Análise de Diferentes Algoritmos de Busca para o Quebra-Cabeça de 8

Gabriel Ast dos Santos

April 19, 2025

Abstract

Este trabalho apresenta uma análise comparativa de quatro algoritmos de busca aplicados ao problema do quebra-cabeça de 8 (8-puzzle): Busca em Largura (BFS), Busca em Profundidade (DFS), Busca Gulosa e A* (A-Star). O objetivo é avaliar o desempenho destes algoritmos em termos de eficiência computacional (tempo de execução e consumo de memória) e qualidade da solução (número de movimentos). Foram implementados os algoritmos em Python e testados em dez instâncias diferentes do problema, com complexidade variada. Os resultados demonstram que o algoritmo A* apresenta o melhor equilíbrio entre tempo de execução e qualidade da solução, encontrando sempre o caminho ótimo com baixo custo computacional. A Busca Gulosa mostrou-se competitiva em termos de velocidade, mas nem sempre ótima, enquanto a Busca em Largura, embora sempre encontre a solução ótima, consome significativamente mais recursos em casos complexos. A Busca em Profundidade foi a menos eficiente, geralmente produzindo soluções de qualidade inferior com alto custo temporal. Este estudo contribui para a compreensão da aplicabilidade e limitações de diferentes estratégias de busca em problemas combinatórios, oferecendo insights para a escolha do algoritmo mais adequado em cenários semelhantes. .

1 Introdução

O problema do `8-puzzle` é um clássico desafio da inteligência artificial e da ciência da computação, amplamente utilizado como base para o estudo e desenvolvimento de algoritmos de busca em espaços de estados. Ele consiste em um tabuleiro de 3×3 contendo oito peças numeradas de 1 a 8 e um espaço vazio representado pelo número 0. O objetivo é, a partir de uma configuração inicial arbitrária, mover as peças para atingir uma configuração final previamente definida, onde os números estejam organizados em ordem crescente com o espaço vazio na última posição.

Este quebra-cabeça foi inventado e popularizado pelo matemático americano Samuel Loyd no final do século XIX, e desde então tem sido extensivamente estudado no campo da teoria dos jogos e da inteligência artificial. Sua relevância persiste até hoje como um excelente caso de teste para algoritmos de busca devido à sua natureza combinatória e à clara definição de estados e operadores.

O espaço de estados deste problema é finito mas extenso - existem exatamente $9!/2 = 181.440$ configurações possíveis e alcançáveis. Esta característica torna o problema suficientemente complexo para desafiar algoritmos ineficientes, mas ainda tratável para análise e comparação. Ademais, nem todas as configurações iniciais são solucionáveis. É possível demonstrar matematicamente que apenas metade das 9! permutações possíveis podem ser resolvidas, dependendo da paridade da permutação inicial em relação à configuração objetivo.

As operações permitidas são limitadas ao movimento do espaço vazio nas quatro direções cardeais (cima, baixo, esquerda e direita), desde que estes movimentos não violem os limites do tabuleiro. Esta restrição estabelece uma estrutura de grafo no espaço de estados, onde cada configuração pode ter de dois a quatro sucessores imediatos, dependendo da posição do espaço vazio.

Neste contexto, técnicas de busca desempenham papel central, permitindo explorar o espaço de estados de maneira sistemática e, muitas vezes, guiada por heurísticas. A eficiência destas técnicas pode ser medida por diversos critérios, incluindo tempo de execução, consumo de memória e qualidade da solução encontrada (número mínimo de movimentos).

Neste trabalho, são implementados e analisados quatro algoritmos de busca distintos aplicados à resolução do 8-puzzle:

- **Busca em Largura (Breadth-First Search - BFS)**: estratégia não informada que explora os estados por níveis, garantindo a solução com o menor número de movimentos, mas com alto consumo de memória. A BFS utiliza uma estrutura de fila (FIFO - First In, First Out) para gerenciar a fronteira de exploração, expandindo todos os nós de um nível antes de passar ao próximo.
- **Busca em Profundidade com Limite (Depth-First Search - DFS)**: explora os estados indo o mais fundo possível antes de retroceder, oferecendo menor uso de memória, mas com riscos de não encontrar a solução ótima ou até mesmo não encontrar solução em tempo razoável. Este algoritmo utiliza uma estrutura de pilha (LIFO - Last In, First Out) para armazenar a fronteira, o que privilegia a exploração em profundidade.
- **Busca Gulosa (Greedy Best-First Search)**: estratégia informada que utiliza uma heurística (neste caso, a distância de Manhattan) para guiar a busca na direção do objetivo, podendo ser eficiente em tempo, mas sem garantia de solução ótima. Este algoritmo ordena a fronteira exclusivamente pelo valor da função heurística, ignorando o custo acumulado até o estado atual.
- **Busca A* (A-Star)**: algoritmo heurístico que combina o custo do caminho já percorrido com a estimativa do custo restante até o objetivo, oferecendo uma das melhores combinações entre tempo de execução e qualidade da solução. O A ordena a fronteira com base na soma f(n) = g(n) + h(n), onde g(n) é o custo real acumulado até o estado n e h(n) é a estimativa heurística do custo restante.

