Henrique Assumpção
Matrícula: 2020006620
henriquesoares@dcc.ufmg.br

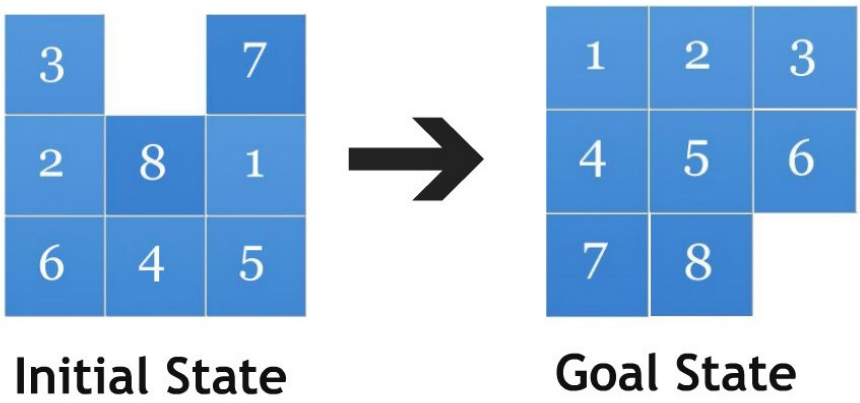
TP1: Relatório
Introdução a Inteligência Artificial
Universidade Federal de Minas Gerais

2023-10-08

1 Introdução

Neste trabalho nos propomos a implementar diversos algoritmos de busca para a resolução do problema conhecido como 8-puzzle. Nesse problema, existe um tabuleiro de dimensão 3×3, e oito das nove posições do tabuleiro são ocupadas por peças numeradas de 1 a 8, enquanto uma posição é considerada vazia, ou alternativamente numerada como 0. Podemos realizar um movimento da posição vazia a cada passo, e tal movimento deve ser um de quatro possíveis: para cima, para baixo, para a esquerda, ou para a direita. O objetivo do jogo é atingir um estado alvo com o menor número de movimentos da posição vazia possível.

Figure 1: Imagem ilustrando um estado inicial qualquer e o estado alvo desejado.



O estado alvo neste trabalho será o estado exibido na imagem anterior: a peça vazia na posição 9 e as demais peças em suas respectivas posições de acordo com seu valor.

2 Modelagem

O programa foi desenvolvido na linguagem Python, no sistema operacional Windows 11. O programa foi implementado e compilado em uma máquina com 16GB de RAM em um processador Intel Core I5-9400F.

2.1 Estruturas de Dados

Para modelar os diversos componentes envolvidos em algoritmos de busca, foram implementadas três estruturas de dados como classes em Python, além de algumas funções de manipulam tais estruturas e auxiliam na implementação dos algoritmos que serão apresentados nas próximas seções.

- MATRIX: Classe representando uma matriz com entradas arbitrárias. Essa classe é essencialmente um wrapper de acesso a uma lista de dados que é passado como parâmetro na inicialização, de modo que o usuário possa acessar posições na matriz que na verdade são elementos da lista. Essa classe é utilizada para representar o tabuleiro do jogo 8-puzzle, ou seja, inicialmente uma lista de inteiros que representa as peças no tabuleiro é passada para o método construtor, e então a partir desse momento o usuário pode acessar o tabuleiro em forma de matriz. Essa classe possui três métodos além do construtor: CHECK_INPUT, que checa se uma dada entrada em forma de índice de linha e coluna é válida, GET, e SET.
- STATE: Classe que representa um estado do jogo. Armazena um objeto de tipo Matrix, e dois inteiros indicando a posição do espaço vazio no tabuleiro.
- NODE: Classe que representa um nó na árvore de busca. Armazena um objeto de tipo State, um inteiro indicando a profundidade do nó na árvore, um objeto de tipo Node que indica o nó pai, um inteiro indicando o custo associado ao nó, e uma string que armazena o *hash* do estado, isto é, uma

representação em string da lista de inteiros que representa o tabuleiro. O aspecto mais importante dessa classe é o fato de que ela sobrecarrega as operações de igualdade e de "menor que": dois nós são iguais se o hash de seus estados é igual, e um nó e menor que o outro se o seu custo for menor que o custo do outro nó.

Todas as estruturas anteriores também definem um método COPY, que retorna uma cópia dos seus conteúdos armazenados. Com respeito as funções que auxiliam a busca de estados, as seguintes foram implementadas:

- IS_GOAL: recebe um Nó e retorna 1 caso o estado desse nó seja o estado alvo, e 0 caso contrário.
- EXPAND_NODE: função sucessora que recebe um Nó e retorna uma lista contendo os estados que são atingíveis a partir desse nó, isto é, tenta mover o espaço vazio nas quatro direções possíveis (cima, baixo, direita, esquerda), e então cria novos estados para cada um dos novos tabuleiros.
- PRINT_STATE: recebe um estado e imprime o tabuleiro em forma de matriz 3×3.
- PRINT_PATH: recebe um Nó e imprime o caminho feito desde a raiz até ele.
- PROCESS_INPUT: recebe uma lista de inteiros representando o estado inicial do tabuleiro e retorna um Nó que representa a raiz da árvore.

