Centro Federal de Educação Tecnológica de Minas Gerais
Curso Superior de Engenharia da Computação
Disciplina: Inteligência Artifical

Relatório para Trabalho de Busca no Labirinto (Não Informada e Informada)

João Pedro Rodrigues Silva
Jader Oliveira Sila
Prof. Thiago Alves de Oliveira

Divinópolis/MG
Outubro de 2025

1 Introdução:

A busca em espaços de estado é uma técnica fundamental em Inteligência Artificial para resolução de problemas de navegação e planejamento. Neste trabalho implementamos e comparamos algoritmos de busca *não informada* (por exemplo, Breadth-First Search) e *informada* (Busca Gulosa e A*) aplicados ao problema do labirinto apresentado no enunciado. Avaliamos aspectos essenciais: completude, optimalidade, tempo de execução, uso de memória e número de nós expandidos. Para as buscas informadas adotamos heurísticas apropriadas ao grid (por exemplo, distância de Manhattan e euclidiana), discutindo os efeitos de admissibilidade e consistência sobre A* e sobre a qualidade das soluções encontradas [2, 3].

2 Fundamentação Teórica

Figura 1: Labirinto utilizado nos experimentos (Start em U, Goal em E). Paredes grossas representam obstáculos intransponíveis.

2.1 Representação do problema

O problema do labirinto é modelado como um *espaço de estados* discreto sobre uma grade (grid) 2D. Cada célula livre da grade corresponde a um estado $s = (x, y)$. As paredes (obstáculos) são células inacessíveis e, portanto, removidas do espaço de busca.

Elementos do formalismo:

- **Estado inicial** s_0: célula marcada como "Start" (no exemplo: U).
- **Estado objetivo** s_g: célula marcada como "Goal" (no exemplo: E).
- **Ações**: movimentos entre células vizinhas (por exemplo, {Norte, Sul, Leste, Oeste} em uma grade 4-conectada). Cada ação tem custo $c(s, a, s')$. Neste trabalho consideramos custo uniforme $c = 1$ por passo.

- **Função sucessora**: gera estados válidos vizinhos que não sejam paredes.
- **Caminho**: sequência de ações que leva de s_0 a s_g.

2.2 Algoritmos de busca estudados

Implementamos e comparamos algoritmos clássicos de busca usados em IA, adaptando para o problema do labirinto:

Breadth-First Search (BFS). Explora a fronteira em ordem crescente de profundidade (camadas). Para grafos com custos iguais por passo, BFS encontra o caminho com o menor número de passos (optimalidade em número de passos) e é completo para grafos finitos. Complexidade em tempo e memória: $O(b^d)$, onde b é o fator de ramificação e d a profundidade da solução.

Depth-First Search (DFS). Explora tão profundamente quanto possível cada ramo antes de retroceder (backtracking). Existem duas formas comuns:

- *Tree-search* (recursiva, sem conjunto de visitados): pode não ser completa se houver caminhos infinitos ou ciclos.
- *Graph-search* (com conjunto `closed` de visitados): evita reexplorar nós e, em grafos finitos, torna-se completa.

Características importantes do DFS:

- **Completude:** DFS como tree-search não é garantidamente completa em espaços infinitos ou com ciclos; como graph-search (com controle de visitados) é completa em grafos finitos.
- **Optimalidade:** não é ótimo — o caminho encontrado não necessariamente minimiza custo nem número de passos.
- **Complexidade:** tempo $O(b^m)$ no pior caso (onde m é a profundidade máxima da árvore de busca), mas **espaço** é $O(b \cdot m)$ (pilha de recursão ou pilha explícita), tipicamente bem menor que BFS/A*.
- **Variações:** *Iterative Deepening DFS (IDDFS)* combina completude de BFS com a baixa memória do DFS, executando DFS com profundidade limitada crescente.

Busca Gulosa (Greedy Best-First Search). Usa uma heurística $h(n)$ e expande o nó que tem menor $h(n)$. Rápida e com consumo de memória geralmente menor; não garante optimalidade nem completude em todos os casos.

