A-STAR - 8 PUZZLE

In [1]:
```
# Sessão de bibliotecas
import math
import copy
```

Função A *

- Função do Algoritmo A *O algoritmo A* é uma das maneiras mais conhecidas de busca pela melhor escolha. Cada novo nodo n é avaliado de acordo com o custo para alcançar o mesmo a partir da raiz da árvore de busca (g(n)) e e uma heurística que estima o custo para ir de n até o nodo objetivo (h(n)). Ou seja, sendo f(n) a função que avalia n, procuramos o mínimo de: f(n) = g(n) + h(n)

Uma das propriedades que A* possui é que ele garante encontrar sempre a melhor solução.

In [2]:
```
class aEstrela:
    def __init__(self, estadoInicial=None):
# setando variáveis do algoritmo, estado receberá a entrada inicial e demais itens são
        self.estado = estadoInicial
        self.h = 0
        self.g = 0
        self.f = 0
        self.pai = None
        self.acao = None
# definindo próximo estado
    def next(self, estado):
    # vetor que armazena movimentos aceitos
        movimentoAceito = []
    # laço para percorrer tabuleiro estado
        for i in range(0, 3):
            for j in range(0, 3):
                if estado[i][j] == 0:
                    r, c = i, j
   # Condicionais para possibilidades do tabuleiro utilizando eixos X e Y
    # MOVIMENTAR PARA CIMA
        if r > 0:
            node = copy.deepcopy(estado)
            r1 = r - 1
            node[r][c] = node[r1][c]
            node[r1][c] = 0
            movimentoAceito.append((node, 'CIMA'))
    # MOVIMENTAR PARA ESQUERDA
        if c > 0:
            node = copy.deepcopy(estado)
            c1 = c - 1
            node[r][c] = node[r][c1]
            node[r][c1] = 0
            movimentoAceito.append((node, 'ESQUERDA'))
    # MOVIMENTAR PARA BAIXO
        if r < 2:
            node = copy.deepcopy(estado)
            r1 = r + 1
            node[r][c] = node[r1][c]
            node[r1][c] = 0
```

```
            movimentoAceito.append((node, 'BAIXO'))
    # MOVIMENTAR PARA DIREITA
        if c < 2:
            node = copy.deepcopy(estado)
            c1 = c + 1
            node[r][c] = node[r][c1]
            node[r][c1] = 0
            movimentoAceito.append((node, 'DIREITA'))
     # retorna movimenta aceito
        return movimentoAceito
```

Função Custo / Caminho

- Função para calcular o custo das operações realizadas para encontrar a solução

In [3]:
```
    def caminho(self, node):
     # vetores para armazenar operações realizads (caminho, movimentos e custo)
        movimentos = []
        caminho = []
        custoCaminho = node.g
     # enquanto um nó estiver sendo analisado, verificar possibilidades e inserir na lis
     # append -> componente de lista
        while node:
            caminho.append(node.estado)
            movimentos.append(node.mover)
            node = node.nodePai
        movimentos.remove(None)
     # imprimindo o caminho solução
        print('O caminho solução é: ')
     # percorrendo o caminho de forma reversa através dos nós para imprimir as operações
        for node in reversed(caminho):
            imprimirEstado(node)
        print('As operações performadas foram: ')
     # salvando movimentos
        movimentos = reversed(movimentos)
     # .join realizando une itens de uma tupla em uma string
        sequenciaMovimentos = ", ".join(movimentos)
        print(sequenciaMovimentos)
        print('O custo do caminho é: ', custoCaminho)
```