Alguns dos algoritmos discutidos nas próximas seções necessitam de estruturas do tipo Fila, Pilha e Heap, no entanto, é possível realizar uma implementação indireta de tais estruturas somente utilizando a estrutura de Lista nativa do Python. Para implementar uma Fila, basta adicionar elementos somente no fim da Lista e remover elementos somente do começo, para implementar a Pilha basta adicionar elementos somente no começo e também remover elementos somente no começo. Com relação ao Heap, basta utilizar a operação nativa do Python HEAPPUSH, que recebe uma lista de objetos e trata-a como um Heap, inserindo o novo elemento na posição adequada de acordo com a operação de "menor que", ou seja, como sobrecarregamos tal operação para o objeto Node, podemos criar uma Lista de nós e sempre inserir elementos utilizando o HEAPPUSH, de modo a garantir que a lista esteja sempre ordenada de acordo com o nó de menor custo.

2.2 Algoritmos

Seis algoritmos distintos de busca de estados foram implementados para o trabalho em questão. Iremos agora fornecer uma breve discussão acerca do funcionamento de tais algoritmos.

- **Busca sem informação**:
  - **Breadth-First search (BFS)**: o algoritmo BFS é um algoritmo clássico de busca em grafos, e funciona por meio de uma fila seguindo a ordem FIFO. A cada passo, o primeiro nó da fila é expandido, e então seus filhos são adicionados ao final da fila. Esse processo é repetido até que a fila esteja vazia ou até que o estado alvo seja atingido. O BFS é um algoritmo completo e ótimo, isto é, ele sempre encontra a solução, caso exista, e a solução encontrada é ótima.
  - **Iterative Deepening search (IDS)**: o algoritmo IDS consiste em realizar múltiplas buscas em profundidade (DFS) de maneira limitada até que o estado alvo seja atingido ou até que todos os estados sejam explorados. A busca DFS utiliza uma pilha seguindo a ordem LIFO, e a cada passo o nó do topo da pilha é explorado e seus filhos são adicionados ao topo caso sua profundidade não exceda a profundidade máxima permitida, e então esse processo é realizado até que a pilha fique vazia ou até que o estado alvo seja encontrado. Essa busca DFS é realizada para profundidades máximas cada vez maiores, isto é, a cada iteração do algoritmo se realiza uma busca DFS com profundidade máxima igual a anterior acrescida de um. O IDS é um algoritmo completo e ótimo.
  - **Dijkstra search (DKS)**: o algoritmo DKS consiste no algoritmo de Dijkstra para computar o menor caminho entre um par de nós de um grafo com pesos não-negativos. Dado um nó arbitrário, deve-se determinar um peso para se transicionar desse nó para um de seus filhos – no caso do 8-puzzle o peso é uniforme, isto é, toda ação tem o mesmo custo. Cria-se então uma fila de prioridades ordenada pelo custo de se chegar a um dado nó, sendo que tal custo é dado pelo valor do custo do nó anterior mais o custo de se realizar a ação de transição. A cada iteração, o primeiro elemento da fila é explorado, e os seus nós são adicionados a fila e ordenados de acordo com seus custos. Esse procedimento se repete até que o nó alvo seja atingido ou até que a fila fique vazia. O algoritmo DKS é completo e ótimo, e no caso do 8-puzzle ele tem o mesmo comportamento do algoritmo BFS, tendo em vista que toda ação tem o mesmo custo.
- **Busca com informação**:

  - **Greedy Best-First search (GRS)**: o algoritmo GRS consiste em uma busca gulosa no espaço de estados do problema em questão. O algoritmo utiliza uma função heurística para calcular o custo dos sucessores de um dado nó, e então ele prioriza a expansão dos nós com menor custo de acordo com a heurística. O algoritmo não é ótimo nem completo, no entanto é extremamente eficiente.
  - **A* search (ASS)**: o algoritmo ASS consiste de uma combinação do algoritmo DKS com uma função heurística que calcula o custo de um dado estado. O algoritmo possui um funcionamento essencialmente igual ao DKS, no entanto, o custo de cada nó é dado pela soma do custo real do nó com o valor da função heurística para o estado do nó. Esse algoritmo é completo, e é ótimo caso a heurística utilizada seja admissível.

- **Busca Local**:
  - **Hill Climbing search (HCS)**: o algoritmo HCS é um algoritmo guloso de busca local de funcionamento relativamente simples: dado um nó qualquer, ele calcula o custo de todos os nós filhos de acordo com alguma função heurística, caso exista algum filho com custo estritamente menor que o custo do pai, ele passa para este filho e repete o processo, contrário ele termina e retorna o nó atual, que é garantidamente um ponto extremo local com respeito a heurística utilizada. Note que, no caso em que o custo do filho é igual ao do pai, é definida uma quantidade máxima de iterações – referidas como passos laterais – onde o algoritmo continua. Esse algoritmo não é completo e nem ótimo.

