Trabalho Prático 1 - Algoritmos de busca aplicados ao 8-puzzle

Raissa Miranda Maciel
14 de Outubro de 2023
Matrícula: 2020006965

1. Introdução

O 8-puzzle é um quebra cabeça deslizante que envolve um tabuleiro 3x3 com oito peças numeradas e um espaço vazio. O objetivo do jogo é rearranjar as peças a partir de uma configuração inicial para uma configuração final ordenada, movendo uma peça por vez para o espaço vazio.
Este problema pode ser solucionado através de algoritmos de busca. como a Busca em Largura (BFS), Busca em Profundidade (DFS), A* entre outros.
Esta documentação mostra uma visão geral do problema 8-puzzle, descreve e analisa os algoritmos usados para resolução, assim como suas vantagens e desvantagens.

2. Modelagem dos componentes

Esta seção mostrará como a árvore de busca foi criada e as estruturas de dados utilizadas e explicará os componentes de busca escolhidos para encontrar a solução.

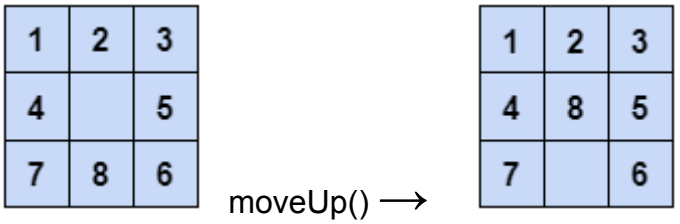
2.1 Classe utilizada

A classe Node, mostrada abaixo, é a representação de um nó na árvore de busca criada para solucionar o problema. Ela simboliza um estado no espaço de busca.

Class Node	
currentBoard	Matrix 3x3 que armazena a configuração atual do tabuleiro. Cada elemento representa uma peça.
parent	Mantém referência ao nó pai no espaço de busca. Útil para rastrear a sequência de movimentos até determinada configuração.
children	Lista que armazena os nós filhos deste nó. Eles representam as configurações possíveis de serem alcançadas a partir dele por um único movimento.
depth	Indica a profundidade deste nó na árvore de busca
cost	Representa o custo para chegar a este nó. Útil em determinadas estratégias.
emptySpace	Guarda as coordenadas do espaço vazio

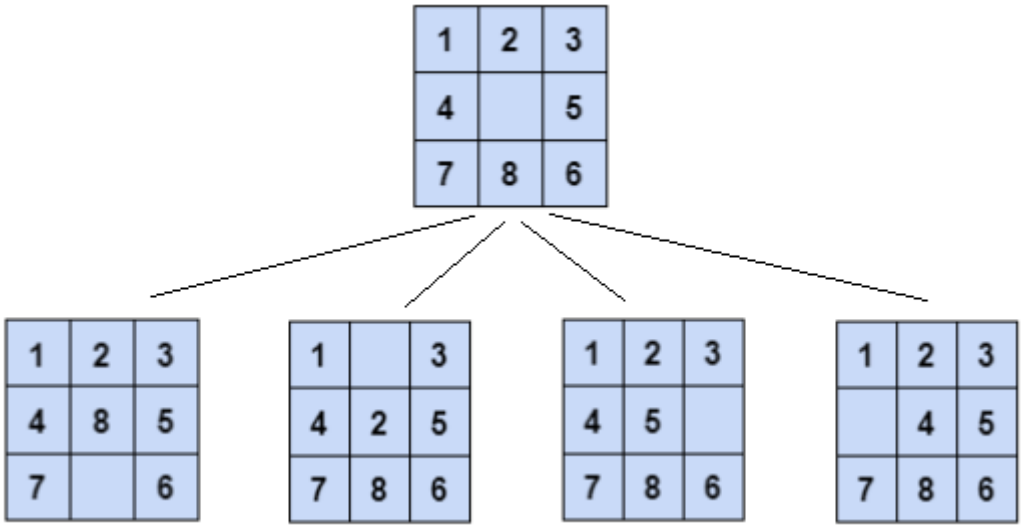
2.2 Componentes de busca

As funções que representam os movimentos no tabuleiro são responsáveis pela navegação e exploração do espaço de estados do 8-puzzle. Ao aplicar essas ações de movimento, é criada a árvore de busca que será explorada pelos algoritmos. Os movimentos podem ser para cima, para baixo, para direita e para esquerda. Eles fazem referência à peça que está se movendo, não ao espaço vazio. Um exemplo é mostrado a seguir:



Além da função *moveUp()*, as funções *moveDown(), moveLeft()* e *moveRight()* são responsáveis pelos outros movimentos.

