Trabalho 1: Análise Comparativa de Algoritmos de Busca em Labirinto

Rafael Adolfo Silva Ferreira
Engenharia de Computação
Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG)

Resumo: Este relatório apresenta a implementação e a análise de desempenho de algoritmos de busca não informada (BFS e DFS) e informada (Gulosa e A*), aplicados à resolução de um problema de labirinto. As métricas de custo da solução, nós expandidos, memória e tempo de execução são comparadas para avaliar a completude, optimalidade e eficiência de cada abordagem.

Introdução

O problema de busca por caminhos é um dos mais fundamentais na área de Inteligência Artificial, com aplicações que vão desde a logística e o planejamento de rotas até a resolução de jogos e quebra-cabeças. Este trabalho foca na resolução de um labirinto, um problema clássico de espaço de estados, utilizando e comparando diferentes estratégias de busca.

O objetivo principal é avaliar empiricamente as características de algoritmos não informados, que exploram o espaço de busca de forma sistemática, e algoritmos informados, que utilizam heurísticas para guiar a busca de forma mais eficiente. As métricas de desempenho analisadas são: completude, optimalidade, tempo de execução, memória e nós expandidos.

Metodologia

Para a resolução do problema, foram implementados quatro algoritmos de busca principais, além de três funções de heurística para as buscas informadas.

Métricas

Para o critério de avaliação foi utilizado as metricas de `custo` que no caso significa o caminho encontrado pra o objetivo, `esforço de busca` se referindo ao número de nós utilizados ao longo da execução para encontrar o caminho, `memória` que é o número máximo de nós expandidos simultâneamente durante a execução do algoritmo e `tempo de execução` que serve de metrica para comparar qual algoritmo executou mais rapidamente a tarefa.

Foi utilizado um labirinto de maior área e com mais de um caminho possível para poder avaliar de forma mais evidente as diferenças entre cada algoritmo.

Material utilizado

O código foi feito em `Python` utilizando somente com biblioteca externa o `matplotlib` para impressão de gráficos, outras bibliotecas utilizadas foram as padrões da versão `3.18` do `Python`.

O hardware onde o código foi executado:

- **Modelo:** Notebook Lenovo Gaming 3 15IMH05
- **Processador:** Intel(R) Core(TM) i5-10300H (8) 4.50 GHz
- **GPU 1:** NVIDIA GeForce GTX 1650 Mobile / Max-Q [Discrete]
- **GPU 2:** Intel UHD Graphics 1.05 GHz [Integrated]
- **Memória RAM**: 16 GB.

- **Memória Disco**: 1 TB SSD.

Algoritmos Não Informados

Ambos os algoritmos foram implementados em Python, utilizando estruturas de dados da biblioteca padrão.

Busca em Largura (BFS)

A Busca em Largura (BFS), conforme apresentada por Cormen et al. (2022), é um algoritmo projetado para atravessar ou pesquisar estruturas de dados em árvore ou grafo de maneira sistemática. A sua abordagem fundamental consiste em explorar o grafo "em camadas". Partindo de um vértice de origem `s`, o algoritmo primeiro visita todos os vizinhos diretos de `s`. Subsequentemente, ele visita todos os vizinhos ainda não visitados desses vizinhos, e assim por diante. Este método de exploração em ondas assegura que o primeiro caminho encontrado para um vértice de destino seja o mais curto em termos de número de arestas. Para gerir a ordem de visitação dos vértices, o BFS emprega uma estrutura de dados de fila (FIFO - First--In, First-Out).

Algoritmo 1: BFS

```
1  function BFS(G, s)
2      for each vertex u in G.V - {s}
3          u.color = WHITE
4          u.d = ∞
5          u.π = NIL
6      end
7      s.color = GRAY
8      s.d = 0
9      s.π = NIL
10     Q = ∅
11     ENQUEUE(Q, s)
12     while Q is not empty
13         u = DEQUEUE(Q)
14         for each vertex v in G.Adj[u]
15             if v.color == WHITE
16                 v.color = GRAY
17                 v.d = u.d + 1
18                 v.π = u
19                 ENQUEUE(Q, v)
20             end
21         end
22     u.color = BLACK
23 end
```

Algoritmo 1: Breadth-First Search (BFS). Source: Cormen et al. (2022)[1].