2.3 Detalhes de Implementação

Nesta seção iremos discutir alguns detalhes relevantes de implementação que distinguem nossa abordagem das definições teóricas dos algoritmos apresentados.

- Um dos principais desafios para implementação de funções de busca é a checagem de estados repetidos, isto é, **como saber se um dado estado já foi explorado**? Nossa solução é simples: utilizaremos a estrutura SET nativa de Python, que representa um conjunto matemático, isto é, um conjunto de elementos sem repetição onde a ordem não importa. Tal conjunto irá armazenar os *hashs* de cada estado, que são representações em string de um dado tabuleiro, e.g., o estado

$$\begin{bmatrix} 1 & 4 & 3 \\ 2 & 0 & 8 \\ 5 & 6 & 7 \end{bmatrix}$$

  é representado como "143208567". Note que essa representação é de fato única para um estado, ou seja, duas representações são iguais se, e somente se, os dois estados forem iguais. O conjunto de estados explorados é então um SET de hashs de estados, e para checar se um estado já foi explorado basta checar se seu hash está no conjunto. Existem diversas outras soluções possíveis, e.g., comparar as matrizes termo a termo ou utilizar um conjunto de listas representando os estados, mas testes empíricos nos mostraram que essa abordagem é a mais eficiente no quesito tempo, majoritariamente devido ao fato da estrutura SET ser altamente otimizada e de fácil manipulação.

- Além isso, existe também a problema da fronteira: **como implementar a fronteira dos algoritmos de maneira eficiente**? Para resolver esse problema, optamos por duas abordagens distintas. Para o algoritmo BFS, utilizamos duas listas, uma contendo objetos do tipo nó e outra contendo os hashs dos estados de cada nó, e para checarmos se um nó já estava na fronteira basta checar se seu hash está na lista de hashs. Essa solução se mostrou mais rápida em termos de tempo do que utilizar a operação sobrecarregada de igualdade de nós para checar se um objeto Node está na lista de Nós, e note que o custo de espaço de tal implementação é $O(2b^d) = O(b^d)$, ou seja, não alteramos o custo de espaço em termos assintóticos. Já para os algoritmos DKS, ASS e GRS, o custo adicional de ordenar duas listas se mostrou mais caro do que utilizar a operação de igualdade sobrecarregada, e portanto mantemos somente uma lista de objetos Node para representar a fronteira.

- Por fim, optamos por realizar uma implementação mais cautelosa do algoritmo GRS. A definição teórica do algoritmo a princípio não requer que um um conjunto de nós explorados sejam mantidos, no entanto, observamos que o custo adicional de manter tal estrutura é preferível tendo em vista o ganho em eficiência que ela provê. Ademais, é possível garantir que o uso desse estrutura faz sentido no contexto do GRS: se um estado já foi explorado, como o custo é baseado somente no valor da função heurística, não existe motivo para explorá-lo de novo, e o mesmo vale para estados que já estão na fronteira. Portanto, nossa implementação do algoritmo GRS se assemelha a implementação do algoritmo DKS, no entanto, o custo leva somente em conta o valor da heurística aplicada em um dado nó, ignorando completamente o custo de se chegar naquele nó.

2.4 Heurísticas

Para os algoritmos de busca com informação, utilizados duas funções heurísticas para estimar o custo de transição para um dado estado. Tais funções são descritas a seguir.

- **Soma das distâncias de Manhattan**: Essa heurística consiste em calcular, para cada posição do quebra-cabeça, a distância de Manhattan entre a posição atual ocupada por uma peça e a posição que tal peça deve ocupar no estado alvo. A distância de Manhattan é definida como a menor quantidade de passos necessários para chegar de uma posição (i, j) qualquer no tabuleiro até uma posição (l, k) somente por meio de passos horizontais e verticais, isto é, não são permitidos passos diagonais. Tais distâncias são então somadas para se obter o valor final da Heurística.
- **Número de peças fora de posição**: Essa heurística consiste simplesmente de contar a quantidade de peças que se encontram fora de suas posições no estado alvo.

Note que ambas a heurísticas anteriores são admissíveis. De fato, no caso da distância de Manhattan, ela mede precisamente a quantidade de passos válidos necessários para colocar cada peça em sua posição adequada, no entanto ela não considera as possíveis interações entre os movimentos de cada peça, ou seja, ela é otimista, e portanto é menor ou igual a quantidade real de passos necessários. A segunda heurística é trivialmente admissível: certamente se uma peça está fora de posição será necessário ao menos um movimento para coloca-la de volta em posição, e portanto o valor de tal heurística é menor ou igual a quantidade real de passos necessários para se chegar ao estado alvo.
Por padrão, o algoritmo ASS utiliza a heurística de distância de Manhattan, e os algoritmos GRS e HCS utilizam a heurística de número de peças fora de posição (que nada mais é que a função de custo real do problema). Ademais, o algoritmo HCS utiliza um valor máximo de passos laterais de 20.

