Trabalho Prático - Grafos
Projeto e Análise de Algoritmos

Daniel Pimentel Kansaon

INTRODUÇÃO

Para a implementação do N-Puzzle, foi desenvolvido um algoritmo utilizando a linguagem Python na versão 2.7.12 Para o desenvolvimento, alguns ajustes foram feitos no algoritmo Breadth-First Search (BFS), para que assim, fosse possível encontrar o custo (nº passos) da melhor solução do N-Puzzle dada uma entrada. Todo o código desenvolvido no TP, está contido no script "calcular_passos_npuzzle.py".

No algoritmo proposto, cada possibilidade de movimento é representada como um vértice, já os movimentos possíveis a partir de um estado atual, é tratado como vértices adjacentes.

A Figura 1, exibe um diagrama simplificado que representa os passos que o algoritmo realiza para encontrar a solução do N-Puzzle. A partir do input inicial, que é fornecido pelo arquivo [in], o algoritmo gera todos os movimentos possíveis, seguindo as ordens (CBED). Após gerar as possibilidades, o algoritmo verifica se a solução do N-Puzzle está entre os movimentos gerados. Caso a solução não seja encontrada, todos os movimentos são adicionados em uma fila (FIFO), para posteriormente cada elemento ser visitado e assim gerar os novos movimentos possíveis. No código não existe uma representação explícita de uma árvore, mas o conceito geral da estrutura de dados é utilizada na aplicação através de outras estruturas. A Figura 2 exibe como a solução N-Puzzle é representada. Os vértices pretos são os vértices já visitados, enquanto o verde é a solução. Pode-se observar que, o input [1,2,,3,4,0,6,7,8] é considerado como raiz da árvore, a partir da raiz, são gerados todas as possibilidades existentes, que são adicionadas em outro nível da árvore. O processo é repetido até encontrar a solução. No exemplo, o custo da solução é 3, ou seja, está no terceiro nível da árvore. Por fim, existe um contador que é incrementado toda vez que um novo nível é gerado na árvore, assim, a quantidade de passos até encontrar a solução é representada pela quantidade de níveis gerados na árvore.

No algoritmo, vale destacar que, se o custo retornado for igual a -1, significa que não existe solução. Já se for igual a 0, significa que o input fornecido já é a solução. Por fim, custo maior que 0, indica o número de passos necessários para achar a solução.

Na implementação, a principal função da solução é a "main", a partir dela, todas as etapas do algoritmo são acionadas. Já a função "calcular_passos" é onde contém a lógica da solução, nela todas as etapas para cálculo do custo é feita. Por fim, a função "retornar_acoes_possiveis" determina todas as movimentações possíveis a partir de um estado inicial. O método "retornar_acoes_possiveis", retorna os possíveis movimentos dado

um estado inicial. Nesse caso, é seguido a seguinte ordem de visita: Cima, Baixo, Esquerda, Direita. (CBED).