Analisando o pseudocódigo Algoritmo 1:

- **Inicialização:** O algoritmo inicializa cada vértice do grafo, atribuindo-lhe uma cor (`WHITE` para não descoberto), uma distância infinita da origem (`d = infinity`), e um predecessor nulo (`pi = NIL`). A origem `s` é marcada como `GRAY` (descoberta), a sua distância é definida como 0, e é adicionada a uma fila `Q`.
- **Loop Principal:** Enquanto a fila não estiver vazia, o algoritmo retira um vértice `u` e explora os seus vizinhos. Para cada vizinho `v` ainda não descoberto (`color == WHITE`), o algoritmo atualiza a sua cor para `GRAY`, a sua distância (`u.d + 1`), o seu predecessor (`pi = u`), e adiciona-o à fila. Após explorar todos os seus vizinhos, o vértice `u` é marcado como `BLACK` (totalmente explorado).

Adaptações da Implementação

A implementação em Python do BFS para a resolução do labirinto aplica a lógica central do pseudocódigo de Cormen, mas com adaptações pragmáticas para otimizar e simplificar o código para esta tarefa específica.

- **Representação Implícita do Grafo:** Em vez de uma estrutura de dados de grafo explícita, o labirinto em si funciona como uma representação implícita do grafo. Os "vértices" são as coordenadas `(linha, coluna)` e as "arestas" são os movimentos válidos para as células adjacentes, que são obtidos através da função `maze.get_neighbors(current)`.
- **Controlo de Vértices Visitados:** O sistema de cores (`WHITE`, `GRAY`, `BLACK`) é substituído por uma única estrutura de dados, o dicionário `came_from`. A verificação `if neighbor not in came_from:` serve o mesmo propósito que `if v.color == WHITE`. Um nó está em `came_from` se já foi descoberto; caso contrário, é novo. Esta é uma simplificação eficiente e comum em problemas de busca de caminho.
- **Rastreio de Predecessores:** O dicionário `came_from` acumula a função do atributo `pi` do pseudocódigo. A linha `came_from[neighbor] = current` armazena o predecessor de cada nó visitado, permitindo a reconstrução do caminho no final da busca.
- **Cálculo de Custo e Condição de Parada:** O algoritmo é otimizado para parar assim que encontra o objetivo (`if current == goal_node`), em vez de explorar todo o grafo alcançável. O custo do caminho, que corresponde à distância `d` no pseudocódigo, não é armazenado por nó durante a busca. Em vez disso, é calculado no final, simplesmente contando os passos no caminho reconstruído (`len(path) - 1`), uma vez que cada passo tem um custo uniforme de 1.

Busca em Profundidade (DFS)

A Busca em Profundidade (DFS), do inglês Depth-First Search, é um algoritmo que explora um grafo seguindo um caminho o mais longe possível antes de retroceder (backtracking). Diferentemente do BFS, que explora em camadas, o DFS mergulha num ramo do grafo até atingir um beco sem saída (um nó sem vizinhos não visitados) ou o objetivo. Só então ele retorna para o nó anterior e explora um caminho alternativo. Este comportamento, que lembra o desenrolar de um novelo de lã, é naturalmente implementado com uma estrutura de dados de pilha (LIFO - Last-In, First-Out), seja de forma explícita (com uma pilha) ou implícita (através da recursão).

Algoritmo 2: DFS

```
1  function DFS(G)
2      for each vertex u in G.V
3          u.color = WHITE
4          u.π = NIL
5      end
6      time = 0
7      for each vertex u in G.V
8          if u.color == WHITE
9              DFS-VISIT(G, u)
10         end
11     end
12     function DFS-VISIT(G, u)
13     time = time + 1
14     u.d = time
15     u.color = GRAY
16     for each vertex v in G.Adj[u]
17         if v.color == WHITE
18             v.π = u
19             DFS-VISIT(G, v)
20         end
21     end
22     u.color = BLACK
23     time = time + 1
24     u.f = time
```

Algoritmo 2: Depth-First Search (DFS). Source: Cormen et al. (2022)[1].

O pseudocódigo de descreve uma versão recursiva completa:

- **Estrutura Principal (`DFS`):** A função principal inicializa todos os vértices com a cor `WHITE` (não descoberto) e predecessor `NIL`. Em seguida, itera sobre todos os vértices e, para cada um que ainda for `WHITE`, chama a função auxiliar `DFS-VISIT` para iniciar a exploração a partir daquele ponto.
- **Função de Visita (`DFS-VISIT`):** Esta é a parte recursiva do algoritmo. Quando um vértice `u` é visitado, ele é marcado como `GRAY` (descoberto) e um "timestamp" de descoberta (`u.d`) é registado. O algoritmo então itera sobre os vizinhos `v` de `u`. Se um vizinho `v` for `WHITE`, o seu predecessor (`pi`) é definido como `u` e `DFS-VISIT` é chamada recursivamente para `v`. Este passo aprofunda a busca. Após todos os vizinhos de `u` terem sido explorados (ou seja, após o retorno de todas as chamadas recursivas), `u` é marcado como `BLACK` (totalmente explorado) e um "timestamp" de finalização (`u.f`) é registado.