3 Análise Quantitativa

Nesta seção, iremos realizar um experimento para avaliar a eficiência e eficácia dos algoritmos propostos. Primeiro iremos comparar os algoritmos de cada grupo, e então faremos uma comparação entre todos os algoritmos. Os dados do experimento serão os disponibilizados no arquivo "npuzzle.pdf" fornecido no enunciado do trabalho, e estes consistem de 32 instâncias do problema, com soluções que vão de 0 até 31. O eixo x de todos os gráficos desse seção será dado pela número ótimo de passos para a solução, que também indexa a instância dada. Alguns exemplos de soluções podem ser encontrados no apêndice A deste documento.

3.1 Busca sem informação

Nesta seção, iremos avaliar e comparar a performance dos algoritmos BFS, IDS e DKS. A análise irá focar na quantidade de nós expandidos e tempo gasto para execução, tendo em vista que todos os algoritmos são garantidamente ótimos, ou seja, a solução encontrada é sempre a melhor possível. Iremos avaliar o desempenho dos algoritmos BFS e DKS separadamente, tendo em vista que estes possuem um tempo de execução razoável para as 32 instâncias fornecidas no enunciado. O algoritmo IDS por outro lado se torna praticamente intratável no quesito tempo quando a solução necessita mais de vinte passos, devido ao fato de tal algoritmo não utilizar nenhum método de detecção de ciclos ou de nós explorados.
A figura 2 nos mostra que os algoritmos BFS e DKS expandem um número aproximadamente igual de nós em cada instância, e na verdade esse número só é diferente pois a implementação do BFS utiliza um mecanismo de checagem preventiva, ou seja, checa todos os filhos do nó atualmente explorado sem de fato explorá-los. No quesito tempo no entanto o DKS é substancialmente pior que o BFS, novamente algo esperado: o custo adicional de ordenar o heap a cada expansão faz com que o algoritmo seja mais caro, e no contexto do 8-puzzle esse heap é redundante, pois o custo de transição entre nós é constante e igual a um.
A figura 3 mostra os resultados para o algoritmo IDS para as 21 primeiras instâncias fornecidas. Podemos notar que a partir da vigésima instância, o tempo de execução do IDS já se aproxima de uma hora, e a partir disso ele cresce substancialmente, fazendo com que o problema seja praticamente intratável. A quantidade de nós expandidos também é extremamente alta, tendo em vista que o IDS não guarda os nós explorados, e portanto explora estados já explorados diversas vezes. De modo geral, para o problema do 8-puzzle, o algoritmo IDS não parece ser adequado para instâncias que requerem muitos passos até a solução ótima.

3.2 Busca com informação

Nesta seção, iremos avaliar e comparar a performance dos algoritmos ASS e GRS. A análise irá focar na quantidade de nós expandidos, no tempo de execução, e na quantidade de passos da solução. Essa última métrica é essencial para analisarmos o algoritmo GRS, tendo em vista que este não é ótimo.

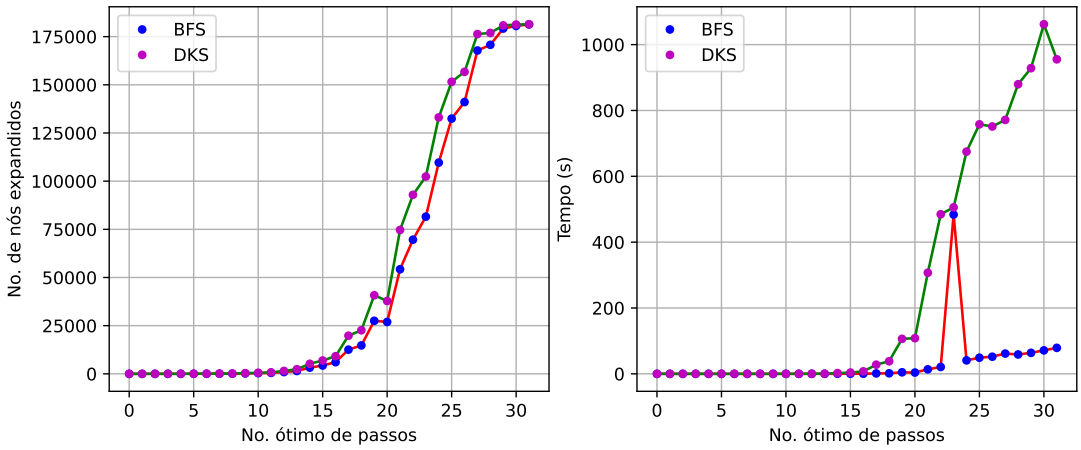


Figure 2: Performance dos algoritmos BFS e DKS.