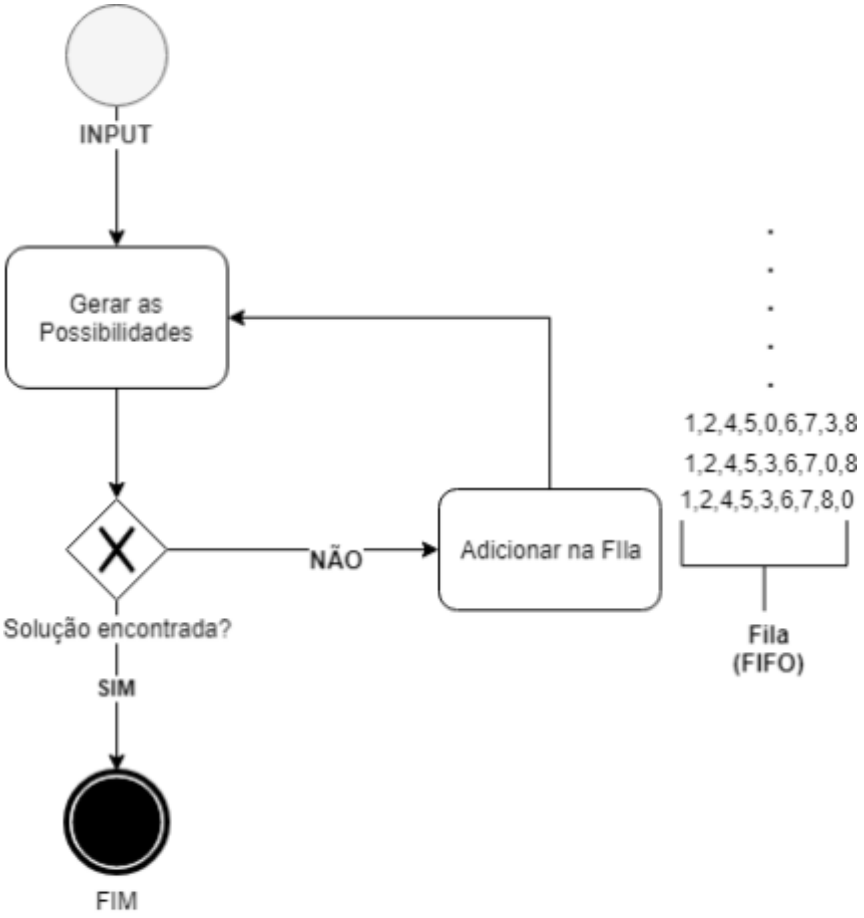

Figura 1. Diagrama de passos do algoritmo.

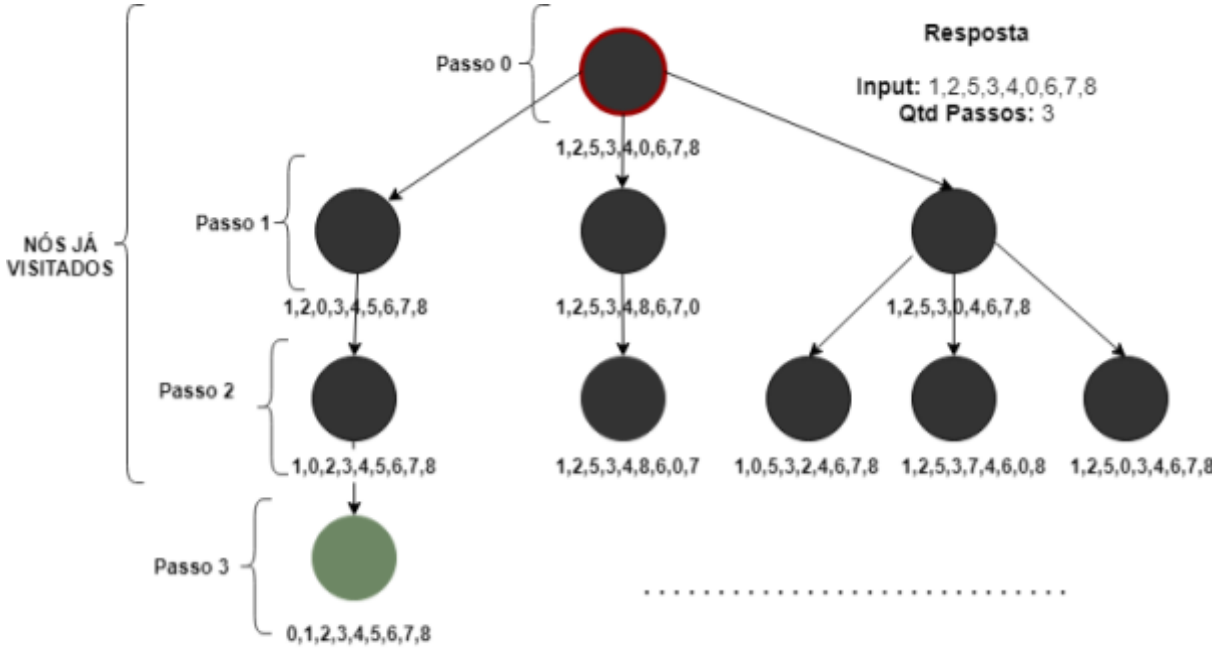

Figura 2. Representação do algoritmo proposto.

Além disso, no trabalho, foi necessário tratar a ocorrência de movimentos repetidos, para que um nó seja visitado apenas uma vez. Assim, não existem estados repetidos na árvore, pois, se a solução não tratasse essa situação, o código poderia entrar em um loop infinito e nunca encontrar a solução do N-Puzzle. Nó código, existe uma lista "movimentos_ja_feitos_vertpretos" que contém todos os vértices já visitados, que são os vértices pretos. Toda vez que o algoritmo vai gerar as novas possibilidades, é verificado se o estado atual já está na lista de movimentos já feitos, caso esteja, o movimento não é considerado. Dessa forma é possível garantir que o algoritmo não entrará em um loop sem fim.