Adaptações da Implementação

A implementação em Python para o problema do labirinto utiliza uma abordagem iterativa do DFS, que é funcionalmente equivalente à recursiva, mas mais adequada para grafos muito profundos onde a recursão poderia exceder o limite da pilha de chamadas do sistema.

- **Implementação Iterativa com Pilha Explícita:** Em vez de chamadas de função recursivas, o código utiliza uma lista (`stack`) que funciona como uma pilha explícita. O loop `while stack:` continua enquanto houver nós a serem explorados. A operação `stack.pop()` retira o último nó adicionado, imitando o comportamento LIFO da recursão.
- **Simplificação do Controlo de Visita:** Assim como na implementação do BFS, o sistema de cores é simplificado. O dicionário `came_from` é usado para rastrear tanto os nós visitados como os seus predecessores. A condição `if neighbor not in came_from:` substitui a verificação `if v.color == WHITE`.
- **Ausência de Timestamps:** Os timestamps de descoberta e finalização (`d` e `f`) são características úteis para outras aplicações de DFS (como ordenação topológica), mas são desnecessários para a simples tarefa de encontrar um caminho. Portanto, foram omitidos para simplificar o código.
- **Foco no Caminho:** O algoritmo é otimizado para o seu objetivo. Ele termina imediatamente quando o nó de destino (`goal_node`) é encontrado (`if current == goal_node:`), em vez de continuar a explorar o resto do grafo, o que torna a busca mais eficiente para este problema específico.

Algoritmos Informados

Estes algoritmos utilizam uma função de heurística $h(n)$ para estimar o custo de um nó até o objetivo.

Busca Gulosa (*Greedy Best-First Search*)

A Busca Gulosa é um algoritmo de busca informada que utiliza uma função heurística para tomar decisões. A sua estratégia é sempre expandir o nó que parece estar mais próximo do objetivo, de acordo com o valor fornecido pela heurística. O termo "gulosa" reflete o seu comportamento de fazer a escolha localmente ótima (o nó com o menor custo heurístico), na esperança de que isso leve rapidamente a uma solução global. Diferentemente de algoritmos como o A*, a Busca Gulosa não considera o custo do caminho já percorrido para chegar a um nó; a sua decisão é baseada unicamente na estimativa do custo restante até o destino. Isso a torna rápida e eficiente em termos de memória, mas não garante que o caminho encontrado seja o mais curto.

Algoritmo 3: Greedy Search

```
1  function GREEDY-BEST-FIRST-SEARCH(problem)
2      node = Node(problem.INITIAL-STATE)
3      frontier = PriorityQueue(node, h(node))
4      reached = { problem.INITIAL-STATE: node }
5      while frontier is not empty
6          node = POP(frontier)
7          if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
8          for each child in EXPAND(node, problem)
9              s = child.STATE
10             if s is not in reached or child.PATH-COST < reached[s].PATH-COST
11                 reached[s] = child
12                 ADD(frontier, child, h(child))
13             end
14         end
15     end
16 return failure
```

Algoritmo 3: Greedy Best-First Search. Source: Russell & Norving et al. (2020) [2]

O pseudocódigo ilustra este processo:

- **Estruturas de Dados:** O algoritmo utiliza uma fila de prioridade (`frontier`) para armazenar os nós a serem explorados, ordenada pelo valor da função heurística `h`. Um conjunto ou mapa (`reached`) é usado para registar os nós já alcançados, evitando ciclos e trabalho redundante.
- **Loop Principal:** O algoritmo retira repetidamente o nó com a menor prioridade (menor valor `h`) da `frontier`. Se este nó for o objetivo, a busca termina com sucesso. Caso contrário, os seus vizinhos (filhos) são expandidos. Para cada vizinho, se ele ainda não foi alcançado ou se um caminho mais curto foi encontrado para ele (esta parte é mais relevante para outros algoritmos como A*), ele é adicionado à `frontier` com a sua prioridade definida pelo seu valor heurístico.

Adaptações da Implementação

A implementação em Python traduz eficientemente a lógica da Busca Gulosa para a resolução do labirinto.

- **Fila de Prioridade:** A lista `priority_queue`, gerida com o módulo `heapq` do Python, implementa a fila de prioridade (`frontier`). Ela armazena tuplos `(priority, neighbor)`, garantindo que `heapq.heappop` sempre retorne o nó com a menor prioridade (menor valor heurístico).
- **Rastreio de Nós Visitados:** O dicionário `came_from` desempenha o papel do conjunto `reached`, registando os nós que já foram adicionados à fronteira de busca e, ao mesmo tempo, armazenando os seus predecessores para a reconstrução do caminho. A condição `if neighbor not in came_from:` é uma forma simplificada e eficaz de verificar se um nó já foi visitado.

