


Algoritmos	Referências	Responsável
Busca em Largura		Natam
Busca em Profundidade Iterativa		Mateus
Busca A* com heurística de quantidade de peças erradas		Gabriel
Busca A* com heurística de distância de Manhattan	<a href="https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2">https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2</a> <a href="https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html">https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html</a>  A* (A Star) Search Alg...	Gabriel
Busca Bidirecional com A*	<a href="https://www.geeksforgeeks.org/bidirectional-search/">https://www.geeksforgeeks.org/bidirectional-search/</a>	Gabriel

Linguagem: Python

A fazer:

1. Menu
  - a. Permitir escolher qual algoritmo de resolução
  - b. Definir tamanho do N-Puzzle
  - c. (opcional) Imprimir os relatórios de cada algoritmo
2. [Algoritmos](#)
3. Método de representação do resultado
  - a. Matrizes para representar o puzzle
  - b. Informar no fim da execução do algoritmo: quantidade de nós expandidos (nós aptos a serem expandidos), tempo de execução, quantidade de passos (quantidade de ações até a solução)
  - c. média de nós
4. [Relatório](#)
5. [Github](#)

## Funções

## Auxiliares

Entre as funções auxiliares, estão:

Move\_X: Responsável por mover o espaço vazio, representado por "X", pela matriz, de acordo com uma direção informada, tal movimento ocorre ao substituir a posição do X por a posição do outro valor.

Ex: Para mover o X para a esquerda, substituímos sua posição pela do elemento na posição  $[x][y-1]$ , e para cima, substituímos pelo elemento  $[x - \text{número de linhas}][y]$

Soluvel: Confere se o estado inicial gerado pela função grid é capaz de alcançar o estado objetivo, faz isso através da contagem das inversões, que incrementa a cada vez que um valor da lista estado inicial é maior que seus sucessores, excluindo o elemento "X".

Caso o valor de n (proporção da matriz) seja impar, o valor da variável inversões deverá ser um número par, caso seja par, o valor, somado ao índice da posição de X deverá ser par também.

Grid: Cria um vetor que vai de 1 até  $n - 1$  inteiros e insere o "X" no final da lista, o estado objetivo passa então a ser uma cópia desta lista, formatada como matriz  $N \times N$ .

O valor inicial é feito a partir do embaralhamento dos elementos, após serem embaralhados com a função `random.shuffle()`, é chamada a função "Solve" para conferir se é um estado válido ou não, caso não seja, o estado é embaralhado novamente até gerar uma configuração válida.

Além dessas funções, no Auxiliares, é definido também os métodos e atributos do objeto Node, composto por:

no\_pai -> Para armazenar qual seu antecessor

position -> Guarda sua configuração atual no algoritmo

g -> Quantidade de movimentos feitos até aquele nó

h -> Valor heurístico

f -> Soma do h e g

Além de um método que define como o objeto deverá ser exibido quando chamado e outro que define como calcular qual objeto é menor em uma comparação entre dois nós.

## BFS

Este é um solucionador para o problema do n-puzzle. Abaixo está uma explicação passo a passo do funcionamento das funções do código:

### 1. **generate\_grid(n):**

Esta função cria uma grade inicial de tamanho "n" por "n" com números de 1 a  $n^2 - 1$ , mais o espaço vazio representado por 'X'. A grade é então embaralhada para criar uma configuração inicial aleatória.

### 2. **create\_goal(n):**

Gera o estado objetivo do quebra-cabeça, sendo uma grade ordenada de 1 a  $n^2 - 1$  mais o espaço vazio 'X'.

### 3. **generate\_neighbors(state):**

Recebe um estado do quebra-cabeça e gera todos os estados alcançáveis a partir desse estado mediante movimentos válidos (direita, esquerda, cima, baixo). Por exemplo, se o

espaço vazio 'X' estiver em uma posição específica, os vizinhos são gerados movendo o 'X' nessa direção.

#### **4. build\_graph(initial\_state):**

Esta função constrói um grafo representando o espaço de estados do quebra-cabeça. Cada estado é um nó no grafo, e as arestas são os movimentos válidos que levam de um estado a outro.

#### **5. bfs(graph, start, goal):**

Implementa uma busca em largura (BFS) no grafo gerado. Começando pelo estado inicial, explora todos os estados possíveis em cada etapa até encontrar o estado objetivo.

