Trabalho Prático 1

Algoritmos de Busca para 8-Puzzle Problem

Brisa do Nascimento Barbosa
Universidade Federal de Minas Gerais (UFMG)

1. Introdução

No presente trabalho, foram implementados seis algoritmos de busca para encontrar a sequência de movimentos ideal para a solução de um 8-Puzzle. Esse *toy problem* consiste de um quebra-cabeça deslizante que tem um tabuleiro com 3x3 espaços com 8 peças numeradas, além de um espaço em branco. O objetivo do jogo é reorganizar as peças do quebra-cabeça, movendo-as uma de cada vez para o espaço em branco, até que todas as peças estejam em ordem numérica. A seguir, detalharemos cada algoritmo: Breadth-First Search, Iterative Deepening Search, Uniform-Cost Search, A* Search, Greedy Best-First Search e Hill Climbing.

2. Modelagem

O programa foi modelado de acordo com as estrutura principais da Classe Nodo, que armazena a árvore de possibilidades, e a Classe Puzzle, que encapsula as características do quebra-cabeça, e, por fim, um folder com os algoritmos.

2.1 Node

A classe Node é usada para representar estados no espaço de busca desse quebra-cabeça, assim, cada nodo contém informações da configuração atual das peças: o nodo pai, a ação que levou a este estado e o custo associado. Para expandir as buscas, cada nó possui método de **`generate_children`**, que gera os possíveis estados futuros a partir do próprio estado por meio de movimentos permitidos (mover para direita, esquerda, cima ou baixo) de acordo com o espaço vazio e suas peças adjacentes. Ademais, para validar movimentos, o método **`is_valid_move`** garante que a peça movida não saia dos limites do tabuleiro, e o método **`swap`** simula a aplicação de uma ação trocando peças de lugar. Por fim, **`get_solution_path`** reconstrói o caminho da solução a partir do nó atual até o nó raiz para encontrar a sequência de ações utilizadas para resolver o jogo.

2.1 Puzzle

A classe EightPuzzle representa o tabuleiro em si, ou seja, o seu estado inicial, fornecido na inicialização, e o estado objetivo, que é o tabuleiro numerado de 1 até 8, com o 0 no final. Além disso, também contém métodos para imprimir a solução, bem como a sequência de estados executados para chegar até ela.

3. Algoritmos

Apresentamos, agora, uma descrição sucinta de cada algoritmo e como foi utilizado para resolver o 8-Puzzle.

3.1. Busca sem Informação

A busca sem informação envolve a exploração sistemática do espaço de estados possíveis do quebra-cabeça sem usar qualquer conhecimento específico sobre o objetivo ou o ambiente. Isso significa que o algoritmo não utiliza informações heurísticas para orientar a busca em direção à solução.

3.1.1. Breadth First Search

O BFS inicia a busca a partir do estado inicial dado e explora todos os possíveis estados alcançáveis através de movimentos válidos (ou seja, expande em largura). É utilizada uma fila para armazenar os estados a serem explorados, garantindo que todos os estados em um nível sejam visitados antes de passar para o próximo nível. Em seguida, expande os estados vizinhos sucessivamente até encontrar o estado objetivo.

3.1.2. Iterative Deepening Search

O IDS combina os princípios da busca em largura (BFS) e da busca em profundidade (DFS) ao realizar buscas sucessivas em profundidades crescentes a partir do estado inicial, permitindo a exploração sistemática de todos os estados alcançáveis, evitando o consumo excessivo de memória. Para isso, o código implementa função de busca em profundidade limitada para explorar estados até atingir a profundidade limite ou encontrar o estado objetivo. A cada iteração, o código aumenta a profundidade limite e continua a busca até encontrar uma solução ou esgotar todas as profundidades possíveis.