Este estudo visa não apenas implementar estes algoritmos, mas também compará-los empiricamente em instâncias variadas do problema, proporcionando uma análise crítica das vantagens e desvantagens de cada abordagem.

2 Descrição da Implementação e Configuração dos Testes

A implementação dos algoritmos foi realizada em Python, utilizando estruturas básicas da linguagem e bibliotecas auxiliares para medição de desempenho. O código foi dividido em dois arquivos principais: um para a modelagem do problema do *8-puzzle* e outro para os algoritmos de busca.

O problema foi representado por meio da classe `Estado`, que armazena a configuração atual do tabuleiro, o estado pai (anterior), o custo acumulado e a ação tomada. Já a classe `Problema` contém os métodos para geração de sucessores, verificação do estado objetivo e cálculo da heurística de Manhattan, utilizada nos algoritmos informados.

Cada algoritmo de busca foi implementado em uma função separada, mantendo uma interface comum para facilitar a execução e a comparação entre eles. As estruturas de dados utilizadas variam de acordo com a estratégia: filas (`deque`) para a busca em largura, pilhas para a busca em profundidade, e filas de prioridade (`heapq`) para os algoritmos informados.

As instâncias de teste foram fornecidas em um arquivo CSV contendo 10 configurações iniciais do quebra-cabeça. Cada linha representa uma instância com os 9 valores do tabuleiro (de 0 a 8), onde 0 representa o espaço vazio. Essas instâncias foram carregadas, convertidas em objetos da classe `Estado`, e utilizadas como entrada para os algoritmos.

Para cada execução, foram coletadas as seguintes métricas:

- **Tempo de execução**, medido com a biblioteca `time`.
- **Uso de memória**, estimado com a biblioteca `tracemalloc`, registrando o pico de alocação durante a execução.
- **Qualidade da solução**, medida pelo número de movimentos até o estado objetivo.

A automatização dos testes permitiu aplicar os algoritmos de forma padronizada a todas as instâncias. Algumas melhorias no código e ajustes de desempenho foram realizados com auxílio pontual de ferramentas de Inteligência Artificial para revisão e organização da lógica.

3 Resultados

A Tabela 1 apresenta os resultados obtidos pelos quatro algoritmos nas dez instâncias do problema 8-puzzle. As métricas analisadas foram: qualidade da solução (número total de movimentos até alcançar o estado objetivo), tempo de execução (em segundos, com 4 casas decimais) e uso máximo de memória (em megabytes).

Table 1: Comparação dos algoritmos de busca para o problema 8-puzzle

Instância	**BFS** Passos / Tempo(s) / Memória(MB)	**DFS** Passos / Tempo(s) / Memória(MB)	**Gulosa** Passos / Tempo(s) / Memória(MB)	**A*** Passos / Tempo(s) / Memória(MB)
1	2 / 0.0026 / 0.00	2 / 0.0007 / 0.00	2 / 0.0033 / 0.00	2 / 0.0011 / 0.00
2	4 / 0.0022 / 0.01	96 / 0.0685 / 0.15	4 / 0.0012 / 0.01	4 / 0.0016 / 0.01
3	6 / 0.0132 / 0.06	90 / 2.8640 / 6.29	6 / 0.0018 / 0.01	6 / 0.0024 / 0.01
4	8 / 0.0244 / 0.12	100 / 2.8629 / 6.28	8 / 0.0086 / 0.04	8 / 0.0032 / 0.01
5	6 / 0.0079 / 0.02	6 / 7.2574 / 14.96	6 / 0.0017 / 0.01	6 / 0.0017 / 0.01
6	6 / 0.0113 / 0.04	98 / 2.8678 / 6.30	6 / 0.0022 / 0.01	6 / 0.0028 / 0.01
7	8 / 0.0204 / 0.10	98 / 2.4406 / 5.65	8 / 0.0030 / 0.01	8 / 0.0027 / 0.01
8	3 / 0.0019 / 0.00	99 / 6.8468 / 14.60	3 / 0.0015 / 0.01	3 / 0.0015 / 0.00
9	13 / 0.1155 / 1.20	99 / 4.6536 / 12.22	13 / 0.0041 / 0.01	13 / 0.0050 / 0.02
10	13 / 0.1360 / 1.63	99 / 2.4499 / 5.80	39 / 0.0158 / 0.15	13 / 0.0157 / 0.07