Durante a busca, a função *expand()* expande o nó atual, isto significa que todas as configurações válidas possíveis de serem alcançadas através de um único movimento serão listadas. Essas configurações são aquelas geradas pelas funções mencionadas anteriormente. A lista de expansão gerada é armazenada no membro *children* deste nó. Um exemplo de expansão é mostrado a seguir:



3. Descrição dos algoritmos

Diversas estratégias podem ser utilizadas para a resolução do 8-puzzle. A seguir são mostradas as vantagens e desvantagens dos algoritmos implementados.

3.1 Busca em Largura - BFS

Este algoritmo explora todos os estados em camadas, o que significa que considera todos os possíveis movimentos de um estado antes de passar para o próximo nível. Iniciando do nó raiz, uma fila FIFO é utilizada e, enquanto ela não estiver vazia, o nó

é retirado, verificado se é a solução e em seguida explorado, gerando seus filhos. Caso seja a solução, este nó é retornado.O processo continua até que a fila esteja vazia, o que significa que não há caminho para a solução.
É um algoritmo completo, uma vez que encontrará a solução se houver caminho até ela. A solução encontrada é garantidamente ótima, ou seja, com o menor número de movimentos. Entretanto, para situações em que a solução se encontra em um estado muito distante da raiz, a ampla busca explora um número muito grande de nós (em ordem exponencial), o que pode mostrar ineficiência em relação ao tempo e memória gastos.

3.2 Dijkstra - UCS

Este algoritmo possui uma fila de prioridade para a escolha do nó que será explorado em relação ao menor custo. O custo associado a esse problema é a profundidade dos nós visitados que ainda não foram explorados, o que faz parecer a busca em largura. Iniciando a partir do nó raiz, o seu filho escolhido para exploração é aquele de menor custo, e assim sucessivamente, até que a solução seja encontrada ou percorra todos os caminhos possíveis sem sucesso. Este algoritmo expandirá todos os estados com um custo menor que o da solução, realizando uma vasta exploração ao seu redor com raio igual ao custo da solução. É possível fazer buscas mais inteligentes e direcionadas, considerando o caminho percorrido e outras informações para encontrar a melhor direção. Essas estratégias serão vistas nos próximos algoritmos. A complexidade de tempo e espaço podem variar dependendo da configuração inicial do quebra-cabeça, mas continua exponencial, uma vez que o custo é a profundidade do nó.

3.3 Busca Interativa Profunda - IDS

Este algoritmo combina a estratégia da busca em profundidade e em largura, de forma interativa. Em vez de dar uma profundidade fixa para a busca em profundidade, o IDS começa com a profundidade limite 0, realiza a exploração desses estados como uma BFS, e caso a solução não tenha sido encontrada, aumenta o limite gradualmente até que toda a árvore seja percorrida. É um algoritmo completo e ótimo, pois encontra o menor caminho para a solução, se ele existir e sua principal vantagem é o menor uso de memória pois, como é realizado de forma iterativa, não precisa manter todos os nós da busca anterior em memória. Em casos complexos, o tempo gasto é exponencial, pois todo o espaço de busca será explorado, mas sua memória gasta é da ordem da profundidade máxima da busca.

3.4 Algoritmo Guloso - GS

Este algoritmo é uma estratégia de busca que faz a escolha de expansão com base em critérios heurísticos, priorizando a expansão de nós que parecem mais promissores em relação à solução. As heurísticas usadas neste programa são explicadas na seção 4. Ele é eficiente para resolução de problemas em que a heurística é uma boa estimativa da distância real para a solução, mas em casos em

que a escolha não for adequada pode não encontrar solução (seguiu uma direção que foi erroneamente considerada promissora) e também não garante a otimalidade pois pode ficar preso em mínimos locais, levando a soluções subótimas.