A* (A-estrela). Expande nós por ordem de $f(n) = g(n) + h(n)$. Com heurística admissível (e preferivelmente consistente) A* é ótimo e completo (em grafos finitos e com custos positivos). Oferece bom trade-off entre tempo e qualidade da solução.

2.3 Heurísticas para grade

Duas heurísticas comuns para grids 4-conectados:

- **Distância de Manhattan:**

$$(h_{\text{man}}(n) = |x_n - x_g| + |y_n - y_g|). \tag{1}$$

– Admissível para movimentos ortogonais unitários; consistente.

- **Distância Euclidiana:**

$$(h_{\text{euc}}(n) = \sqrt{(x_n - x_g)^2 + (y_n - y_g)^2} \tag{2}$$

– Admissível; pode ser menos informativa que Manhattan em grids estritamente ortogonais.

Observação: **DFS não usa heurística** — é uma busca não informada; heurísticas aplicam-se a Greedy e A*.

2.4 Propriedades importantes (comparativo)

- **Completude:** BFS e A* (com condições usuais) são completos; DFS é completo apenas se implementado como busca em grafo em grafos finitos (ou com limitação de profundidade adequada).
- **Optimalidade:** BFS é ótimo quando passos têm custo igual; A* é ótimo se h for admissível. DFS e Greedy não são ótimos em geral.
- **Complexidade:** BFS/A* têm memória $O(b^m)$ (problema em grids maiores). DFS tem vantagem espacial $O(b \cdot m)$, o que permite aplicar em labirintos maiores quando a memória é restrita, porém à custa de possíveis caminhos subótimos e maior tempo em certos casos.

2.5 Pseudocódigos

A implementação real segue a mesma lógica com estruturas adequadas (fila para BFS, pilha para DFS, fila de prioridade para A*/Greedy). Abaixo pseudocódigos compactos.

BFS (grafo):

```
BFS(s0):
  frontier <- queue(); frontier.enqueue(s0)
  explored <- {}
  parent[s0] <- null
  while not frontier.empty():
    n <- frontier.dequeue()
    if IsGoal(n): return ReconstructPath(parent,n)
    explored.add(n)
    for each neighbor n' of n:
      if n' not in explored and n' not in frontier:
        parent[n'] <- n
        frontier.enqueue(n')
  return failure
```

DFS (iterativa, versão com conjunto de visitados):

```
DFS(s0):
  frontier <- stack(); frontier.push(s0)
  explored <- {}
  parent[s0] <- null
  while not frontier.empty():
```

```
    n <- frontier.pop()
    if IsGoal(n): return ReconstructPath(parent,n)
    if n not in explored:
      explored.add(n)
      for each neighbor n' of n:
        if n' not in explored:
          parent[n'] <- n
          frontier.push(n')
  return failure
```

DFS (recursiva):

```
DFS-Rec(n):
  if IsGoal(n): return true
  mark n as visited
  for each neighbor n' of n:
    if n' not visited:
      parent[n'] <- n
      if DFS-Rec(n'): return true
  return false
```

A*:

```
A*(s0):
  open <- priority_queue() ordered by f=g+h
  open.insert(s0, f(s0))
  g[s0] <- 0; parent[s0] <- null
  closed <- {}
  while not open.empty():
    n <- open.pop()
    if IsGoal(n): return ReconstructPath(parent,n)
    closed.add(n)
    for each neighbor n' of n:
      tentative_g <- g[n] + 1
      if n' in closed and tentative_g >= g[n']: continue
      if n' not in open or tentative_g < g[n']:
        parent[n'] <- n
        g[n'] <- tentative_g
        f[n'] <- g[n'] + h(n')
        if n' not in open: open.insert(n', f[n'])
  return failure
```

Em resumo, a escolha entre DFS, BFS, Greedy e A* depende de restrições de memória, necessidade de optimalidade e características das instâncias (tamanho/obstáculos).

3 Metodologia:

A metodologia adotada descreve a representação do problema, a implementação dos algoritmos, a instrumentação das medições e o procedimento experimental para garantir reprodutibilidade dos resultados.

