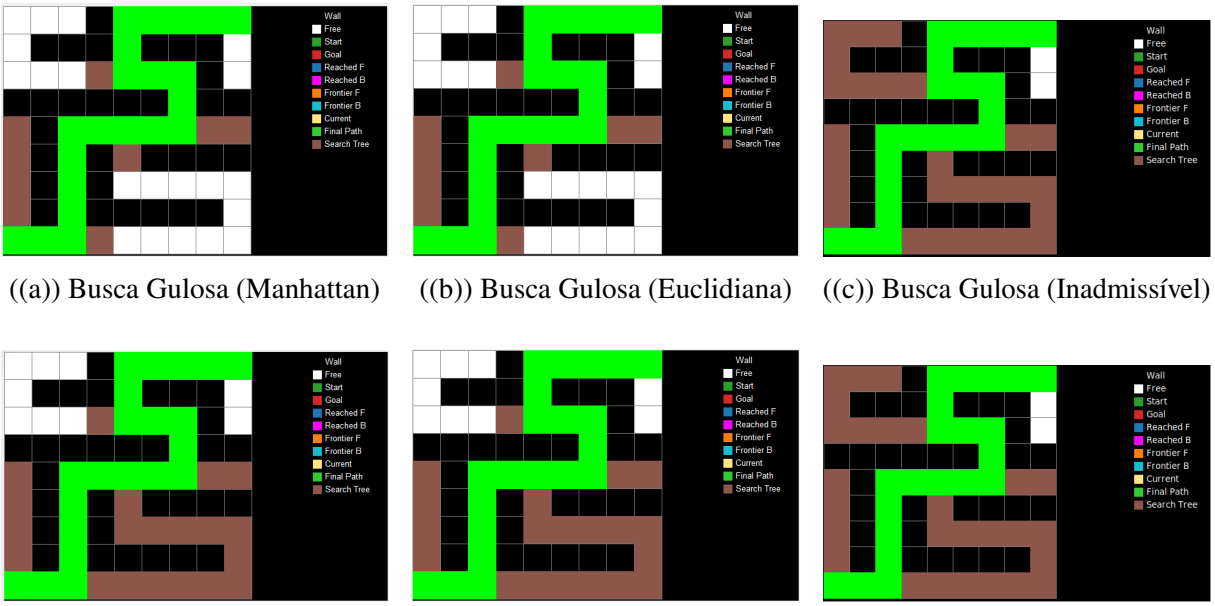
No pior caso, o algoritmo de busca gulosa terá de procurar em todo o espaço de estados, portanto é $O(|V|)$ em tempo e espaço. Ou, então, poderá atingir $O(bm)$ caso a heurística utilizada seja boa e admissível. Já o A*, possui a mesma complexidade do algoritmo de busca uniforme (*Dijkstra*) e depende do custo da solução ótima para determinar sua complexidade de tempo e memória. O cálculo da heurística, apesar de contabilizado, não influencia tanto no tempo de execução, já que a estrutura do algoritmo é muito parecida com a do algoritmo de busca uniforme não informada.

Soluções Encontradas

Em cenários reais, as heurísticas utilizadas podem guiar os algoritmos por caminhos distintos. A Busca Gulosa, por não possuir a característica de otimalidade, pode encontrar soluções com custos diferentes dependendo da heurística. O A*, quando utilizado com heurísticas admissíveis

(Manhattan e Euclidiana), encontrará uma solução ótima, embora o caminho exato possa variar. Com a heurística inadmissível, o A* perde sua garantia de otimalidade e pode se comportar de maneira imprevisível. Para o problema em questão, como há apenas uma solução, todos os algoritmos encontraram a solução ótima.

As figuras a seguir, 8, exemplificam as soluções encontradas e as árvores de busca para cada algoritmo e heurística.



((a)) Busca Gulosa (Manhattan) ((b)) Busca Gulosa (Euclidiana) ((c)) Busca Gulosa (Inadmissível)

((d)) A* (Manhattan) ((e)) A* (Euclidiana) ((f)) A* (Inadmissível)

Figura 8 – Árvores de Busca dos Algoritmos Informados

Comparação de Desempenho Real

A Tabela 4 abaixo conterá as métricas de desempenho obtidas a partir da execução dos algoritmos. Para cada combinação de algoritmo e heurística, foram realizadas 15 execuções, e os valores médios foram calculados.