3.5 A Estrela - A*

Este algoritmo realiza uma busca informada que combina os aspectos de Dijkstra com a estratégia gulosa. Ele utiliza uma heurística para estimar o custo total da solução a partir do nó atual, assim como considera o custo anterior até chegar no nó atual. Ele é representado pela função f(n) = g(n) + h(n), onde g(n) é o custo atual da raiz até o nó n (a profundidade) e h(n) é a estimativa de custo restante do nó até a solução (a heurística). Dessa forma, é mantida uma fila ordenada de acordo com essa solução, e o primeiro é retirado até que a solução seja encontrada ou toda a árvore tenha sido percorrida. A solução é garantidamente completa e ótima se a heurística escolhida for admissível, ou seja, nunca superestima o custo real para solução. Para boas escolhas, ele tende a ser eficiente, mas caso contrário, pode ter caráter temporal e espacial exponencial.

3.6 Hill Climbing

Este algoritmo é uma técnica de busca local usada para encontrar a sequência de movimentos para solução ótima. Ela possui uma abordagem gulosa mas que sempre vai parar em mínimos locais. A implementação para esse problema, permite movimentos laterais, isto é, continua explorando nós por um número específico de iterações, com a esperança de que saia deste mínimo local. Dessa forma, quando há a expansão de um nó, os vizinhos (filhos) são verificados em relação a uma heurística. Caso o vizinho possua um custo menor que o estado atual, ele é escolhido como próximo a ser expandido. Se o custo for o mesmo, ele dará a oportunidade de realizar esse movimento lateral um número específico de vezes, esta é a tentativa de sair de um mínimo local. Se os vizinhos possuírem apenas custos maiores, significa que já encontraram o melhor possível e esta é a solução. Entretanto, essa solução, retornada tanto quando o número de movimentos laterais foi atingido ou quando não há mais vizinhos promissores, pode não ser a solução final desejada.

4. Especificação das heurísticas

Duas heurísticas diferentes foram utilizadas para a resolução do problema, elas são discutidas a seguir:

4.1 Número de peças na posição errada

Esta é uma função de avaliação usada para estimar a distância de um estado atual até o estado objetivo. É uma heurística admissível porque nunca superestima o custo real para encontrar a solução. Uma vez que são contados o número de peças que se encontram fora do lugar, a contagem nunca será maior do que a quantidade real de movimentos necessários para alcançar a solução. Também é uma heurística

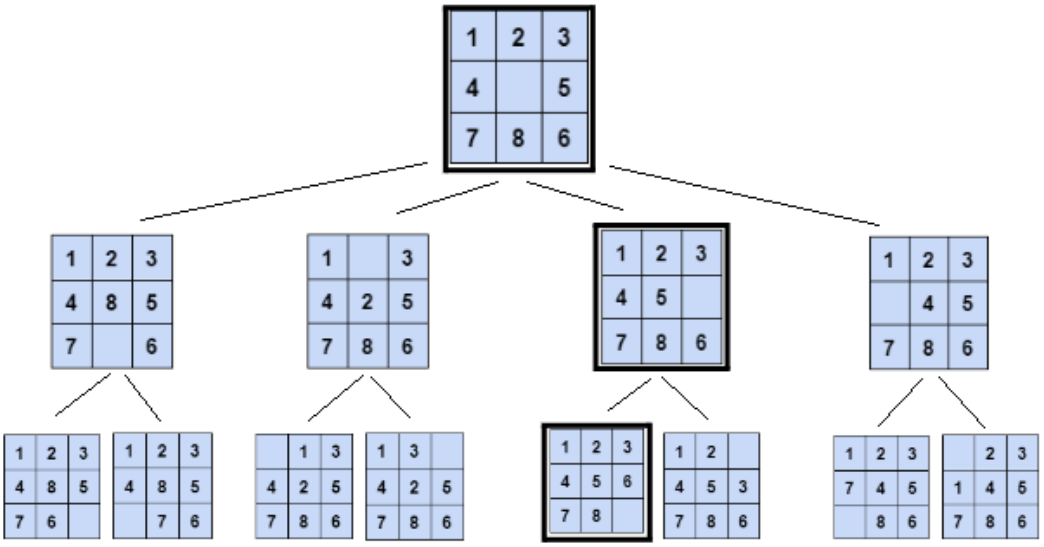
consistente pois, para cada estado, a estimativa da heurística para qualquer estado sucessor somada do custo para alcançar esse estado sucessor sempre será menor ou igual a estimativa da heurística para o estado atual. Isso significa que a heurística é uma medida de progresso que,não apenas é admissível, mas também se comporta de forma consistente. Em resumo, à medida que estados sucessores são alcançados, a contagem de peças fora de lugar não diminuirá.