- **Função Heurística como Parâmetro:** A implementação é flexível, permitindo que a função heurística (`heuristic`) seja passada como um argumento. Isto desacopla a lógica da busca da heurística específica utilizada (por exemplo, a distância de Manhattan), facilitando a experimentação com diferentes estratégias de guia.
- **Simplificação:** A verificação `child.PATH-COST < reached[s].PATH-COST` do pseudocódigo é omitida. No contexto de um labirinto onde todos os movimentos têm custo uniforme (1), a primeira vez que um nó é alcançado por esta busca gulosa é suficiente; não há necessidade de reavaliar caminhos para nós já visitados, pois um caminho mais curto para eles não será encontrado desta forma.

A* (*A-Star*)

O algoritmo A* (pronuncia-se "A-estrela") é um dos mais eficientes e populares algoritmos de busca de caminho. Ele combina as vantagens de buscas como a de Dijkstra, que garante o caminho mais curto, com a velocidade de buscas gulosas, que usam heurísticas para se guiar. O A* alcança este equilíbrio através da sua função de avaliação, $f(n) = g(n) + h(n)$, que estima a "promessa" de cada nó `n`:

- $g(n)$**:** É o **custo real** do caminho desde o nó inicial até o nó `n`. Esta é a parte do algoritmo que garante a otimalidade, pois leva em conta o esforço já despendido.
- $h(n)$**:** É o **custo estimado** (heurística) do nó `n` até o nó objetivo. Esta é a parte "informada" do algoritmo, que o guia na direção mais promissora, tal como na busca gulosa.

Ao expandir sempre o nó com o menor valor de $f(n)$, o A* explora de forma inteligente o espaço de busca. Se a função heurística $h(n)$ for "admissível" (ou seja, nunca superestimar o custo real para alcançar o objetivo), o A* garante encontrar o caminho de menor custo.

Algoritmo 4: A* Search

```
1  function A-STAR-SEARCH(problem)
2      node = Node(problem.INITIAL-STATE)
3      frontier = PriorityQueue(node, g(node) + h(node))
4      reached = { problem.INITIAL-STATE: node }
5      while frontier is not empty
6          node = POP(frontier)
7          if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
8          for each child in EXPAND(node, problem)
9              s = child.STATE
10             if s is not in reached or child.PATH-COST < reached[s].PATH-COST
11                 reached[s] = child
12                 ADD(frontier, child, g(child) + h(child))
13             end
14         end
15     end
16 return failure
```

Algoritmo 4: A* Search. Source: Russell & Norvig et al. (2020) [2]

O pseudocódigo descreve um processo metódico para encontrar o caminho ótimo:

- **Estruturas de Dados:** O algoritmo utiliza uma fila de prioridade (`frontier`) para armazenar os nós a serem explorados, ordenada pela função de avaliação completa $f(n) = g(n) + h(n)$. Um mapa (`reached`) regista os nós já alcançados e o custo (`g(n)`) do melhor caminho encontrado até eles.
- **Loop Principal:** O algoritmo retira repetidamente o nó com a menor prioridade (menor valor $f(n)$) da `frontier`. Se este nó for o objetivo, a busca termina. Caso contrário, os seus vizinhos são expandidos.
- **Atualização de Caminhos:** Para cada vizinho, o algoritmo verifica se ele é novo ou se o caminho recém-descoberto para ele é mais curto (`child.PATH-COST < reached[s].PATH-COST`). Se uma destas condições for verdadeira, o nó é adicionado à `frontier` com a sua prioridade $f(n)$ atualizada. Este passo de reavaliação é o que garante que o A* encontre o caminho ótimo.

Adaptações da Implementação

A implementação em Python traduz fielmente a lógica do pseudocódigo:

- **Custo do Caminho `g(n)`:** O dicionário `cost_so_far` corresponde à informação de custo (`PATH-COST`) armazenada na estrutura `reached` do pseudocódigo. Ele guarda o valor de $g(n)$ para cada nó.
- **Fila de Prioridade `frontier`:** A lista `priority_queue`, gerida pelo módulo `heapq`, funciona como a `frontier`, sempre fornecendo o nó com o menor valor $f(n)$.
- **Atualização de Caminhos:** A condição `if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:` na implementação Python é a tradução direta da linha 10 do pseudocódigo. Ela garante que o algoritmo explore novos nós e, crucialmente, atualize o caminho para nós existentes se uma rota mais barata for descoberta, o que é a chave para a garantia de otimalidade do A*.