Tabela 4 – Comparação entre os Algoritmos Informados - Desempenho Real

	Busca A*			**Busca Gulosa**		
Critério	**Manhattan**	**Euclidiana**	**Inadmissível**	**Manhattan**	**Euclidiana**	**Inadmissível**
Tempo de Execução (ms)	0,472	0,498	0,642	0,344	0,367	0,482
Nós Explorados	41,0	41,0	56,0	29,0	29,0	47,0
[1]Memória Máxima Gasta (KB)	36,259	36,449	36,777	36,145	36,145	36,145
[1]Memória Gasta ao Final (KB)	2,638	2,828	3,156	2,523	2,523	2,523
Custo da Solução	20,0	20,0	20,0	20,0	20,0	20,0
Soluções Encontradas	15/15	15/15	15/15	15/15	15/15	15/15

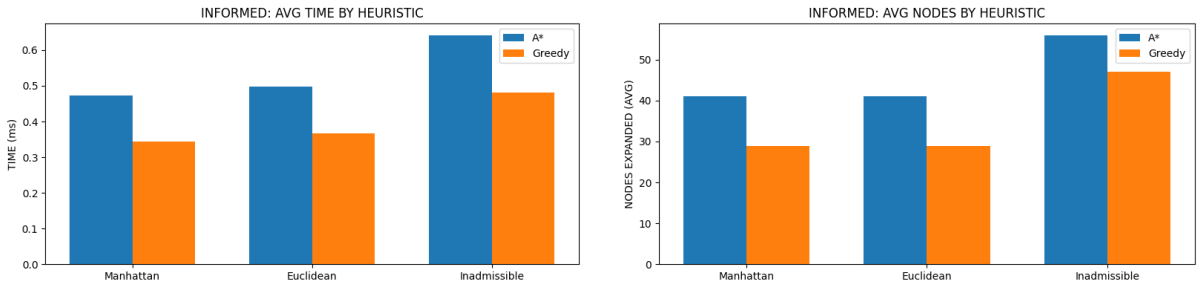
[1]Medidos com *tracemalloc*, diz respeito à memória virtual alocada e controlada pelo interpretador da linguagem *Python*, não necessariamente com endereços físicos.

A Tabela 4 mostra que a Busca Gulosa é geralmente mais rápida e explora menos nós, pois avança diretamente em direção ao que a heurística indica como o melhor caminho. No entanto, deve-se atentar ao fato de que isso não significa que a solução da Busca Gulosa é sempre ótima, apesar

de ter sido nesse caso. O A*, por outro lado, explora mais nós para garantir a otimalidade (com heurísticas admissíveis), resultando em um tempo de execução e uso de memória maiores. A heurística inadmissível leva ambos os algoritmos a comportamentos menos eficientes, explorando mais nós que o necessário e resultando em um maior gasto computacional. No caso, como existe apenas uma solução, aquela encontrada com essa heurística também foi ótima em ambos os algoritmos. Porém, em um labirinto maior, ou com uma dispersão de paredes que permite soluções não ótimas, ela poderia performar de maneira menos eficiente.

Comparação Gráfica do Desempenho

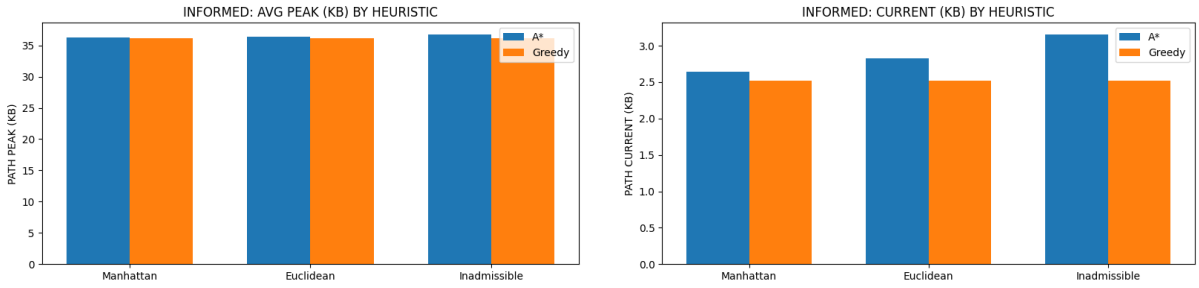
Os gráficos a seguir, representados pelas Figuras 9(a) e 9(b), ilustram visualmente a relação entre o tempo gasto para execução e a quantidade de nós explorados. Em ambos, conclui-se que, de fato, a heurística atrapalhou o funcionamento dos algoritmos, visto que realizam passos desnecessários, que em um cenário real poderiam refletir em uma solução não ótima encontrada.



((a)) Comparação de Tempo de Execução ((b)) Comparação de Nós Explorados

Figura 9 – Comparação de Tempo/Nós Explorados

Já as Figuras 10(a) e 10(b), mostram o pico de consumo de memória durante a execução, e a quantidade de memória utilizada ao final. É possível perceber que durante a execução não houve tanta alteração, apenas para o A*, que por natureza expande mais nós que o de Busca Gulosa.