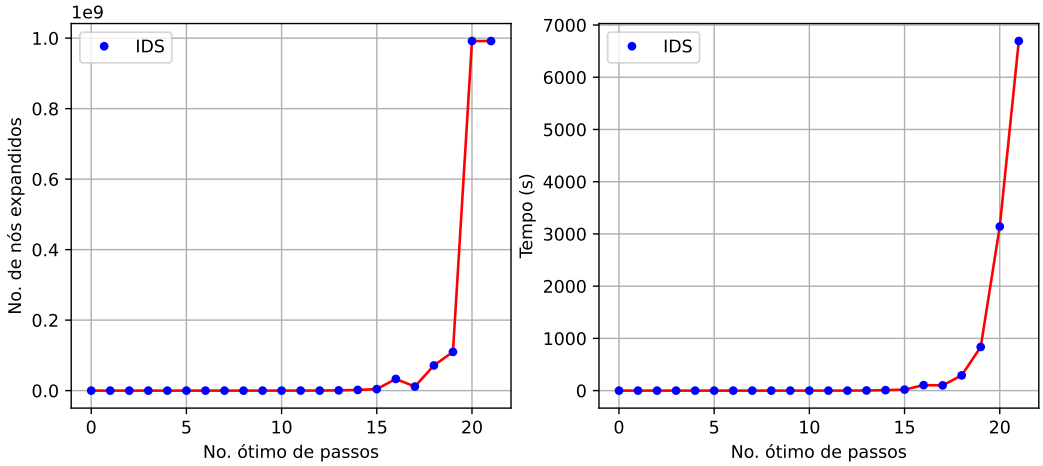


Figure 3: Performance do algoritmo IDS.

Primeiro, iremos comparar a performance dos algoritmos ASS e GRS com respeito às duas heurísticas propostas. O sufixo _M irá indicar a heurística de manhattan, e o sufixo _O irá indicar a heurística de quantidade de peças fora de posição.

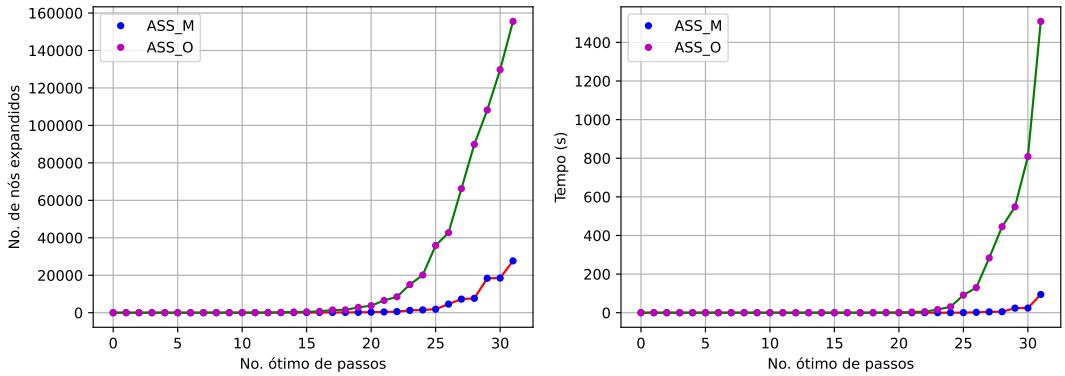


Figure 4: Performance do algoritmo ASS com duas heurísticas distintas.

Na figura 4, observarmos que, tanto com respeito ao tempo quanto com respeito à quantidade de nós expandidos, a heurística de manhattan se mostra substancialmente melhor para o algoritmo ASS. Isso se justifica pelo fato de que a heurística de manhattan domina a heurística de quantidade de nós expandidos, isto é, o valor de uma é sempre maior ou igual ao de outra.

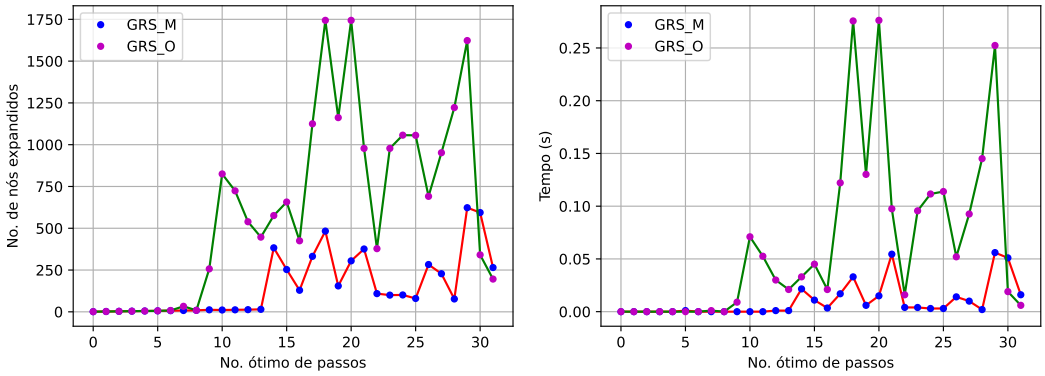


Figure 5: Performance do algoritmo GRS com duas heurísticas distintas.