Funções de Heurística

Uma heurística, $h(n)$, é uma função que estima o custo do caminho mais barato de um estado `n` até um estado objetivo. Para que o algoritmo A* garanta a otimalidade, a heurística deve ser **admissível**, ou seja, nunca superestimar o custo real [2].

As três funções implementadas são exemplos clássicos de heurísticas para problemas de busca em grade. A escolha, a definição e as fórmulas a seguir são baseadas nos conceitos de heurísticas para busca de caminho apresentados em Russell e Norvig (2020) [2].

- **Distância Euclidiana:**

$$h(n) = \sqrt{(x_n - x_G)^2 + (y_n - y_G)^2} \tag{1}$$

  - Corresponde à "distância em linha reta", a heurística admissível mais comum para problemas de busca de rota.
  - Esta heurística é garantidamente **admissível**, pois nenhum caminho no labirinto pode ser mais curto que uma linha reta. No entanto, ela não é muito "informada" para este problema específico. Como os movimentos são restritos à grade (cima, baixo, esquerda, direita), o custo real será sempre maior ou igual à distância euclidiana.

- **Distância de Manhattan:**

$$h(n) = |x_n - x_G| + |y_n - y_G| \tag{2}$$

- Derivada da soma das distâncias absolutas nos eixos, é a heurística padrão para grades onde apenas movimentos em 4 direções (horizontal e vertical) são permitidos.
- Esta é, teoricamente, a **melhor heurística** para este labirinto. Como o movimento é restrito a 4 direções, a Distância de Manhattan representa exatamente o custo mínimo para ir de um ponto a outro **se não houvesse paredes**. Ela é admissível e a mais informada possível sem conhecer o layout do labirinto, pois se aproxima mais do custo real do que as outras.

- **Distância Diagonal (Chebyshev):**

$$h(n) = \max(|x_n - x_G|, |y_n - y_G|) \tag{3}$$

- Mede a distância considerando movimentos em 8 direções (incluindo diagonais), como um rei no xadrez.
- Esta heurística é admissível, mas não é ideal para um labirinto com apenas 4 direções de movimento. Ela assume que movimentos diagonais são possíveis e têm o mesmo custo que os movimentos retos. Como isso não é verdade no nosso caso, ela subestima o custo real do caminho. Ela é mais informada que a Euclidiana (pois `max(|dx|, |dy|)` é sempre maior ou igual a `sqrt(dx²+dy²)` ajustado para custo de movimento), mas menos informada que a de Manhattan. Portanto, seu desempenho deve ser intermediário entre as outras duas.

Resultados e Discussão

Os algoritmos foram executados no labirinto fornecido e as métricas de desempenho foram coletadas. Os resultados completos estão apresentados na Tabela 1.

Tabela 1: Tabela comparativa de desempenho dos algoritmos de busca.

Algoritmo	Custo	Nós Expandidos	Memória Máx.	Tempo (s)
BFS	60	206	207	0.0005
DFS	88	149	164	0.0003
Gulosa (Manhattan)	60	81	98	0.0002
Gulosa (Diagonal)	72	98	113	0.0006
Gulosa (Euclidiana)	60	77	94	0.0002
A* (Manhattan)	60	208	207	0.0009
A* (Diagonal)	60	206	207	0.0007
A* (Euclidiana)	60	206	207	0.0007

Com esses valores obtidos podemos analisa-los de forma gráfica.

Análise de Optimalidade e Completude

A optimalidade de um algoritmo refere-se à sua capacidade de garantir a solução de menor custo. Conforme exibido na Figura 1, o caminho ótimo para o labirinto testado tem um custo de 60 passos.

- Os algoritmos **BFS** e todas as três variações do **A*** encontraram a solução ótima de custo 60, confirmando suas propriedades teóricas de optimalidade para problemas com custo de passo uniforme.
- O **DFS** demonstrou sua natureza não ótima ao encontrar um caminho significativamente mais longo, com custo 88.

- A **Busca Gulosa** apresentou resultados mistos: enquanto as versões com heurística de Manhattan e Euclidiana encontraram o caminho ótimo (provavelmente por sorte, dada a topologia do labirinto), a versão com a heurística Diagonal encontrou um caminho subótimo de custo 72. Isso evidencia a falta de garantia de optimalidade do algoritmo guloso.

Todos os algoritmos implementados são completos para este tipo de problema (espaço de estados finito), ou seja, todos foram capazes de encontrar uma solução.