((a)) Comparação de Pico de Utilização de Memória ((b)) Comparação de Utilização de Memória ao Final

Figura 10 – Comparação de Memória Gasta Pico/Final

Já o gráfico da Figura 11 demonstra a relação entre o tempo de execução e a memória utilizada, permitindo uma análise do *trade-off* entre esses dois recursos e a garantia de otimalidade da solução entre as duas abordagens.

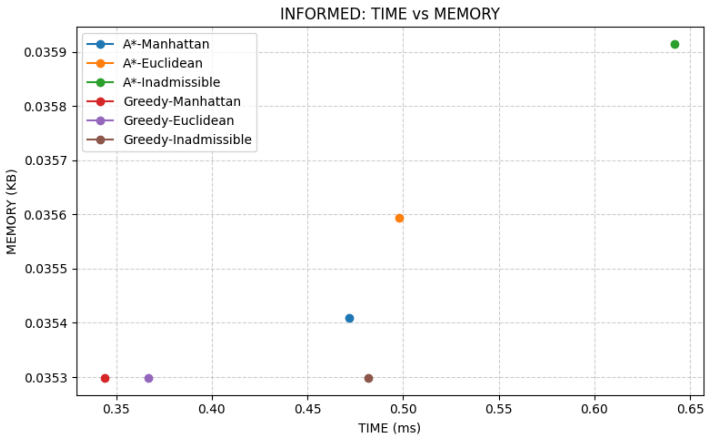


Figura 11 – Gráfico de Tempo vs Memória

A análise dos algoritmos de busca informada, Busca Gulosa e A*, revela um claro *trade-off* entre a velocidade de busca e a otimalidade da solução. A Busca Gulosa, priorizando exclusivamente a estimativa heurística, tende a encontrar uma solução de forma mais rápida, porém sem qualquer garantia de que este seja o caminho de menor custo. Sua eficiência é atrativa em problemas onde uma solução "boa o suficiente" e rápida é preferível a uma solução ótima que demande mais tempo e recursos.

Por outro lado, o A*, ao balancear o custo real percorrido ($g(n)$) e o custo estimado futuro ($h(n)$), estabelece um padrão de otimalidade, contanto que a heurística seja admissível. Essa garantia, no entanto, geralmente implica em um maior consumo de tempo e memória, uma vez que o algoritmo precisa explorar um número maior de nós para se certificar de que nenhum caminho mais curto foi deixado para trás. A qualidade da heurística é fundamental: uma heurística mais precisa (próxima do custo real, mas ainda admissível) pode reduzir drasticamente o espaço de busca do A*, tornando-o mais eficiente. O uso de uma heurística inadmissível quebra a garantia de otimalidade do A*, podendo degradar seu desempenho.

Portanto, a escolha entre a Busca Gulosa e o A* depende intrinsecamente dos requisitos do problema: para otimalidade, o A* com uma heurística admissível é a escolha correta; para velocidade, a Busca Gulosa pode ser uma alternativa viável.

Conclusão

A realização deste trabalho permitiu compreender de forma prática e teórica o funcionamento dos algoritmos de busca informada e não informada aplicados à resolução de labirintos, por meio da modelagem do problema como um grafo, da implementação dos algoritmos e da coleta de métricas de desempenho.

Nos algoritmos de busca não informada, observou-se que tanto o *Dijkstra* quanto o *BFS* Bidirecional apresentaram completude e otimalidade, encontrando sempre a solução de menor custo quando existente. Contudo, o *BFS* Bidirecional demonstrou um desempenho superior em relação ao tempo de execução e ao número de nós explorados, confirmando o esperado teoricamente: ao dividir a busca em duas frentes, reduz-se significativamente o espaço de estados a ser explorado. Em contrapartida, verificou-se um maior consumo de memória, consequência direta da manutenção simultânea das duas fronteiras e das duas listas de nós alcançados. Essa relação evidencia o clássico *trade-off* entre tempo e espaço, cuja escolha depende do contexto e das restrições computacionais do problema.

Já nos algoritmos de busca informada, as diferenças entre o A* e a Busca Gulosa tornaram-se evidentes. A Busca Gulosa, ao utilizar apenas o valor heurístico h(n), mostrou-se mais rápida e econômica em termos de memória, mas sem garantia de otimalidade. Em contraste, o A* combina o custo acumulado g(n) com a estimativa heurística h(n), garantindo soluções ótimas sempre que a heurística é admissível. Essa característica, entretanto, vem acompanhada de maior tempo de execução e maior consumo de memória, uma vez que o algoritmo precisa explorar mais nós para garantir que nenhum caminho de menor custo seja negligenciado.