3.1.3. Uniform-Cost Search

O UCS, também conhecido como Dijkstra, começa a partir do estado inicial e explora os estados alcançáveis através de movimentos válidos, mas em vez de expandir em largura como o BFS, ele prioriza os estados com menor custo. Para tal, é utilizada uma fila prioritária para garantir que os estados com menor custo sejam visitados primeiro. Ele expande sucessivamente os estados vizinhos com os menores custos até encontrar o estado objetivo.

3.2. Busca com Informação

A busca com informação utiliza uma função heurística que estima o custo ou a distância até o objetivo a partir de um estado atual, permitindo priorizar estados mais promissores. Assim, avaliamos as peças do quebra-cabeça dando preferência às ações que levam a um estado que parece mais próximo da solução. Essa técnica melhora significativamente a eficiência da busca.

Com isso, utilizamos as seguintes heurísticas:

Distância de Manhattan: Calcula a estimativa de custo para atingir o estado objetivo contando o número de movimentos horizontais e verticais necessários para posicionar cada peça no estado atual até suas posições corretas no estado objetivo. Essa heurística é admissível para o 8-puzzle, o que significa que ela nunca superestima o custo para atingir o objetivo real, já que a movimentação de uma peça em direção ao seu destino desejado não pode ser feita em menos etapas do que a distância de Manhattan.

Número de mismatches: Conta o número de peças que estão na posição errada em relação ao estado objetivo e assume que cada uma dessas peças requer um movimento para ser colocada no lugar certo. Essa heurística é admissível, pois nunca superestima o custo para alcançar a solução, uma vez que é impossível mover uma peça para a posição correta sem pelo menos um movimento.

3.2.1. A* Star Search

O A* combina a avaliação do custo atual do caminho percorrido com uma heurística que estima o custo restante até o objetivo. Começando no estado inicial, o A* mantém uma fila de prioridade, onde os estados a serem explorados são priorizados com base na soma do custo atual e da heurística. Ele seleciona estados da fila, os marca como visitados e gera estados vizinhos, calculando os custos e adicionando-os à fila. A busca continua até encontrar o estado. Utilizamos a heurística Distância de Manhattan.

3.2.2. Greedy Best-Find Search

O GBFS prioriza estados que parecem estar mais próximos do objetivo com base na heurística utilizada. Ele não leva em consideração o custo real para alcançar um estado, apenas a estimativa heurística. Isso significa que o GBF pode encontrar soluções rápidas em alguns casos, mas não garante a solução ótima, pois pode ser influenciado por escolhas de estados com base apenas na heurística, sem considerar o custo real para alcançar o objetivo. Utilizamos a heurística de Mismatch.