Na implementação, o algoritmo tem dois pontos de parada. O primeiro ponto, é quando a solução do N-Puzzle é encontrada, nesse caso o algoritmo finaliza retornando o custo para encontrar a solução, ou seja, o nível da árvore que a solução está. O segundo ponto, é quando não existe solução. O algoritmo vai gerando todas as possibilidades e chega a um momento em que todas as possibilidades existentes já foram geradas. Nessa situação, é retornado o valor -1 como custo da solução, o valor negativo significa que não existe solução. A resposta do algoritmo, é salva no arquivo [out].

No algoritmo foi utilizado basicamente duas estruturas de dados. A primeira é a Queue, que foi utilizada para armazenar os vértices que ainda precisam ser visitados, é semelhante a lista de vértices brancos no BFS. No código, para usar a queue, foi utilizado a biblioteca: "from collections import deque", pois essa estrutura é mais rápida para adicionar e remover elementos do que a queue tradicional. Um dos motivos que justifica o uso da queue, foi a necessidade de representar a ordem de visita (CBED) dos nós. Dessa forma, basta adicionar os vértices na ordem de visita, pois a queue tem estrutura FIFO. A outra estrutura usada foi o Set, que é uma coleção que representa um conjunto em Python. No código, a lista "movimentos_ja_feitos_vertpretos" é um Set(). Essa estrutura foi escolhida, pelo fato de ser mais rápida que uma lista tradicional. A lista "movimentos_ja_feitos_vertpretos" é verificada constantemente para constatar se vértices já foram visitados, por isso é necessário que a estrutura seja eficiente para evitar gargalos.

Para executar o algoritmo, basta adicionar no arquivo [in], um input válido com 8 dígitos. Assim, o algoritmo iniciará a execução para encontrar o custo da solução. Caso o input seja inválido, será enviado um alerta para o usuário.

ANÁLISE DE COMPLEXIDADE

A análise da complexidade será separada em dois tópicos, complexidade de tempo e memória.

Complexidade de Tempo

Uma observação importante a ser realizada na hora de analisar o tempo para encontrar a solução do N-Puzzle, é determinar quantas possibilidades existem em uma possível partida. Inicialmente podemos fazer uma análise observando o número de posições do N-Puzzle, dessa forma podemos afirmar que existem (n!) possíveis movimentos. Mas quando começamos observar a quantidade de possibilidades considerando que não pode existir um estado repetido e que é preciso respeitar as regras de movimentação (somente são aceitos movimentos: Cima, Baixo, Esquerda e Direita), a análise feita anteriormente precisa ser refeita, pois nesse caso, dado um estado inicial não é possível chegar em todos os movimentos possíveis (n!).

Assim precisamos fazer uma análise mais justa. Para descobrir a quantidade de possibilidades do 8-Puzzle, foi executado o algoritmo com um input que não tem solução (6,1,0,4,8,2,7,3,5). Com isso, foi constatado que o algoritmo percorreu (181.440) movimentos até retornar que não existe solução. Esse valor é bem inferior que (9! = 362.880). Após outras execuções, foi concluído que dado um estado inicial, só é possível encontrar (9!/2) possibilidades. Para alcançar (9!), seria necessário em algum momento remover as peças de lugar e recolocar em uma ordem diferente. Como a regra de movimentação não permite a remoção de peças, podemos concluir que existem (n!/2) possibilidades dado uma entrada inicial.

O pior caso do algoritmo, é quando não existe solução para o problema, assim o algoritmo gera toda a árvore de possibilidades para no fim verificar que não existe solução.