O algoritmo mantém uma fila de estados a serem explorados. A cada passo, ele verifica se o estado atual é o objetivo. Se for, retorna o caminho até esse estado.

Se o objetivo não for alcançado, o algoritmo continua expandindo estados até que todos os estados sejam explorados ou o objetivo seja encontrado.

## **IDS(Busca em Profundidade Iterativa)**

### **1. Profundidade\_Iterativa(max\_Deep):**

Utilizando duas listas: "lista\_aberta" para os estados a serem explorados e "lista\_fechada" para os estados já explorados faz uma busca em profundidade com uma profundidade máxima que é max\_Deep, utilizando a distância do "no" raiz como profundidade dos estados.

Apenas expandindo e adicionando a "lista\_aberta" os estados com profundidade menores que a profundidade máxima assim ficando sem estados para explorar e falhando em encontrar o objetivo naquela iteração.

A max\_Deep começa em 0 e vai iterativamente sendo incrementada, cada iteração que o algoritmo falha em encontrar o estado objetivo, as listas são limpas e a função "Profundidade\_Iterativa(max\_Deep+1):" é chamada para tentar encontrar o estado objetivo com a próxima profundidade máxima.

# Algoritmos A\*

Os algoritmos A\* seguem o mesmo padrão estrutural:

1. Função cálculo da heurística utilizada
2. Função que monta e exhibe o caminho do nó inicial até o final
3. Função que implementa o algoritmo

No caso do Bidirecional, temos uma função extra, responsável por detectar uma intersecção nas listas fechadas, sinalizando assim que existe um caminho que conecta o nó que partiu do estado inicial ao nó que partiu do estado final.

Listas abertas e listas fechadas: Listas abertas são utilizadas para armazenar os nós que ainda vão ser explorados no algoritmo, enquanto as listas fechadas guardam aqueles que já foram explorados ou foram descartados através do cálculo da heurística. Ambas são transformadas em árvores através do uso da biblioteca heapq.

A\* Manhattan: Inicia um nó usando como posição o estado inicial criado no início do programa e o coloca em uma lista aberta, o nó atual (que será movido através da árvore) é definido no começo do loop como a primeira posição da lista, ou seja, nó de menor valor e adicionado à lista fechada.

O nó atual é explorado, conferindo para quais posições pode se mover, se são válidas e se não são repetidas, os nós que ele gera após esse processo são adicionados à lista aberta. Sua condição de parada é: caso a posição do nó atual seja a configuração do objetivo, neste caso ele para o loop e imprime o relatório.

Sua heurística é calculada analisando cada posição do estado atual do nó e a comparando com a configuração objetivo, caso o elemento analisado não esteja na coordenada correta, é calculada a distancia até tal coordenada, isso é repetido para todos os elementos. A soma dos valores representa o h.

O caminho do nó inicial até o objetivo é construído através do no\_pai do nó atual. A posição do nó é adicionada a uma lista e o nó passa a receber a posição de seu pai, isso se repete até alcançar a primeira posição.

A\* Wrong (Misplaced titles): Funciona de maneira e estrutura igual à do Manhattan, porém seu cálculo heurístico foca na quantidade de peças fora do lugar, invés da distancia atual entre a peça analisada e sua posição correta.

Bidirecional: Usando a mesma heurística e lógica do Manhattan porém duas vezes, uma para o nó que sai da posição inicial e outra para o que parte da posição final, a condição de parada desse algoritmo é diferente, pois a função intersecção busca

um ponto comum nas listas fechadas dos nós, caso encontre o caminho começa a ser gerado, utilizando a mesma lógica dos últimos algoritmos e juntando o caminho percorrido pelo nó inicial com o final.

## Relatório

1. Quantidade de passos = tamanho da lista fechada
2. Tempo gasto = cronometrado utilizando a biblioteca time
3. Quantidade máxima de memória = biblioteca tracemalloc monitora qual linha de código utilizou mais memória durante a execução, informando valor e quantas vezes foi chamada.
4. Quantidade de nós expandidos = contador que incrementa a cada remoção da lista aberta.
5. Ramificação média = soma da quantidade de nós expandidos com a quantidade de iterações, divide o resultado por 2.