3.1 Representação do problema e formato de entrada

O labirinto é representado como um grafo implícito construído a partir de uma grade 2D. Cada célula transitável é identificada por uma posição (r, c) e associada a um vetor de 4 bits indicando as passagens possíveis nas direções Norte, Sul, Leste e Oeste. O parser lê um ficheiro de texto com linhas no formato:

```
[r,c]:1001
Start:[r,c]
Goal:[r,c]
```

Linhas podem conter comentários com rótulos (ex.: "# A"). Esta rotina e as convenções de adjacência foram extraídas de `src/maze.py`. [1]

3.2 Implementação dos algoritmos

Implementaram-se as versões clássicas de busca:

- **BFS (Breadth-First Search)** — busca em largura sobre o grafo (fila).
- **DFS (Depth-First Search)** — implementação iterativa com pilha e conjunto de visitados.
- **Gulosa (Greedy Best-First)** — prioriza nós com menor $h(n)$.
- **A* (A-estrela)** — prioriza por $f(n) = g(n) + h(n)$ mantendo g-scores para relaxação.

A lógica dos algoritmos, estrutura `Node` e reconstrutor de caminho encontram-se em `src/search.py`. Todas as funções retornam uma tupla contendo: sequência de ações (path), custo, memória máxima observada, nós expandidos e flag de solução encontrada. [1]

3.3 Heurísticas

Foram implementadas as heurísticas utilizadas pelas buscas informadas:

- **Manhattan (Eq. 1)**.
- **Euclidiana (Eq. 2)**.

As heurísticas estão em `src/heuristics.py` e são usadas por A* e pela Busca Gulosa. [1]

3.4 Instrumentação e métricas

As medições instrumentadas incluem:

1. **Tempo de execução:** medido com `time.perf_counter()` no script de controle principal (`main.py`). O tempo reportado é o tempo de clock entre início e término da chamada de busca. [1]
2. **Memória máxima (estimada):** medida como o máximo observado de elementos na fronteira mais o conjunto de visitados (por exemplo `len(frontier)+len(explored)` ou `len(frontier)+len(g_scores)`), conforme implementação do algoritmo. Esta

é uma métrica de custo em estruturas de dados (número de nós mantidos em memória), não uma medição do RSS do processo. Implementação visível em `src/search.py`. [1]

3. **Nós expandidos:** contador incrementado a cada nó retirado da fronteira para expansão (implementado em cada algoritmo).
4. **Custo do caminho:** soma dos custos de passos (por omissão cada passo tem custo unitário 1.0 — `step_cost` padrão). O `main.py` compara o custo obtido com uma referência de custo ótimo ajustável (`CUSTO_MINIMO_OTIMO`). [1]

3.5 Configuração experimental

- **Código-fonte:** arquivos principais usados — `main.py`, `src/search.py`, `src/maze.py`, `src/heuristics.py`. [1]
- **Ficheiro de instância:** por padrão o ficheiro lido é `data/labirinto.txt` (pode ser alterado no `main.py` ou passado como argumento onde aplicável).
- **Ambiente de execução:** recomenda-se Python 3.8+ (o código usa `time.perf_counter` e type hints). Registar CPU, RAM e SO na máquina experimental para contextualizar resultados.
- **Parâmetros de teste:** a lista de testes padrão em `main.py` inclui: BFS, DFS, Gulosa (Manhattan) e A* (Manhattan). O valor de referência `COSTO_MINIMO_OTIMO` pode ser ajustado conforme a instância.

3.6 Procedimento experimental

Para cada instância de labirinto:

1. Carregar o ficheiro de instância (ex.: `data/labirinto.txt`) e extrair posições `Start` e `Goal` via `Maze.from_file()`. Se `Start` ou `Goal` não existirem o teste é abortado.
2. Para cada algoritmo na lista de testes, executar a função de busca correspondente e recolher a tupla retornada (caminho, custo, memória máxima, nós expandidos, flag de solução). O controlador principal que executa esta sequência é `main.py`.
3. Armazenar os resultados em ficheiro texto padronizado (`medicoes_desempenho.txt`) e imprimir tabela resumo no terminal.