Na figura 5, observarmos também que a heurística de manhattan se mostra substancialmente melhor. Ou seja, em ambos os casos, a heurística de manhattan é a que apresenta a melhor performance.
Agora iremos comparar a performance do algoritmo GRS utilizando a heurística de número de peças fora de posição com a performance do algoritmo ASS utilizando a heurística de Manhattan.

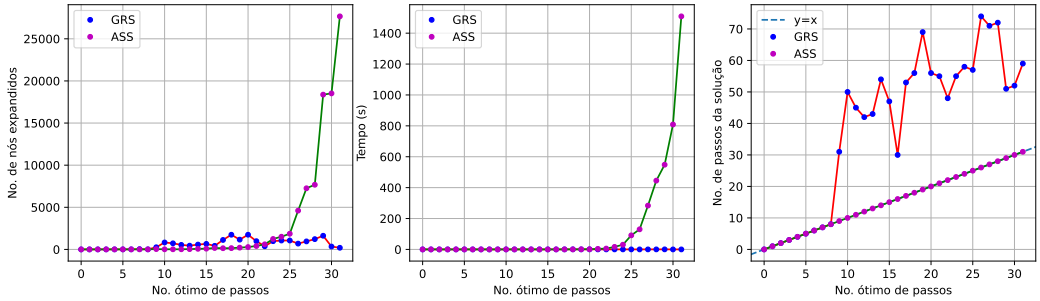


Figure 6: Performance dos algoritmos ASS (heurística de manhattan) e GRS (heurística de número de peças fora de posição).

A figura 6 nos mostra que o algoritmo GRS supera o algoritmo ASS em duas das três métricas: ele é mais rápido e explora substancialmente menos nós. No entanto, o gráfico três nos mostra empiricamente algo que já fora previamente discutido: o ganho de eficiência do algoritmo GRS vem ao custo do mesmo não ser ótimo, isto é, a solução encontrada nem sempre será a de menor custo, ao contrário do algoritmo ASS que sempre encontra a solução ótimo, como pode ser observado na figura anterior. Notamos também que, para as instâncias do problema utilizadas, o algoritmo GRS foi capaz de encontrar uma solução para o problema, isto é, apesar de não ser ótima, ele foi capaz de encontrar um caminho até o tabuleiro alvo.

3.3 Busca local

Nesta seção, iremos avaliar a performance do algoritmo HCS. A análise irá focar na quantidade de nós expandidos, no tempo de execução, e na quantidade de passos da solução. Assim como o GRS, o HCS não é garantidamente ótimo, e portanto a quantidade de passos da solução é essencial para analisarmos a sua performance.

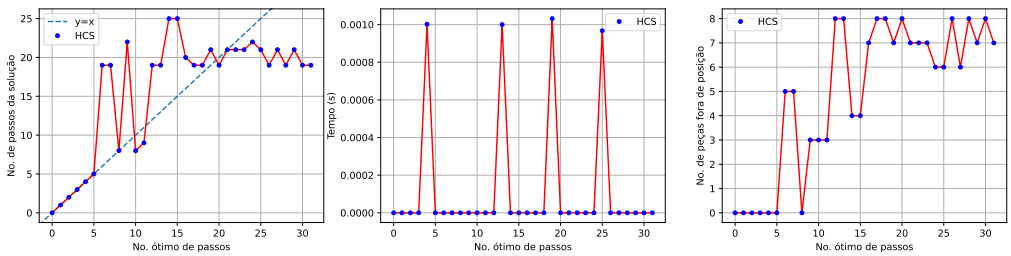


Figure 7: Performance do algoritmo HCS.

A figura 7 nos mostra que, em termos de tempo de execução, o HCS é um dos melhores algoritmos dentre os propostos. O HCS expande, em média, uma quantidade de nós próxima a quantidade de passos laterais permitidos, indicando que, no problema do 8-puzzle, é muito fácil fazer com que o HCS encontre mínimos locais. Por outro lado, também percebe-se que para a maioria das instâncias o HCS apresenta um número relativamente alto de peças fora de posição, ou seja, as soluções encontradas pelo HCS não possuem uma qualidade alta.

3.4 Discussão dos resultados

A análise dos resultados anteriores nos permite realizar uma comparação entre todos os algoritmos propostos. Dentre os algoritmos ótimos e completos, isto é, BFS, ASS, DKS e IDS, o IDS se mostrou o pior nos quesitos tempo de execução e quantidade de nós expandidos. Dado o contexto, o DKS também se mostrou um tanto ineficiente devido a redundância da operação de ordenar um heap para o problema do 8-puzzle. O BFS se mostrou extremamente bom, devido a sua simplicidade e tempo de execução baixo, no entanto isso vem ao custo de gastar mais memória. Finalmente, o algoritmo ASS com a heurística de manhattan se mostrou o melhor dentre os candidatos: seu tempo de execução é ligeiramente maior que o do BFS para instâncias com número de passos alto, mas ele é o algoritmo que explora menos nós. De modo geral, tanto o BFS quanto o ASS se mostraram bons candidatos para solucionar o problema do 8-puzzle de forma eficiente.
Com respeito aos algoritmos que não são completos e nem ótimos, isto é, GRS e HCS, o GRS se mostrou significativamente melhor em comparação com o HCS. Em todas as 32 instâncias, o GRS foi capaz de encontrar uma solução para o problema, mesmo que tal solução não fosse ótima, enquanto o HCS foi capaz de encontrar uma solução somente em 7 das 32 instâncias. Ambos os algoritmos possuem tempo de execução e gastos de memória mínimos, no entanto, isso claramente vem em detrimento da qualidade das soluções encontradas.