A análise experimental confirmou ainda a importância da escolha da heurística: heurísticas mais próximas do custo real, mas sem superestimá-lo, tornam o A* mais eficiente; enquanto heurísticas inadmissíveis, que violam essa condição, degradam o desempenho e comprometem a otimalidade tanto do A* quanto da Busca Gulosa. Dessa forma, observou-se que a precisão e a admissibilidade da heurística são fatores determinantes para o equilíbrio entre desempenho e qualidade da solução.

De modo geral, a comparação entre todas as abordagens demonstrou que nenhum algoritmo é universalmente melhor, sendo cada um mais adequado a diferentes tipos de problema. Para aplicações em que a velocidade é prioritária e uma solução aproximada é aceitável, a Busca Gulosa e o BFS Bidirecional são alternativas eficazes. Já para situações em que a otimalidade e a consistência são essenciais, o A* e o Dijkstra se destacam, especialmente em ambientes com custos variáveis ou onde a precisão é crítica.

Por fim, o desenvolvimento deste projeto possibilitou não apenas a análise quantitativa dos resultados, mas também uma compreensão do comportamento dos algoritmos em termos de exploração, custo e recursos. A implementação, a geração de gráficos comparativos e a visualização das execuções permitiram observar de forma clara o impacto de cada escolha

teórica no desempenho prático. Assim, conclui-se que os objetivos do trabalho foram plenamente alcançados, evidenciando a importância do raciocínio heurístico e do balanceamento entre eficiência e otimalidade na área de Inteligência Artificial.

Créditos e Declaração de Autoria

Autores e Papéis

Guilherme Alvarenga de Azevedo:

1. **Implementação dos Algoritmos**:
    - **Busca Não Informada**: *Busca Uniforme* (ou *Dijkstra*) e *BFS* Bidirecional;
    - **Busca Informada**: Busca Gulosa (ou *Greedy BFS*);
    - **Experimentos**: Função principal + Interação com Terminal. Medição de tempo/memória e Comparação dos Algoritmos Não Informados;
    - **Visualização**: Geração de GIF's (corrigido/melhorado com IA);
2. **Relatório**:
    - Iniciou a seção de *Problema*, descrevendo parte da sua introdução;
    - Elaborou a explicação teórica das buscas *Informada* e *Não Informada*;
    - Redigiu parcialmente a seção de *Buscas Informadas*;
    - Contribuiu significativamente na *Conclusão*, redigindo boa parte dessa seção.

Maria Eduarda Teixeira Souza:

1. **Implementação dos Algoritmos**:
    - **Busca Informada**: Busca A*;
    - **Experimentos**: Função principal + Interação com Terminal. Comparação dos Algoritmos Informados e Não Informados, geração dos gráficos para Comparação;
    - **Visualização**: *GUI* do programa. Carregar arquivos TXT, selecionar opções, ver os GIF's gerados e visualizar problema em Grafo (corrigido/melhorado com IA);
2. **README.md**: Redigiu o `README.md` do preojeto que se encontra no diretório referente a esse trabalho (`/ia-trabalhos/trabalho1`). Além de colocar as dependências do projeto no `requirements.txt`.
3. **Relatório**:
    - Organizou a seção de *Problema*, separando-a em subseções e estruturando sua apresentação, desenvolveu as partes de *Descrição*, *Organização do Projeto* e *Modelagem*;
    - Na seção de *Buscas Informadas*, adicionou as imagens, inseriu as informações das métricas na tabela e reformulou alguns trechos do texto;
    - Na *Conclusão*, realizou a leitura crítica e correção de partes do conteúdo.

Uso de IA

1. **Adição de comentários**: *Copilot*;
2. **Visualização em GIF's**: *Copilot*, *ChatGPT*;
3. **GUI do programa**: *Copilot*, *ChatGPT*, *Gemini*;

Referências - Recursos Externos

[1] AZEVEDO, Guilherme A. SOUZA, Maria E. T. **IA-Trabalhos**: Trabalho 1 - Labirinto. 2025. Disponível em: <https://github.com/dudatsouza/ia-trabalhos> Acesso em: 19 out. 2025.

[2] OLIVEIRA, Tiago A. **Inteligência Artificial 04**: Estruturas e Estratégias de Busca. Slides de Aula. 2025.

[3] RUSSELL, Stuart J; NORVIG, Peter. **Inteligência Artificial: Uma Abordagem Moderna**. 4. ed. Pearson, 2021.

Declaração

Confirmamos que o código entregue foi desenvolvido pela equipe, respeitando as políticas da disciplina. As ferramentas de Inteligência Artificial generativa foram utilizadas para comentar no código já feito e, também, melhorar partes que não seriam avaliadas de acordo com o enunciado do trabalho. Tais como a **Visualização em GIF's** e **GUI do programa**.