3.7 Reprodutibilidade e validade

- O experimento é determinístico (sem fontes de aleatoriedade no código atual), logo execuções repetidas na mesma instância devem produzir os mesmos resultados.
- Para robustez estatística recomenda-se repetir execuções em várias instâncias (labirintos) e, se possível, em máquinas diferentes, reportando médias e desvios-padrão quando apropriado.
- Atenção: a métrica de "memória máxima"é uma estimativa em número de nós (estruturas internas) e não substitui uma medição de uso real de memória do processo; se necessário, pode-se integrar `tracemalloc` ou `psutil` para medições de RSS/heap mais precisas.

3.8 Comandos de execução (exemplo)

```
# Executar todos os testes com par metros citados acima
python main.py

# Caso queira especificar outro ficheiro, edite o valor de 'lab_file' em
    main.py
# Output principal gerado em: medicoes_desempenho.txt
```

Os ficheiros e funções referenciadas nesta seção podem ser encontrados no repositório do projeto (ver referência bibliográfica) e nos ficheiros locais: `main.py`, `src/search.py`, `src/maze.py` e `src/heuristics.py`.

4 Resultados e Discussões:

Nesta sessão serão discutidas diferentes métricas de análise de resultados empregadas neste trabalho.

À começar pela análise de custos, neste trabalho admitiu-se que cada ação de caminhamento, isto é, cada movimento dentro da mátriz de labirinto, seja em qualquer direção admitida - Norte, Sul, Leste ou Oeste - é contabilizada como 1 de custo. Portanto, se o algoritmo utilizado teve custo 5, por exemplo, isto quer dizer que dentro do labirinto esse algortimo tomou 5 ações de movimento **para chegar ao destino final**. Além disso, utilizou-se também métricas de tempo de execução, calculada utilizando a biblioteca *time* do python, uso de memória, calculada aqui não propriamente como a memória gasta em termos de alocação, mas sim como a soma de nós visitados com a de nós na pilha que ainda precisam ser explorados. Ademais adotou-se também a soma dos nós expandidos. Diferente da métrica de custo, essa métrica faz referência a quantidade máxima de nós que o algortimo efetivamente visitou antes de encontrar o final do labirinto. Por fim, tem-se a métrica de transições, que faz referência à quantidade total de movimentos gerados a partir dos nós expandidos durante a busca.

Utilizando essas métricas, reuniu-se dados suficientes para construir a seguinte tabela:

Algoritmo	**Heurística**	**Custo**	**Tempo de Execução**	**Nós Expandidos**	**Memória**	**Transições**
BFS	Não Informada	10	$254 \cdot 10^{-3}$ s	22	25	22
DFS	Não Informada	10	$194 \cdot 10^{-3}$ s	18	25	20
Busca Gulosa	Manhattan	10	$202 \cdot 10^{-3}$ s	14	21	16
A*	Manhattan	10	$232 \cdot 10^{-3}$ s	20	23	20

Tabela 1: Métricas de execução dos algoritmos de busca utilizados..

Utilizando os dados expositivos presentes na tabela 1, serão feitos gráficos comparativos relacionados à cada coluna da mesma. Além disso, para realizar uma análise imparcial e o mais justa possível, os comparativos serão feitos entre os algoritmos **não informados** e **Buscas Gulosa e A***.

Com isto em mente, tem-se os seguintes gráficos:

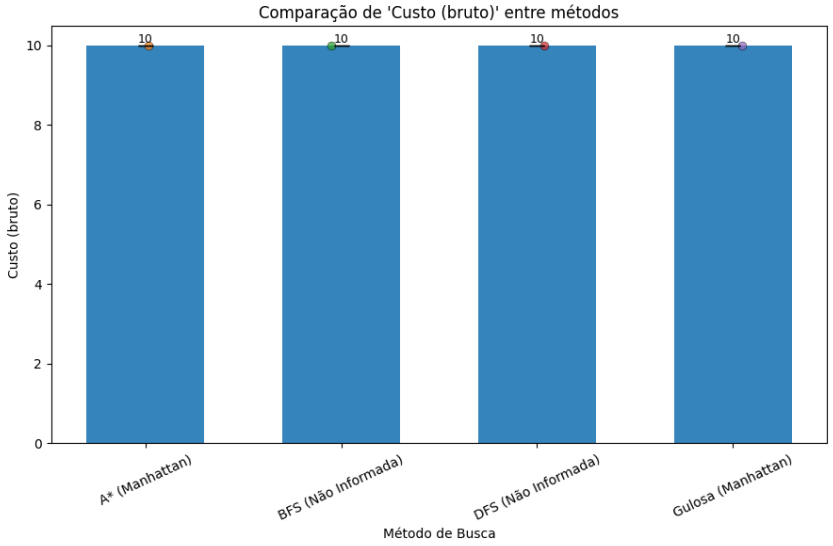

Figura 2: Relação gráfica entre os custos dos algoritmos.

Neste gráfico que relaciona o custo, isto é, a quantidade de movimentos realizados até chegar à saída, pode-se notar que a variação entre eles, para o labirinto de exemplo (Figura 1), foi inexistente. Esse fato pode ser explicado, e mais do que isso era esperado, pois para o labirinto usado há somente dois caminhos até a saída partindo de A. Porém, aliando essa possibilidade binária à forma como cada algoritmo funciona individualmente, é esperado que todos acabem tomando o mesmo caminho para chegar até a saída, e essa tendência é o que gerou a uniformidade dos resultados.

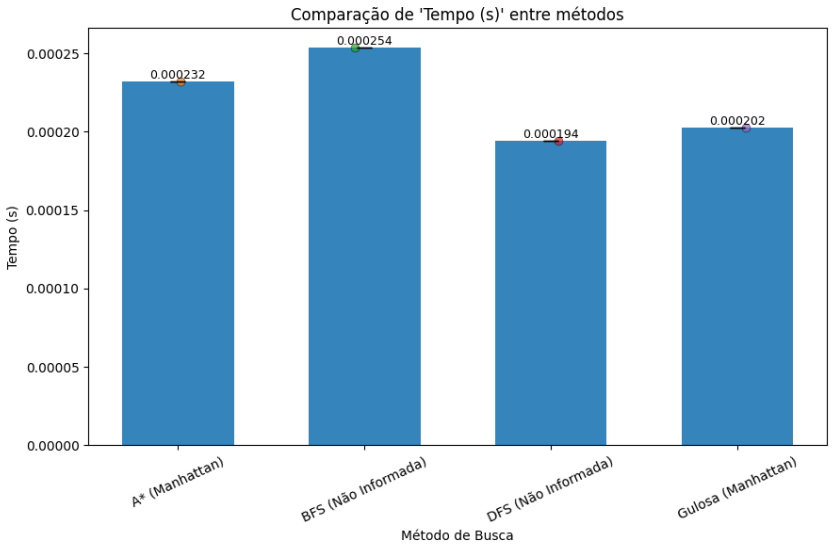

Figura 3: Relação gráfica entre o Tempo de execução dos algoritmos.

Partindo para uma análise temporal, aqui é possível notar algumas diferenças expressivas entre os algoritmos.

Não Informados

Para os algoritmos não informados, é evidente que há uma diferença entre o tempo de execução, diferença de 60 milisegundos em favor do **DFS**. Essa diferença é esperada dada

a forma como os dois algoritmos realizam a exploração do labirinto. O DFS realiza uma exploração em profundidade, o que possibilita neste caso encontrar a saída de forma mais rápida. O BFS, que utiliza uma abordagem de exploração concêntrica em relação à entrada, acaba explorando mais possibilidades de caminho, o que em última instância gera um tempo de execução maior.

Informados

Para os algoritmos informados, tem-se o melhor tempo para o algoritmo de busca gulosa, cujo tempo de execução foi de 202 milisegundos, 30 milisegundos mais rápido que o algoritmo A*. Assim como o caso anterior, este também tem uma boa explicação. O algoritmo de busca gulosa não tenta procurar um caminho ótimo - ele garante um caminho. Neste caso, mecânicas de busca do melhor caminho, como as que existem na busca A*, acabam por tornar a execução mais lenta. Isso explica porque a execução da busca gulosa é mais rápida que o A*.