4 Conclusão

Neste trabalho, foram feitas implementações de diversos algoritmos clássicos de busca em estados para o problema do 8-puzzle. As principais diferenças entre tais algoritmos foram discutidas em detalhes, bem como alguns aspectos técnicos que distinguem nossa implementação das formulações teóricas dos algoritmos.
A performance dos algoritmos foi analisada de acordo com diversas métricas de desempenho que capturam a eficácia e eficiência dos métodos, permitindo com que fizéssemos uma análise comparativa entre os algoritmos propostos. De modo geral, como era de se esperar, o algoritmo A^* foi método completo e ótimo que apresentou melhor desempenho quando comparado com os demais métodos, em especial com os algoritmos de busca sem informação que também são completos e ótimos. Os algoritmos de Hill Climbing e de Busca Gulosa se mostraram extremamente eficientes, no entanto tal eficiência vem ao custo de perder a garantia de otimalidade da solução.
De modo geral, esse trabalho permitiu um exercício empírico dos conceitos vistos em aula, de modo a comprovar os resultados teóricos previstos por meio de experimentos computacionais.

A Exemplos de soluções

Nesta seção, iremos fornecer alguns exemplos de soluções encontradas pelos algoritmos para uma instância específica do problema do 8-puzzle. Por motivos de espaço deste documento, iremos apresentar as soluções encontradas por um algoritmo de cada grupo, isto é, busca sem informação (BFS), busca com informação (ASS),e busca local (HCS), e iremos considerar o seguinte estado inicial:

$$\begin{bmatrix} 1 & 0 & 2 \\ 8 & 5 & 3 \\ 4 & 7 & 6 \end{bmatrix}$$

O número de passos ótimo para tal estado é 9.
Note que os algoritmos BFS e ASS encontram a solução ótima, enquanto o HCS fica preso em um mínimo local, que é ilustrado pela movimentação cíclica das peças 0 e 8 até que a quantidade máxima de passos laterais seja atingida.

Figure 8: Solução encontrada pelo algoritmo BFS.