Para encontrar a solução do 8-Puzzle, é relativamente rápido, no pior caso é feito no máximo (n!/2) permutações. Só que quando se aumenta o tamanho do N-Puzzle, a largura do tabuleiro também aumenta, fazendo assim com que exista mais movimentos possíveis a partir de um estado. Assim, podemos afirmar que a complexidade de tempo a partir que o (n) aumenta é: O(n!), pois no pior caso, o algoritmo precisará percorrer todas as possibilidades. Já as operações executadas no programa em sua maioria são O(1), exceto a operação que consulta a lista para verificar se o movimento já foi realizado. Essa operação é executada em cada iteração, precisando percorrer toda a lista para encontrar um elemento, assim, essa operação é O(n!). **Dessa forma, concluímos que no pior caso, a complexidade do algoritmo é O(n! * n!).**

A Figura 3, exibe um gráfico onde é feito uma comparação da quantidade de memória e tempo gasto durante a execução do algoritmo dado uma entrada. No eixo X, temos a quantidade de passos (custo) necessários para encontrar a solução. Já o eixo Y, representa o valor de cada métrica. Além disso, a memória está representada em Megabyte, já o tempo em Segundos. Para gerar o gráfico, foi obtido os casos de teste fornecidos no Moodle, e em seguida o algoritmo foi executado obtendo as métricas de memória e tempo gasto para plotagem do gráfico. Sabe-se que o tempo gasto em uma execução pode variar dependendo

da máquina e de outros fatores. Visto que o objetivo do trabalho não é determinar um tempo justo, os fatores foram desconsiderados para a geração do Gráfico. Foi utilizado um computador com quatro núcleos de CPU e com quatro gigas de memória Ram. Observando o gráfico, é possível notar que com o aumento do custo da solução, a memória e o tempo também vão aumentando. No gráfico também é visível, que a memória (linha amarela) aumenta mais rapidamente que o tempo (linha azul), ou seja o algoritmo implementado consome uma maior quantidade de memória do que tempo, isso é explicável devido a necessidade de armazenar as possibilidades já percorridas para evitar considerar movimentos repetidos. Então, quanto maior os passos necessários para a solução, maior a quantidade de memória gasta.