4.1 Distância até a posição correta

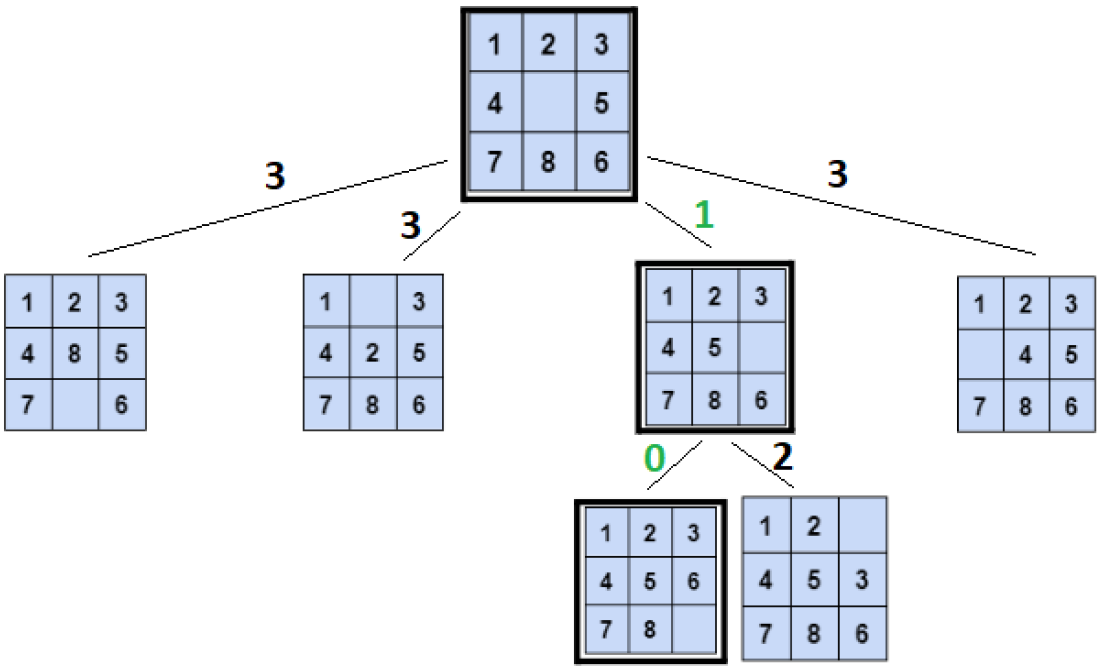
Esta heurística se baseia na ideia de medir o quão longe cada peça do quebra-cabeça está da sua posição correta, visando o estado ordenado. Ela é considerada admissível porque, para mover uma peça para a posição correta, a distância real percorrida não pode ser maior do que a distância que a heurística mede. Ela também é consistente porque, à medida que são alcançados os estados sucessores, a distância das peças até a posição correta diminuirá, garantindo progresso até a solução.

5. Exemplos de soluções

Um exemplo simples para um estado inicial não muito complexo, retornado pelo programa, utilizando o algoritmo BFS é mostrado a seguir. Note-se que estados já expandidos não são expandidos novamente, e em alguns casos em que o espaço vazio está na borda do tabuleiro, podem existir ações inválidas.