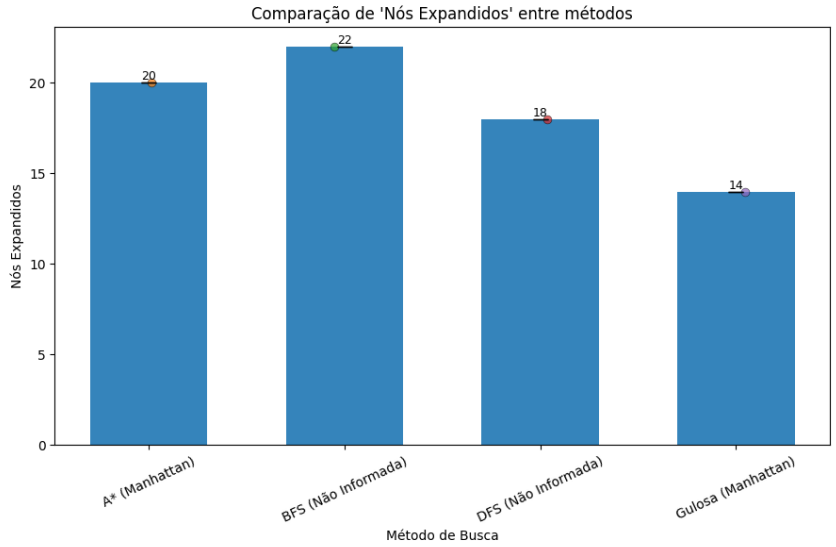

Figura 4: Relação gráfica entre a quantidade de nós expandidos por cada algoritmo.

Realizando uma análise norteada pela quantidade de nós expandidos, nota-se novamente uma heterogeneidade nos dados.

Não informados

Para os algoritmos não informados, nota-se que o BFS apresenta uma quantidade de nós expandidos maior que o DFS. Esse fato corrobora com os pontos citados anteriormente na análise do tempo de execução, demonstrando novamente a natureza do algoritmo de busca em largura, que expande sua busca de forma concêntrica em relação ao nó origem, ação mais custosa que busca garantir soluções sempre ótimas. O mesmo pode ser dito para o DFS, algoritmo de busca em profundidade, que expande menos nós, justamente por sua natureza de busca profunda, mas instável e que não garante solução ótima.

Informados

Semelhante ao que pode ser visto para os algoritmos de busca não informada nota-se, aqui, a tendência do algoritmo A* em abrir mais nós. Análogo ao algoritmo de busca em largura,

o algoritmo de busca A* procura uma solução ótima, e isso envolve necessariamente uma busca mais elaborada. Desta forma, esse algoritmo acaba por expandir mais nós que o de busca gulosa, que não se proprõe a devolver uma solução ótima. Evidentemente, este fato corrobora para que seu tempo de execução seja maior e, para o algoritmo de busca gulosa, menor.

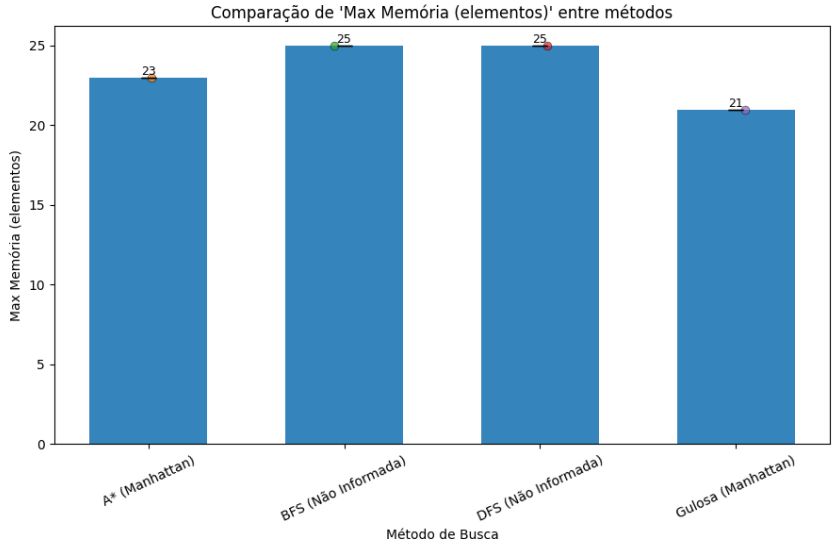

Figura 5: Relação gráfica entre a memória utilizada por cada algoritmo.