3.3. Busca Local

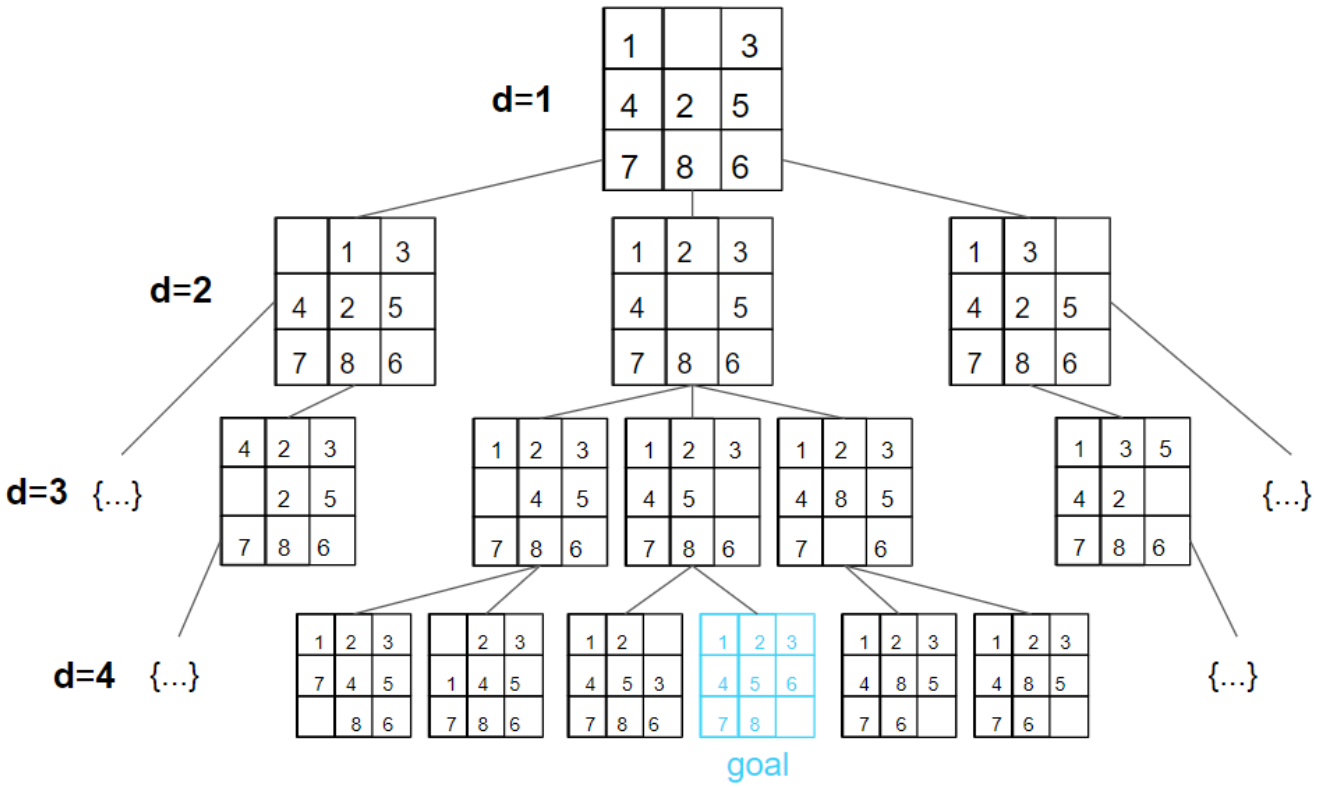
Na busca local, o algoritmo começa a partir de um ponto no espaço de soluções e explora o entorno desse ponto, tomando decisões que se baseiam apenas nas informações disponíveis localmente, sem um conhecimento global do espaço de estados.

3.3.1. Hill Climbing

O Hill Climbing começa com um estado inicial e, iterativamente, gera estados vizinhos por meio de movimentos válidos, selecionando o estado vizinho com a menor heurística (número de peças fora do lugar em relação ao estado objetivo). Ele compara essa heurística com a do estado atual e, se o vizinho for melhor, atualiza o estado atual e repete o processo por um número limitado de iterações. No final, verifica se alcançou o estado objetivo. No caso do algoritmo implementado, ele limita a quantidade de movimentos laterais na esperança de evitar de ficar preso.

4. Execução

Para testar a efetividade dos algoritmos, foram utilizados os exemplos fornecidos por puzzle.pdf. A seguir, observe a árvore de possibilidades da entrada inicial “1 5 2 4 0 3 7 8 6” como exemplo.



Ao conduzir a execução dos algoritmos com essa entrada e avaliar seu desempenho usando a biblioteca 'timeit', foi possível determinar com precisão os tempos de execução de cada algoritmo. Nesse contexto, o algoritmo Greedy Best-First Search (GBF) se destacou como o mais rápido entre todos os algoritmos testados. Notavelmente, o Breadth-First Search (BFS), que, nesse caso, expande apenas um número pequeno de estados, também se mostrou rápido neste caso em específico.

Contudo, é importante mencionar que o Hill Climbing (HC), embora tenha encontrado uma solução para esse problema pequeno em específico, não foi um método eficaz nem exato, falhando em encontrar a solução na maioria dos casos posteriores. Portanto, foi excluído das análises apresentadas a seguir, já que não contribuiria na comparação dos algoritmos significativamente melhores que ele.