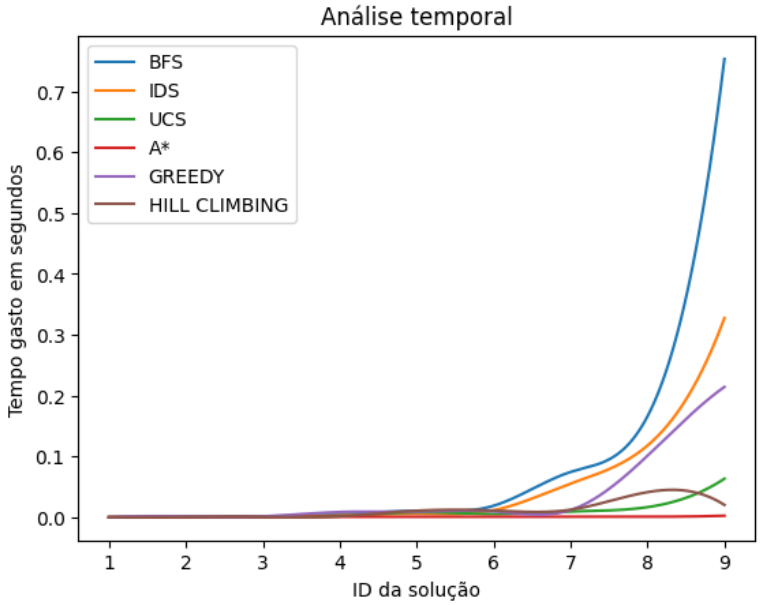


Podemos ver também um exemplo para o algoritmo informado *Greedy* que utiliza a heurística de distância das peças até a posição correta. Dessa forma, as arestas possuem com o menor custo são escolhidas para serem expandidas.