Neste gráfico, é possível ver a relação entre o uso de memória, mensurada aqui como a soma entre os nós visitados e os nós na pilha aguardando visita.

Não informados

Para os algoritmos de busca não informados, tem-se aqui dados não esperados quando baseia-se a análise simplesmente pelo método com o qual os algoritmos funcionam. Tanto o BFS quanto o DFS utilizaram a mesma quantidade de memória. Neste caso, é possível extender a análise pensando sobre a forma como o labirinto foi estruturado. Este labirinto, que possui uma grid 5x5, tem várias paredes. Além disso, existe somente uma entrada para a saída do labirinto.

Com isto em mente, sabe-se que a forma como ambos os algoritmos realizam suas buscas sofrem alterações que, em última instância, afetam seu uso de memória. Caso fosse um labirinto sem paredes, por exemplo, seria esperado que o BFS utilizasse mais memória compara do DFS.

Informados

Para os algoritmos de busca informados, tem-se certa uniformidade nos dados, com uma diferença de 2 unidades em relação aos dois algoritmos. De modo semelhante à análise feita para os algoritmos não informados, aqui a mesma situação se aplica. É esperado que o algoritmo de busca A* explorasse mais nós, de modo que a memória máxima utilizada seja maior. Isso aconteceu de fato, porém com uma diferença pouco expressiva, fato que pode ser explicado pela situação citada anteriormente.

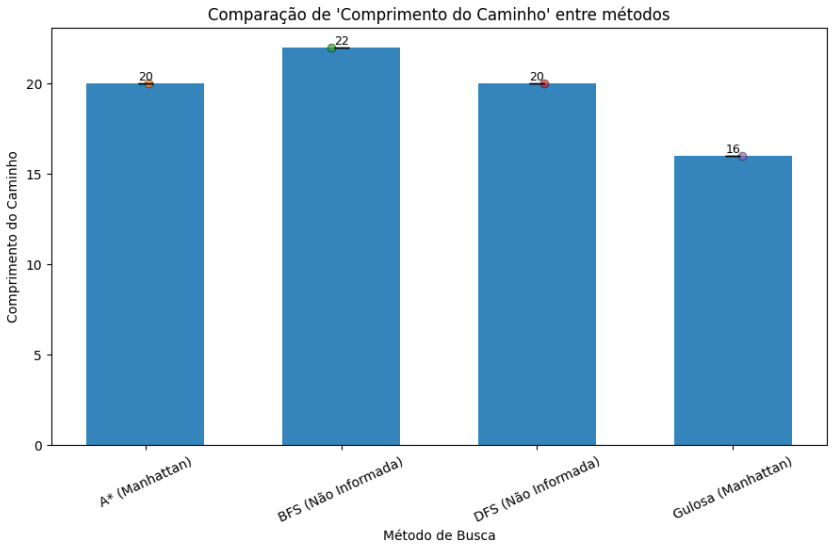

Figura 6: Relação gráfica entre a quantidade de transições feitas em cada algoritmo..

Finalizando as análises gráficas, tem-se a medida de comprimento do caminho, tratada aqui como a coluna de transições, que faz referência à quantidade total de movimentos gerados a partir dos nós expandidos durante a busca.

Não informados

Para os algoritmos de busca não informados, ao analisar a quantidade de transições, é possível observar que ambos apresentaram valores muito próximos entre si, com uma pequena vantagem para o DFS, que realizou duas transições a menos que o BFS. Assim como vem sendo relatado, esse comportamenteo se explica pelas caracteristicas únicas de cada algoritmo. Por realizar uma busca em largura, o BFS acaba explorando mais caminhos antes de alcançar a solução, e isso justifica sua quantidade maior de transições.