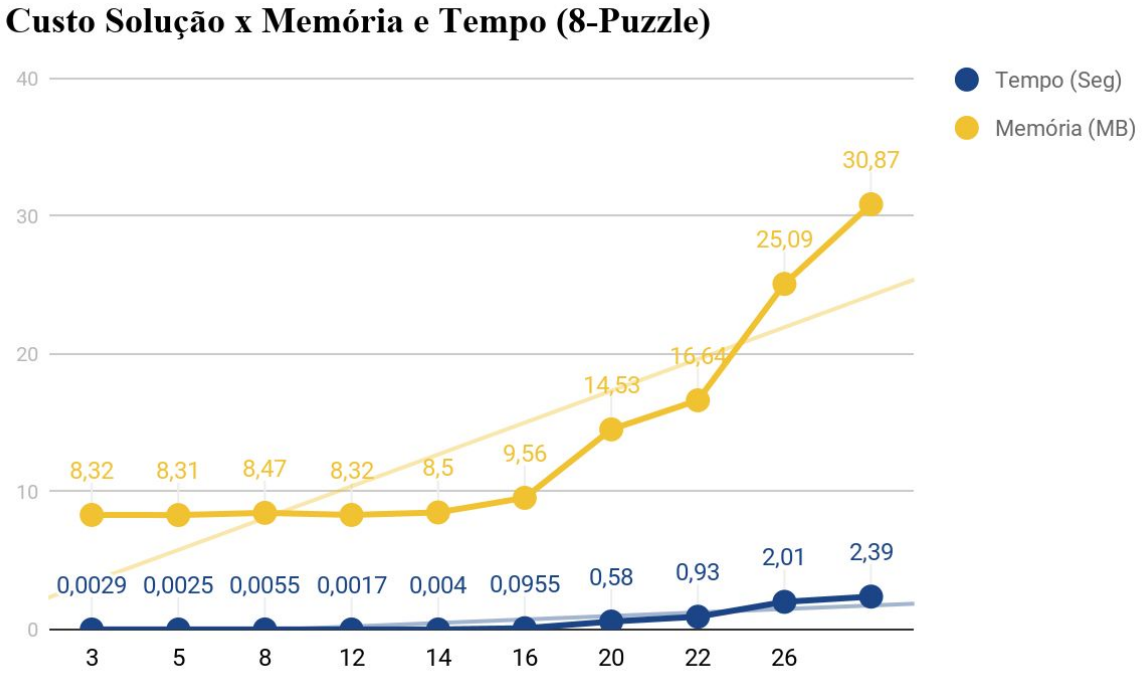

Figura 3. Gráfico de comparação de tempo e memória do 8-Puzzle

Complexidade de Memória

Quando se trata do consumo de memória, é preciso lembrar que grande parte do consumo está relacionado ao armazenamento das possibilidades existentes. A cada iteração, as possibilidades geradas são armazenadas para serem consultadas posteriormente. O tamanho necessário para armazenar a árvore de possibilidades vai variar de acordo com a entrada e o tamanho do N-Puzzle. Quando tratamos do pior caso, consideramos que o algoritmo irá executar todas as possibilidades para só no final retornar a resposta. Assim, o algoritmo irá percorrer n! possibilidades.

No algoritmo, temos uma estrutura Set(), que guarda todas as possibilidades geradas. As possibilidades são salvas em formato de String: “0,1,2,3,4,5,6,7,8”. Considerando a String anterior, podemos afirmar que cada posição do Set() possui n bytes, onde n é igual a quantidade de posições do N-Puzzle. Exemplo, no 8-Puzzle n = 8. **Assim no pior caso de um N-Puzzle, temos O(n * n!) para a complexidade de memória.**

No caso do 8-Puzzle a memória não é um grande problema, pois é possível gerar todas as possibilidade e armazenar sem grandes problemas. Mas se aumentarmos o tamanho do Puzzle, pode ser praticamente inviável rodar a solução, pois precisará de uma grande capacidade de memória para seu pior caso. Se considerarmos por exemplo um Puzzle com tamanho 25, o pior caso torna a solução inviável pelas possibilidades existentes.

QUESTÕES EXTRAS

Questão extra (desafio): Utilize seu algoritmo para determinar o pior caso para 8-puzzle.

Observando o algoritmo desenvolvido, verificamos a existência de dois pontos de parada. O primeiro ponto é quando uma solução é encontrada, o outro ponto é quando não existe solução, dessa forma o algoritmo irá percorrer todas as possibilidades.

A não existência de solução é o pior caso do algoritmo, pois o algoritmo necessita gerar toda a árvore de possibilidades.

Se observarmos, o 8-puzzle possui 9! (362.880) permutações possíveis, dessa forma, o pior caso é quando o algoritmo precisa percorrer todas as possibilidades existentes que é 9!. Quando realiza-se movimentos no 8-Puzzle, é necessário seguir algumas regras básicas, assim, a partir de um estado é possível realizar no máximo 4 movimentos (CBED). Observando essa regra, conclui-se que a partir de um estado inicial, não é possível percorrer todas as possibilidades (9!). Para percorrer todas as possibilidades (9!), só seria possível, se em algum momento, as peças fossem removidas do tabuleiro e recolocadas em outras posições. Assim, conclui-se que dado um estado inicial existem (9!/2) possibilidades, ou seja 181.440 movimentos possíveis.

Em resumo, o pior caso para o 8-Puzzle é quando não existe solução, assim, o algoritmo desenvolvido irá percorrer toda a árvore de possibilidades (181.440), para no fim, identificar que a solução não existe.

Questão extra (desafio ++): Estenda seu algoritmo para solucionar de maneira ótima o 15-puzzle e determine seu pior caso.

O algoritmo proposto desde o início tenta solucionar o problema do N-Puzzle independente do tamanho do N. O algoritmo desenvolvido, já apresenta uma solução ótima para detectar o custo da solução do N-Puzzle. O conceito do BFS usado no algoritmo, já apresenta uma solução satisfatória que é suficiente para resolver o 15-Puzzle.

O pior caso do 15-Puzzle, é semelhante ao 8-Puzzle. O pior caso é quando não existe solução dado um estado inicial, assim, o algoritmo precisará gerar todas as possibilidades da árvore, para no fim descobrir que não existe solução. Aqui, o algoritmo mantém as mesmas heurísticas do início, seguindo a ordem (CBED) para movimentação. No pior caso, a execução pode demorar bastante tempo ou até mesmo nem chegar ao fim, pois será necessário uma grande disponibilidade de memória para armazenar todas as possibilidades.

Para executar o algoritmo, basta adicionar no arquivo [in], um input válido com 15 dígitos. Dessa forma, o algoritmo tentará encontrar a solução do 15-Puzzle ao invés do 8-Puzzle.