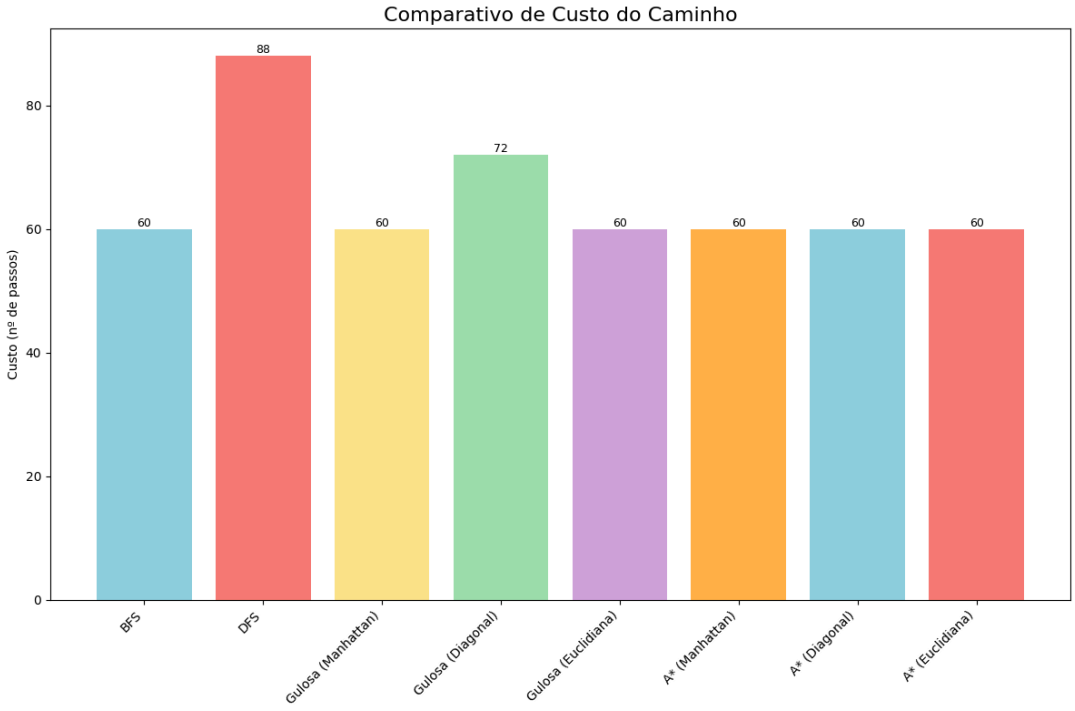

Figura 1: Comparativo de Custo da Solução (Optimalidade).

Análise de Eficiência da Busca

A eficiência de um algoritmo é medida pelo esforço computacional necessário para encontrar uma solução. As métricas de nós expandidos e memória máxima são essenciais para essa análise, pois são independentes do hardware.

Observando a Figura 2, a análise revela que as buscas gulosas, especialmente com a heurística Euclidiana (77 nós) e de Manhattan (81 nós), foram as mais eficientes em termos de esforço de busca. Elas rapidamente convergiram para uma solução, pois priorizam agressivamente os nós que parecem mais próximos do objetivo.

Um resultado notável é o desempenho do **A*** em comparação com o **BFS**. Ambos expandiram uma quantidade quase idêntica de nós (entre 206 e 208). Isso sugere que, para este labirinto, as heurísticas não foram informativas o suficiente para podar significativamente a árvore de busca. O A* precisou explorar quase todo o espaço relevante, assim como o BFS, para ter a garantia matemática de que o caminho encontrado era de fato o ótimo.

O uso de memória (Figura 3) seguiu um padrão similar, com o BFS e o A* consumindo a maior quantidade de recursos devido à grande fronteira de nós que precisaram manter.