Informados

Para os algoritmos informados, os valores foram, também, muito próximos, com uma vantagem para o algoritmo de Busca Gulosa, que apresentou somente 16 transições, enquanto o A*, 20. Novamente, essas caracteristicas corroboram com as características únicas de cada algoritmo. A busca gulosa, por considerar somente o nó mais próximo do algoritmo, tende a realizar menos transições, mesmo que não garanta uma solução ótima.

Finalmente, a análise conjunta das métricas de custo, tempo de execução, nós expandidos, memória e transições nos permitem compreender, de forma mais completa, o comportamento individual desses algoritmos em diferentes condições. Embora todos tenham completude, isto é, foram capazes de alcançar o objetivo, esta análise individual permite entender como as estratégias de exploração resultam em diferentes desempenhos quando analisados sob perspectivas distintas.

5 Conclusões:

Em primeiro lugar, é seguro concluir que os algoritmos de busca se apresentam como soluções eficazes na solução de problemas do estilo labirinto. A observação individual de cada métrica utilizada traz luz ao fato de que a escolha do algoritmo tem um impacto

profundo no desempenho computacional. Cada um pode fornecer qualidades distintas, ao mesmo tempo que apresenta problemas, e a escolha do algoritmo certo e da heurística correta demandam estudo profundo e um conhecimento claro daquilo que se deseja obter, de forma que seja possível determinar o conjunto que fornecerá o resultado que o usuário espera.

Em segundo lugar, os resultados obtidos mostram que, em cenários onde o custo de caminhamento é uniforme e o caminho é bem definido a métrica de custo tende a ser equivalente, independente do algoritmo utilizado, e isso é, inclusive, esperado. Além disso, a análise individual das métricas expostas aqui confimam que não existe um "melhor"algoritmo, mas sim um conjunto de características que devem ser observadas para guiar a escolha do método que, dentro de cada contexto individual, pode fornecer a melhor solução, tendo em conta diversas caracteristicas.

Por fim, este trabalho contribuiu para aprofundar a compreensão do comportamento individual de cada algoritmo de busca, oferecendo uma visão mais clara sobre cada um.

6 Créditos e Autoria:

Essa seção tem-se como foi feita a divisão de tarefa dentro do grupo, adjunto aos recursos externos utilizados (que estarão explicitados dentro das referências).

6.1 João Pedro Rodrigues Silva (Autor)

- Implementação inicial dos metódos de busca informada e não informada (baseado nos materiais disponibilizados pelo Professor). [5]
- Documentação inicial do relatório (utilizando-se de fontes bibliográficas para embasamento teórico). [2, 3]
- Questionamentos e elucidações com o Professor orientador do projeto.

6.2 Jader Oliveira Silva (Autor)

- Revisão dos algoritmos de busca implementados e aprimorando as métricas utilizadas. [1]
- Revisão e finalizamento do relatório acerca do trabalho.
- Questionamentos e elucidações com o Professor orientador do projeto.

Uso de IA: A ferramenta utilizada foi o ChatGPT apenas para revisão textual ortográfica. Nenhuma parte de código de algoritmos foi gerada por IA.
Declaração: Confirmamos que o código entregue foi desenvolvido pela equipe, respeitando as politicas da disciplina.

Referências

[1] João Pedro Rodrigues e Jader Oliveira, *MAZE-SEARCH*, repositório no GitHub(Jottynha), 2025. Disponível: `https://github.com/Jottynha/MAZE-SEARCH`. Acesso em: 3 de out. 2025.

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

[3] P. E. Hart, N. J. Nilsson e B. Raphael, "A formal basis for the heuristic determination of minimum cost paths", *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[4] A. Thiago, *Capítulo III: Conhecimento*, Sistema Integrado de Gestão de Atividades Acadêmicas. CEFET-MG. Acesso em set. 2025.

[5] A. Thiago, *Capítulo IV: Busca*, Sistema Integrado de Gestão de Atividades Acadêmicas. CEFET-MG. Acesso em set. 2025.