Na seguinte tabela, comparamos os algoritmos em relação à profundidade máxima (esperada pela solução ótima) e o tempo de execução.

Tabela 1. Comparação de tempo em relação à profundidade esperada

	BFS	IDS	UDC	A*	GBF
7	0,000830	0,001027	0,001536	0,000221	0,000451
11	0,006808	0,007340	0,012375	0,000347	0,006912
15	0,035230	0,035936	0,106566	0,001040	0,007238
20	0,343865	0,130622	0,696449	0,002322	0,014981
23	3,057053	0,374668	5,685127	0,013872	0,012363

Inicialmente, observamos uma tendência clara de aumento no tempo de execução à medida que a profundidade da busca aumenta, o que está de acordo com a complexidade polinomial desses algoritmos, pois a busca explora estados de maneira crescente em uma árvore de possibilidades. Além disso, observamos que, dentre os algoritmos de busca sem informação, o BFS e o UDC ainda são equiparáveis com o IDS para casos pequenos, de até 15 de profundidade esperada. No entanto, percebe-se uma mudança significativa no desempenho do BFS e UDC ao atingir profundidades maiores que 20, uma vez que esses algoritmos são de busca completa, logo, gastam consideravelmente mais tempo com o aumento da profundidade. Nesse âmbito, IDC se destaca como o mais rápido nos de busca sem informação, já que controla a profundidade de busca.

Assim, executada mais uma rodada, dessa vez comparando a quantidade passos, ou seja, a profundidade ‘d’ que caminharam na árvore de estados, e o tempo de execução ‘t’ em segundos, temos a seguinte tabela para a entrada inicial sendo “8 4 7 6 2 3 5 1 0”.

8	4	7
6	2	3
5	1	

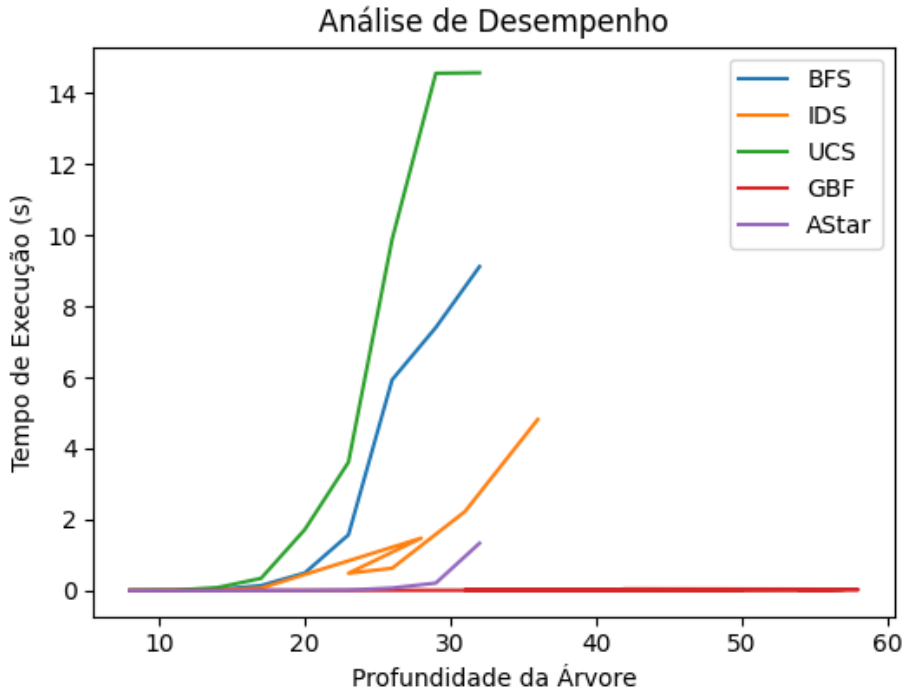
Tabela 2. Comparação de Tempo e Profundidade