3.1 Análise dos Dados por Algoritmo

Os dados apresentados na tabela acima mostram características importantes de cada algoritmo:

Busca em Largura (BFS):

- Sempre encontrou a solução ótima (menor número de passos)
- Tempo de execução aumentou consideravelmente para instâncias mais complexas (9 e 10)
- Uso de memória mostrou crescimento exponencial em relação à profundidade da solução
- Para a instância 10, utilizou 1.63MB de memória, cerca de 23 vezes mais que o A* para o mesmo problema

Busca em Profundidade (DFS):

- Foi o algoritmo mais inconsistente em termos de qualidade da solução
- Encontrou caminhos muito longos (90-100 passos) para problemas que poderiam ser resolvidos em menos de 10 movimentos
- Apresentou os maiores tempos de execução, chegando a 7.2574 segundos na instância 5
- Consumo de memória elevado devido à exploração de caminhos longos e ineficientes

Busca Gulosa:

- Foi o algoritmo mais rápido na maioria das instâncias
- Encontrou a solução ótima em 9 das 10 instâncias
- Na instância 10, produziu um caminho com 39 passos (3 vezes maior que o ótimo)
- Uso de memória foi geralmente baixo, com pequenas variações entre instâncias

A* (A-Star):

- Sempre encontrou a solução ótima, com o mesmo número de passos que o BFS
- Tempos de execução consistentemente baixos, próximos aos da Busca Gulosa
- Menor uso de memória entre os algoritmos para problemas complexos
- Mostrou o melhor equilíbrio entre tempo, memória e qualidade da solução

4 Discussão dos Resultados

A partir dos testes realizados em dez diferentes instâncias do problema 8-puzzle, foi possível observar diferenças significativas no desempenho dos algoritmos de busca implementados. A Tabela 1 resume os principais dados obtidos em relação ao tempo de execução, memória utilizada e à qualidade da solução (número de passos).

4.1 Comparação de Eficiência e Qualidade

O algoritmo A* demonstrou ser o mais eficiente em todos os critérios avaliados. Ele foi consistentemente rápido e encontrou soluções ótimas em todas as instâncias, com o menor número de passos. Isso confirma o esperado, já que o A* combina custo de caminho e estimativa heurística de forma balanceada. A garantia teórica de otimalidade do A* quando se utiliza uma heurística admissível foi confirmada empiricamente nos testes.

A eficiência do A* pode ser atribuída a dois fatores principais:

1. O uso da heurística de Manhattan, que fornece uma boa estimativa do custo restante
2. A consideração do custo acumulado (g), que evita a exploração de caminhos ineficientes

Em segundo lugar, a busca gulosa também teve bom desempenho em termos de tempo, sendo quase tão rápida quanto o A*. No entanto, por não considerar o custo real acumulado, ela encontrou soluções subótimas em alguns casos, embora próximas do ideal. O resultado na instância 10 é particularmente ilustrativo: a busca gulosa produziu um caminho com 39 passos, enquanto o caminho ótimo tinha apenas 13 movimentos. Este comportamento é consistente com a teoria, já que a busca gulosa toma decisões localmente ótimas, sem garantia de otimalidade global.

A busca em largura, por sua vez, garantiu sempre soluções ótimas, mas com maior tempo de execução comparado ao A* e à busca gulosa, especialmente nas instâncias mais complexas (9 e 10). Isso se deve ao seu alto consumo de memória e ao crescimento exponencial da fronteira durante a exploração. A BFS explora exaustivamente todos os estados em um nível antes de passar ao próximo, o que garante a solução ótima, mas com alto custo computacional.

Por fim, a busca em profundidade se mostrou a estratégia menos eficiente. Apesar de sua simplicidade, ela teve os maiores tempos de execução e, em diversas instâncias, encontrou caminhos com número elevado de passos ou explorou caminhos longos e desnecessários, devido à sua natureza de exploração até o fundo antes de retroceder. É importante notar que, mesmo com a implementação de limite de profundidade e verificação de estados repetidos, o DFS ainda apresentou resultados significativamente inferiores aos demais algoritmos.

4.2 Análise do Consumo de Memória

O consumo de memória foi outro aspecto crítico observado nos testes. Os dados mostram que:

- A busca em largura teve consumo de memória crescente com a complexidade do problema, chegando a 1.63MB na instância 10. Isto ocorre porque o BFS mantém na fronteira todos os nós de um nível antes de explorá-los.
- A busca em profundidade, contrariando a expectativa teórica de baixo consumo de memória, utilizou quantidades significativas de memória em várias instâncias. Isto pode ser explicado pela implementação que armazena estados explorados para evitar ciclos, além da exploração de caminhos muito longos.
- Os algoritmos informados (Gulosa e A*) tiveram um consumo de memória significativamente menor, especialmente o A*, que conseguiu direcionar a busca de forma mais eficiente, reduzindo o número de estados expandidos e armazenados.