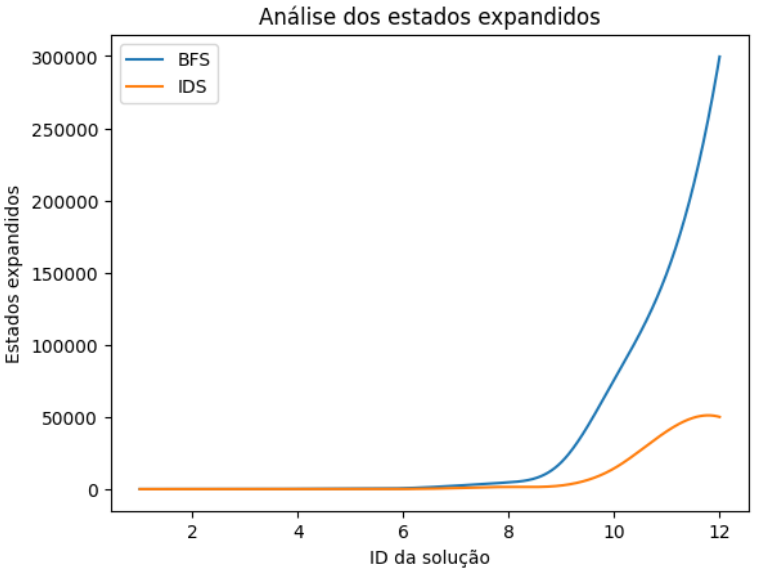
6. Análise quantitativa

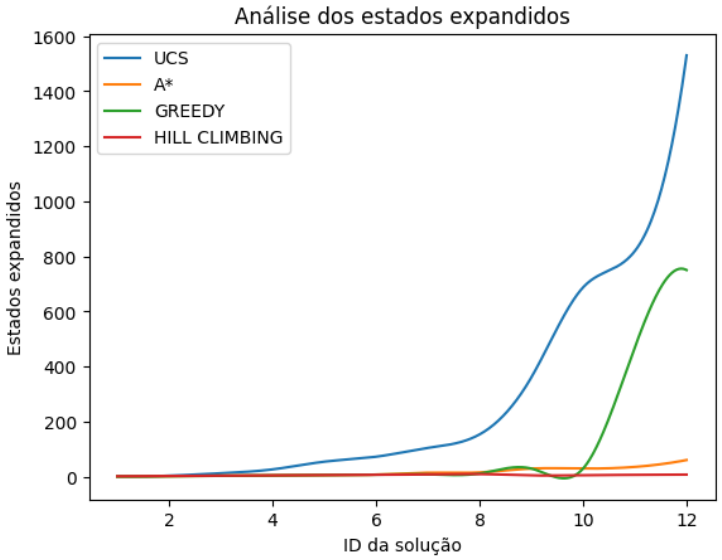
Foram coletados os tempos para 10 estados iniciais de tabuleiro e o número de estados expandidos até que a solução tenha sido encontrada. Com o auxílio da biblioteca *matplotlib* do python, foram plotados os gráficos abaixo e comparados os resultados para cada algoritmo. A enumeração das soluções segue a mesma enumeração de casos fornecida durante a especificação do trabalho. Para a comparação do número de estados expandidos, foi realizada a BFS e IDS, que possuem ordem de número semelhantes e muito discrepantes dos demais, separadamente.



A análise temporal aponta o algoritmo BFS e IDS como sendo os com maior custo de execução e contagem de estados expandidos. Nesse experimento, os valores de BFS são maiores pois, apesar de o IDS repetir a exploração a cada limite incrementado, o maior número de nós se encontra

nas camadas mais profundas, onde não foram necessárias serem visitadas. No geral, esses dois algoritmos realizam uma ampla busca extensiva, sem considerar nenhuma ideia promissora para encontrar a solução. O algoritmo de Dijkstra começa a observar o custo do caminho até chegar no nó atual e, por isso, diminui consideravelmente o tempo e o número de estados expandidos.





O algoritmo Greedy, já considera uma estratégia gulosa considerando a heurística escolhida, mostrando seu comportamento mais vantajoso em relação aos anteriores, tanto no tempo quanto no número de estados expandidos. Entretanto, como explicado anteriormente, ele não garante que encontrará uma solução, nem otimalidade, podendo não ser tão interessante em determinados casos. Já o A*, combina as ideias de Dijkstra e Greedy, limitando bastante o número de estados expandidos e, com uma boa heurística escolhida, consegue encontrar os movimentos em direção a solução de forma muito mais eficiente. Como as heurísticas utilizadas são admissíveis e consistentes, esse comportamento é mostrado nos gráficos com um menor número de estados e menor tempo comparado aos demais. Entretanto, se a heurística não fosse ideal, a complexidade do problema aumentaria indefinidamente, percorrendo caminhos na direção errada antes de passar pelo caminho promissor. Por fim, o Hill Climbing se mostra bastante vantajoso em relação às duas métricas utilizadas. Isso se deve pelo fato de ser permitido que ele pare em mínimos locais e não encontre a solução especificada. Nos experimentos realizados, a solução não foi encontrada em alguns testes, isso explica o número de

espaços expandidos diminuir enquanto a complexidade do problema aumenta, representando um caso de mínimo local.

7. Conclusão

A escolha dos algoritmos e a implementação utilizada para a busca aplicada ao problema 8-puzzle são fundamentais para a eficiência e eficácia da resolução do quebra-cabeça. Cada algoritmo possui vantagens e desvantagens específicas discutidas ao longo do trabalho e a escolha do algoritmo certo depende de características como o estado inicial, a heurística utilizada e os recursos computacionais disponíveis. Enquanto algoritmos não informados, como a BFS, IDS e UCS são mais simples e mais lentos, os algoritmos informados fazem uma busca mais inteligente, mas dependem da escolha de uma boa heurística. Realizar a implementação de cada um deles permitiu que fosse explorado e percebido na prática as vantagens e desvantagens de cada um. Também foi notado durante a realização dos testes, a importância de realizar uma boa organização do problema, construindo a representação da classe e a escolha das ações realizadas dentro dos algoritmos. Caso sejam realizadas ações muito custosas para objetivos simples dentro do algoritmo, por exemplo, a escolha do *contêiner* correto para alocar nós visitados, explorados e a fila de prioridade, o algoritmo pode apresentar um desempenho assintótico diferente do esperado, perdendo as suas vantagens em relação aos demais.