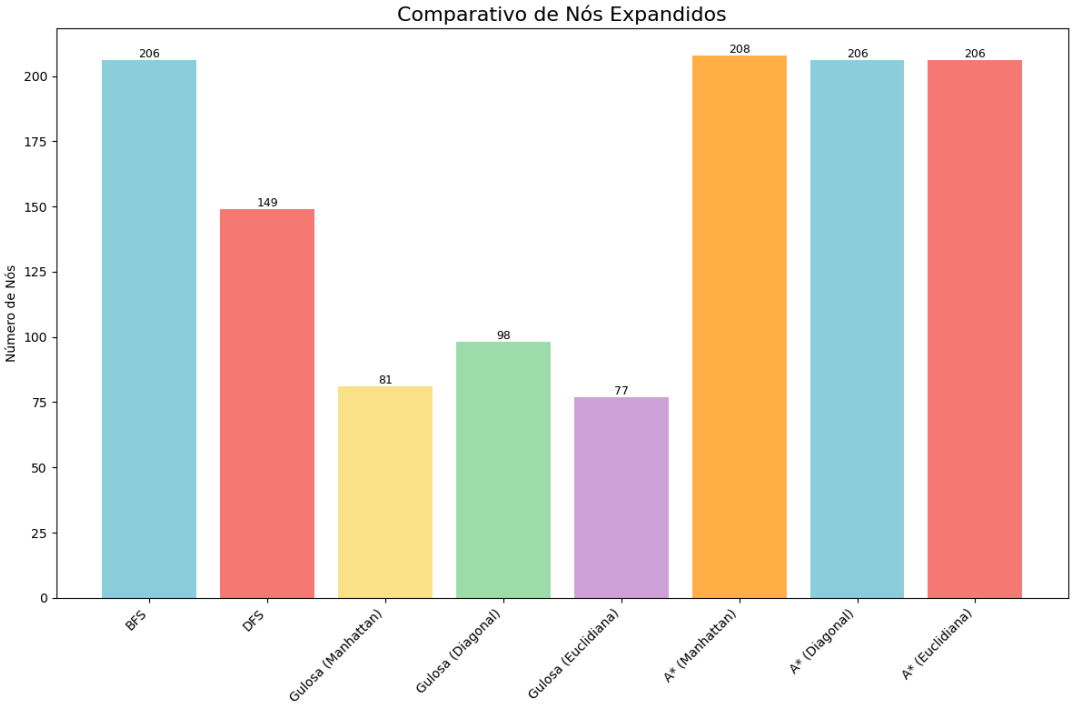

Figura 2: Comparativo de Nós Expandidos (Esforço da Busca).

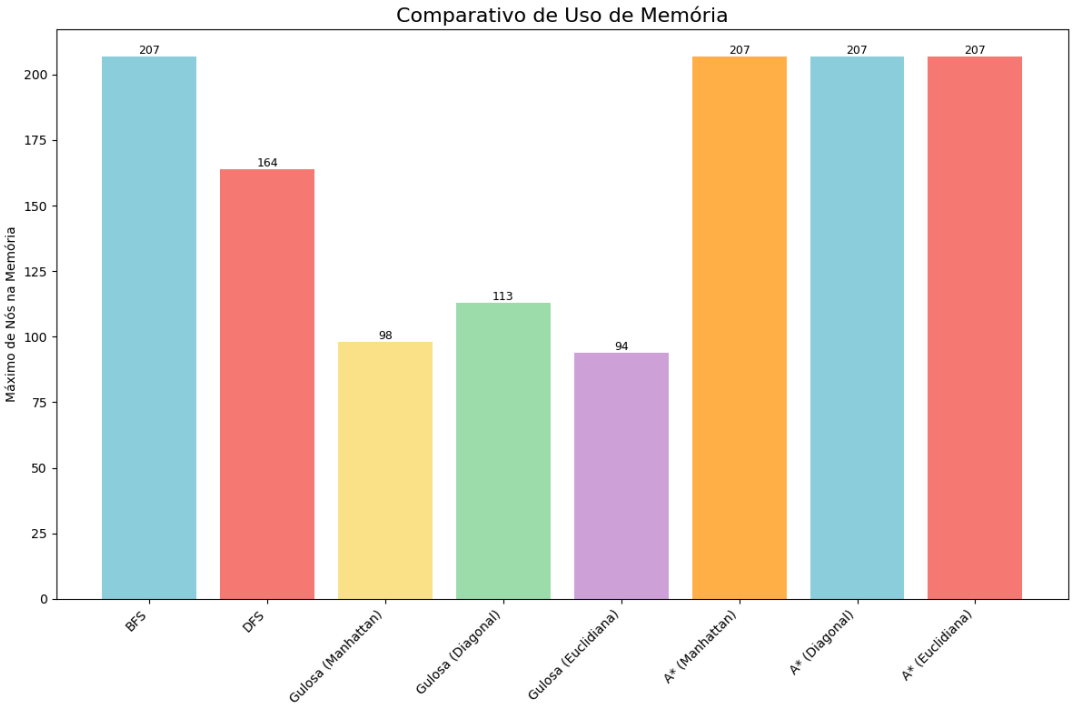

Figura 3: Comparativo de Uso de Memória.

Impacto da Heurística

A escolha da heurística impactou diretamente o comportamento dos algoritmos informados. Para a **Busca Gulosa**, a heurística Euclidiana não só levou à solução ótima, como também o fez com o menor esforço (77 nós). A heurística Diagonal, por outro lado, guiou a busca para um caminho mais longo e ineficiente.

No caso do **A***, as três heurísticas levaram ao mesmo resultado ótimo com esforço de busca praticamente idêntico. Isso indica que, embora todas sejam admissíveis e consistentes para este problema, nenhuma ofereceu uma vantagem decisiva sobre as outras em termos de redução do espaço de busca. O tempo de execução (Figura 4) também reflete essa análise, com as buscas gulosas sendo as mais rápidas devido ao menor número de nós processados.