4.3 Implicações Práticas

Esses resultados evidenciam a importância de utilizar algoritmos informados, como o A*, quando se dispõe de uma boa heurística, como a distância de Manhattan, que é admissível e consistente

no contexto do 8-puzzle. Para problemas de espaço de estados com características semelhantes ao 8-puzzle, o A* representa a melhor escolha em termos de equilíbrio entre qualidade da solução e eficiência computacional.

A busca gulosa pode ser uma alternativa viável quando o tempo de execução é mais crítico que a garantia de otimalidade, enquanto a busca em largura é recomendada apenas para problemas de pequena dimensão ou quando a garantia de otimalidade é essencial e não se dispõe de uma heurística adequada.

A busca em profundidade, pelo menos na implementação testada, mostrou-se inadequada para o problema do 8-puzzle, sugerindo que algoritmos não informados de exploração vertical têm utilidade limitada em problemas combinatórios com espaços de estados amplos.

5 Conclusão

Neste trabalho, foram implementados e comparados quatro algoritmos de busca aplicados ao problema clássico do 8-puzzle: busca em largura (BFS), busca em profundidade (DFS), busca gulosa e A*. Por meio de testes em múltiplas instâncias, foi possível avaliar o desempenho de cada estratégia com base em três critérios principais: tempo de execução, uso de memória e qualidade da solução.

5.1 Síntese dos Resultados

Os resultados mostraram que o algoritmo A* foi o mais eficiente e robusto, oferecendo um ótimo equilíbrio entre tempo de execução e qualidade das soluções. A busca gulosa também apresentou um desempenho competitivo, sendo bastante rápida, embora menos precisa. A busca em largura, apesar de garantir soluções ótimas, foi menos eficiente em termos computacionais, enquanto a busca em profundidade obteve os piores resultados, tanto em tempo quanto em qualidade das soluções.

5.2 Limitações do Estudo

É importante reconhecer algumas limitações deste trabalho:

- O número limitado de instâncias de teste (dez) pode não representar completamente o espectro de dificuldade do problema.
- As medições de memória utilizadas, embora suficientes para comparação, não capturam completamente o uso real de recursos do sistema.
- O estudo focou apenas em um tipo de heurística (distância de Manhattan) para os algoritmos informados.

5.3 Melhorias e Trabalhos Futuros

Como melhorias futuras, seria interessante:

- **Explorar diferentes heurísticas no A***, como a contagem de peças fora do lugar ou a heurística de Nilsson (soma das distâncias de Manhattan mais uma penalidade para configurações não lineares), para comparar seus impactos no desempenho.
- **Implementar variações dos algoritmos**, como Busca em Profundidade Iterativa (IDFS), que combina vantagens da BFS e DFS, ou variantes do A* como o IDA* (Iterative Deepening A*) para problemas maiores.
- **Analisar o uso real de memória por cada algoritmo**, por meio de ferramentas de medição mais precisas e detalhadas, como profilers específicos para Python que possam detalhar a alocação de memória por estrutura de dados.
- **Expandir os testes para o problema do 15-puzzle** (tabuleiro 4x4), avaliando a escalabilidade das estratégias implementadas em um espaço de estados significativamente maior (aproximadamente 10^{13} estados).

- **Estudar o uso de técnicas de otimização e poda para reduzir o espaço de busca** em algoritmos não informados, como detecção de estados transponíveis ou simetrias no quebra-cabeça.
- **Paralelizar os algoritmos** para explorar como técnicas de computação paralela poderiam melhorar o desempenho, especialmente em algoritmos como o BFS e o A* que se beneficiariam da expansão simultânea de múltiplos caminhos.
- **Aplicar os algoritmos estudados a outros problemas de busca em espaço de estados**, como o problema das N-rainhas ou jogos como Sokoban, para avaliar a generalidade das conclusões obtidas.

5.4 Considerações Finais

A comparação evidenciou o valor do uso de algoritmos informados e bem projetados, especialmente em problemas de natureza combinatória como o 8-puzzle, onde a eficiência na exploração do espaço de estados é crucial. O estudo reforça a importância do equilíbrio entre completude, otimalidade, tempo de execução e uso de memória na escolha de algoritmos de busca para problemas específicos.

A implementação e análise realizadas neste trabalho contribuem não apenas para a compreensão prática dos algoritmos de busca, mas também para a consolidação de conhecimentos teóricos sobre complexidade computacional e estratégias de resolução de problemas em inteligência artificial.