	BFS	IDS	UCS	A*	GBF
t	3,362064	1,859579	6,909115	0,863770	0,020684
d	28	30	28	28	56

Desse modo, observou-se que o IDS de fato é o mais rápido da busca sem informação para maior profundidade, enquanto o UCS se provou o mais lento, devido à sua expansão ingênua. Isso porque o UCS expande os nós em ordem crescente de custo acumulado, priorizando os caminhos mais baratos, de forma que ele pode expandir caminhos que têm custos muito altos, se forem melhores no momento.

Por outro lado, dentro dos algoritmos com informação, ambos A* e GBF são expressivamente mais rápidos em comparação com os sem informação, principalmente devido à sua abordagem heurística e estratégia de seleção de nós. Nesse sentido, a heurística informada permite que os algoritmos direcionem seus esforços para os caminhos mais promissores, economizando tempo ao evitar a exploração de caminhos que parecem menos propensos a levar à solução. De fato, esses dois algoritmos executaram todos os casos menos de um segundo, sendo o GBF mais veloz, apesar de explorar mais estados. Isso porque, em vez de explorar todas as possibilidades igualmente, como o A*, o GBF segue um caminho que parece levar mais diretamente à solução. O GBF está disposto a sacrificar a exatidão em favor da eficiência, enquanto, o A* busca uma solução ótima.

Assim, para visualizar tais diferenças de eficiência, exibimos esse gráfico criado com a biblioteca ‘matplotlib’. Omitimos mais uma vez o Hill Climbing, que muito rapidamente concluiu para cada caso que não havia uma solução disponível.



Agora, comparando heurísticas, apresentamos os seguintes resultados utilizando alguns casos fornecidos.
h_1: número de peças fora do lugar
h_2: distância de manhattan

	A*(h_1)	**A*(h_2)**	**GBF(h_1)**	**GBF(h_2)**
8 0 7 5 4 2 1 6 3	0,046220	0,003413	0,009417	0,012178
8 4 7 5 6 2 1 3 0	1,756488	0,093824	0,011629	0,011426
8 6 7 2 5 4 3 0 1	11,64157	0,903762	0,008629	0,003617

A partir da análise da tabela, evidencia-se que a heurística desempenha um papel essencial na determinação da eficiência dos algoritmos de busca. Particularmente, a heurística de Manhattan se destaca como a mais eficaz, e quando empregada no GBF, resulta na abordagem mais rápida na busca de solução do 8-puzzle.

5. Conclusão

A partir do desenvolvimento do programa, pode-se compreender as diferenças de implementação e eficácia de seis algoritmos de busca. Em suma, dentre os busca sem informação, representado pelo Breadth-First Search (BFS), Iterative Deepening Search (IDS), e Uniform-Cost Search (UCS), revelou-se eficiente em cenários de baixa profundidade, mas se tornou mais lento medida que a profundidade da busca aumentava. Dentre eles, o IDS se mostrou o mais veloz, enquanto o UDS foi o mais lento.

No contexto dos algoritmos de busca com informação, sendo eles o A* e o Greedy Best-First Search (GBF), ficou evidente que eles são os mais eficazes em nosso estudo. O A* prioriza a busca de uma solução ótima, enquanto o GBF busca soluções de forma mais rápida, priorizando a eficiência em detrimento da otimalidade. É importante destacar que a escolha da heurística exerceu um impacto significativo no desempenho, sendo a heurística de Manhattan aquela que se sobressaiu como a mais eficaz.

Por fim, a busca com local, representada pelo Hill Climbing (HC), demonstrou ser inadequada para a resolução do 8-Puzzle, já que não é um algoritmo que garante encontrar soluções consistentes. Em particular, sequer encontrou solução na maioria dos casos executados.