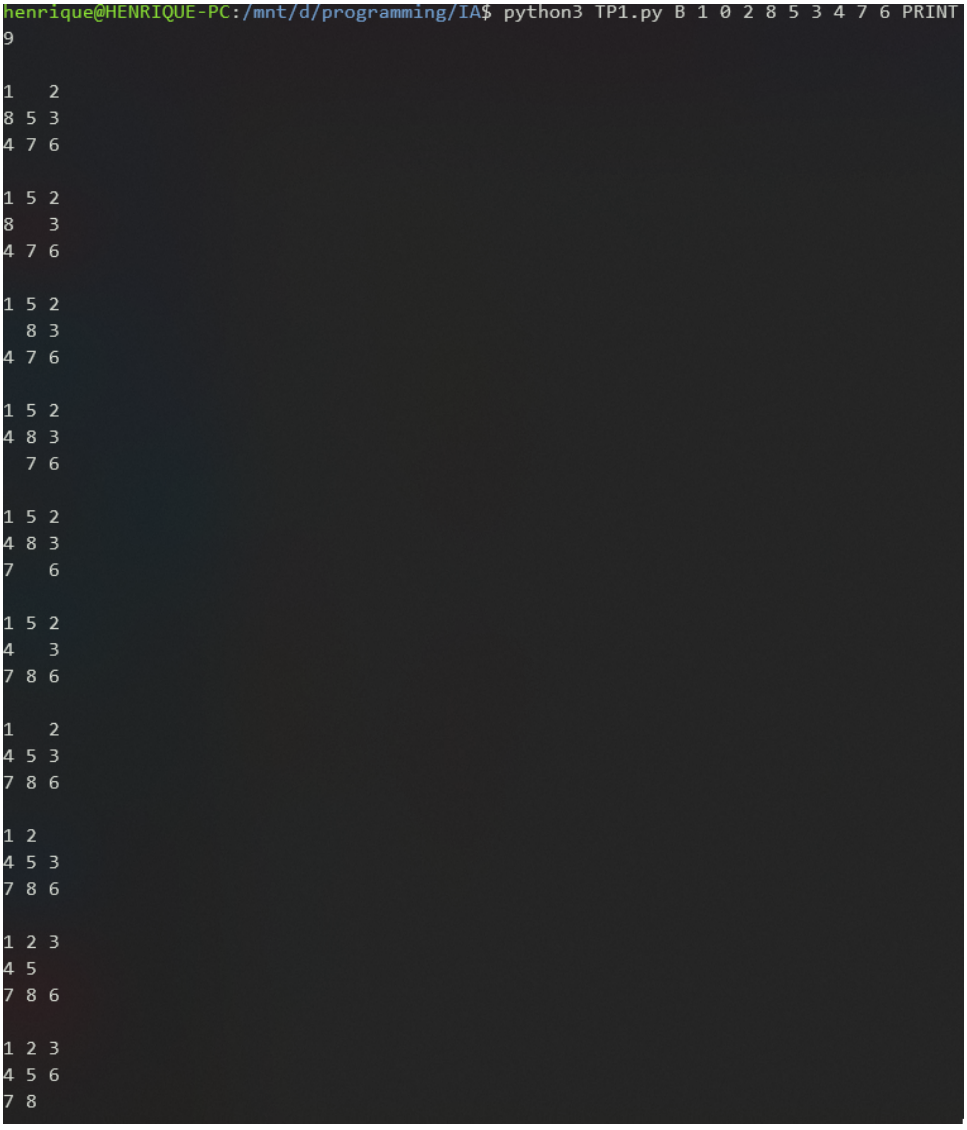


Figure 9: Solução encontrada pelo algoritmo ASS.

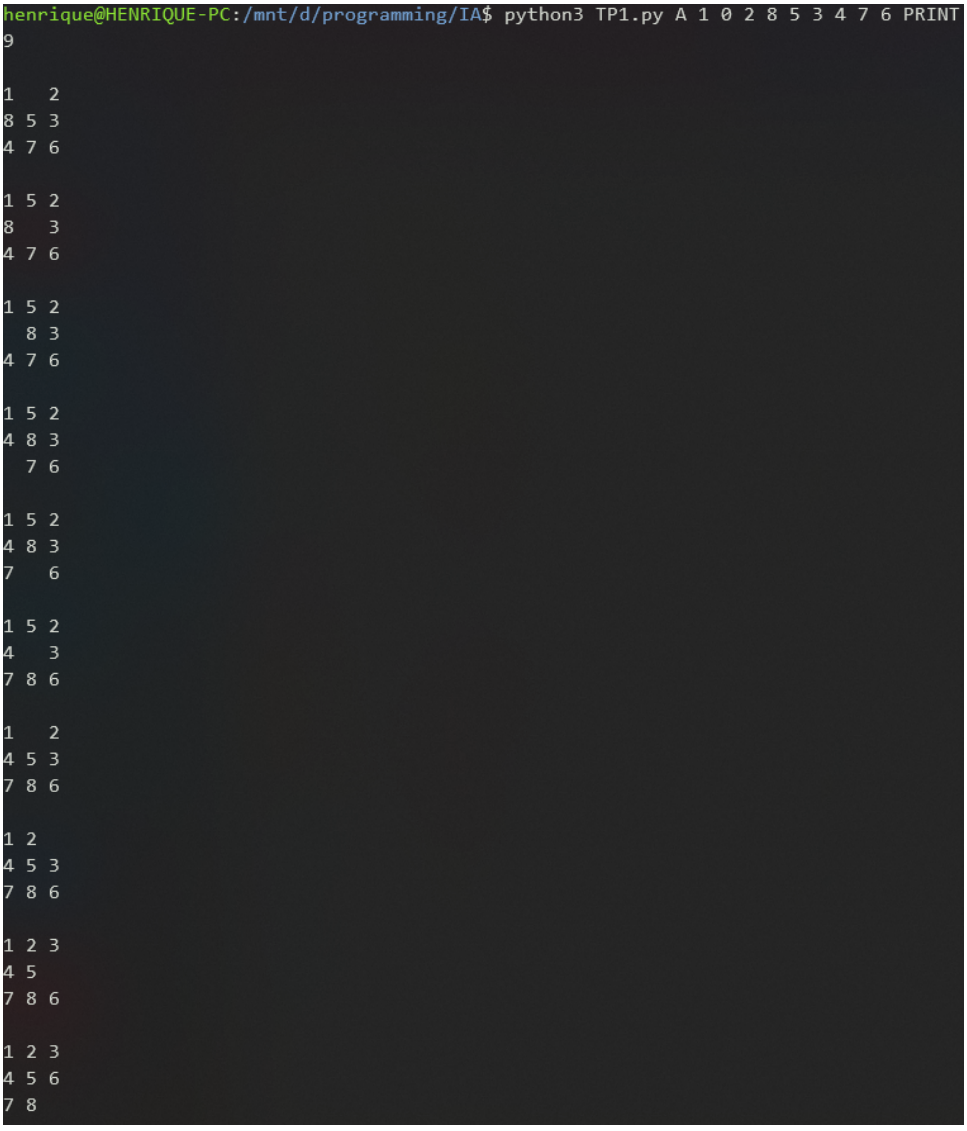


Figure 10: Solução encontrada pelo algoritmo HCS. A solução foi cortada devido ao espaço da página, mas as demais iterações são precisamente o movimento cíclico das peças 8 e 0.