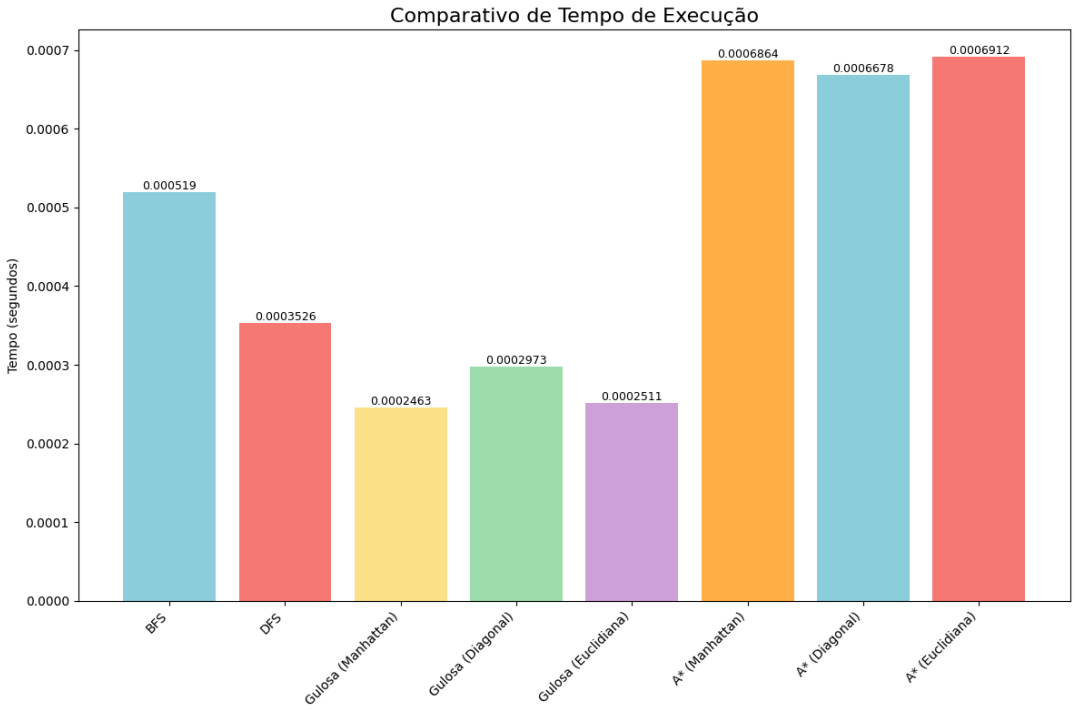

Figura 4: Comparativo de Tempo de Execução.

Conclusão

Este trabalho implementou e comparou com sucesso diversos algoritmos de busca na resolução de um labirinto. Os resultados práticos confirmaram as propriedades teóricas de cada algoritmo.

Conclui-se que existe um claro *trade-off* entre a garantia de optimalidade e a eficiência da busca. Algoritmos como o **BFS** e o **A*** garantem a melhor solução possível, mas essa garantia pode ter um custo computacional elevado, como observado pelo alto número de nós expandidos. Em contrapartida, a **Busca Gulosa** pode ser drasticamente mais rápida e leve, mas sacrifica a confiabilidade, podendo retornar soluções de qualidade inferior.

A análise também demonstrou que a eficácia de uma heurística no A* depende fortemente da estrutura do problema. Para o labirinto testado, as heurísticas não foram capazes de otimizar significativamente a busca em comparação com a abordagem "cega" do BFS. Portanto, a escolha do algoritmo ideal depende dos requisitos do problema: se a optimalidade for mandatória, A* ou BFS são as escolhas corretas; se uma solução "boa o suficiente" e rápida for aceitável, a Busca Gulosa pode ser uma alternativa viável.

Créditos e Declaração de Autoria

- **Uso de IA:** A ferramenta Gemini (Google) foi utilizada como assistente de programação para gerar trechos de código boilerplate (como a estrutura dos testes e a automação de gráficos), para depuração e para auxiliar na estruturação e revisão textual deste relatório. Nenhuma parte da lógica central dos algoritmos de busca foi gerada por IA.
- **Declaração:** Confirmo que o código entregue foi desenvolvido por mim, respeitando as políticas da disciplina.

Referências

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, e C. Stein, *Introduction to Algorithms*, 4th ed. The MIT Press, 2022.

[2] S. Russell e P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. em Pearson Series in Artifical Intelligence. Pearson, 2020.