Função Solução

- Função solução utilizando as heurísticas de Manhattan e de peças faltantes

In [4]:
```
def solucao(self, inicial, objetivo, func='Manhattan'):
    # setando variáveis iniciais
        contadorNodes = 0
        contadorExpandidos = 0
        margemManhattan = []
        expandido = []
    # verificando se o estado é igual ao estado objetivo
    # se sim, imprime que há uma solução e seus respectivos contadores
        if inicial.estado == objetivo.estado:
            print("Show! Solução encontrada.")
            self.caminho(inicial)
            print("Contador de nós gerados: ", contadorNodes)
            print("Contador de nós expandidos: ", contadorExpandidos)
```

```
            return
        # se a heurística da função recebida via parâmtro for peças faltantes
            if func == 'misplacedTiles':
            # chama heurística de peças faltantes
                inicial.h = h1_Misplaced(inicial.estado, objetivo.estado)
            # senão, utiliza heurística de manhattana
            else:
                inicial.h = h2_Manhattan(inicial.estado, objetivo.estado)
            # os estados são somados
            inicial.f = inicial.g + inicial.h
            # inicializando nós e movimento zerados
            inicial.nodePai = None
            inicial.mover = None
            # criando lista para a heurística
            margemManhattan.append(inicial)

            # utilizando lógica de manhattan
            while margemManhattan:
                pilha = margemManhattan.pop(0)
                vizinhanca = self.next(pilha.estado)
                expandido.append(pilha)
                contadorExpandidos += 1
                for vizinho in vizinhanca:
                    filho = aEstrela()
                    filho.estado = vizinho[0]
                    filho.mover = vizinho[1]
                    filho.g = pilha.g + 1
                # verificando novamente chamada da função
                    if func == 'misplacedTiles':
                        filho.h = h1_Misplaced(filho.estado, objetivo.estado)
                    else:
                        filho.h = h2_Manhattan(filho.estado, objetivo.estado)
                    filho.f = filho.g + filho.h
                    filho.nodePai = pilha
                    contadorNodes += 1

                    if (filho.estado == objetivo.estado):
                        print("Show! Solução encontrada.")
                        self.caminho(filho)
                        print("Contador de nós gerados: ", contadorNodes)
                        print("Contador de nós expandidos: ", contadorExpandidos)
                        return
                    # verificando se algum nó foi expandido
                    isexpandido = False
                    try:
                        expandido.index(filho.estado, )
                    except ValueError:
                        isexpandido = False
                    # se não foi expandido
                    if not isexpandido:
                        encontrado = False
                        k = 0
                        for item in margemManhattan:
                            if item.estado == filho.estado:
                                encontrado = True
                                if filho.f < item.f:
                                    item.f = filho.f
                                    margemManhattan[k] = item
                                    break
                            k += 1
                        # se não foi encontrado novo movimento
```

```
                    if not encontrado:
                        margemManhattan.append(filho)
                # lambda cria uma função 'inline' para evitar uso duplicado da função
                margemManhattan = sorted(margemManhattan, key=lambda x: x.f)
        print('Eita! Sem solução.')
        return
```

Heurística H1 - Peças Faltantes

- Função de heurística para peças faltantes

In [5]:
```
def h1_Misplaced(estado1, estado2):
    h = 0
    for i in range(0, 3, 1):
        for j in range(0, 3, 1):
            if estado1[i][j] != estado2[i][j] and estado1[i][j] != 0:
                h += 1
    return h
```

Heurística H2 - Manhattan

- Função de heurística para distância de Manhattan

In [6]:
```
def h2_Manhattan(estado1, estado2):
    vetorManhattan = []
    distanciaManhattan = 0
    for i in range(0, 3):
        for j in range(0, 3):
            vetorManhattan.append(estado2[i][j])

    for i in range(0, 3, 1):
        for j in range(0, 3, 1):
            pilhaent_ij = estado1[i][j]
            i_pilhaent = i
            j_pilhaent = j
            index = vetorManhattan.index(pilhaent_ij)
            i_objetivo, j_objetivo = index // 3, index % 3
            if pilhaent_ij != 0:
                distanciaManhattan += (math.fabs(i_objetivo - i_pilhaent) + math.fabs(j
    return distanciaManhattan
```

Função de Imprimir

- Função para imprimir o tabuleiro

In [7]:
```
def imprimirEstado(estado):
    for i in range(3):
        resultado = ""
        for j in range(3):
            resultado += str(estado[i][j]) + " "
        print(resultado)
    print("")
```

Função de Entrada

- Função para receber entradas de ESTADO INICIAL e FINAL para o algoritmo

In [8]:
```
def entradaJogo():
    print("Insira o Quebra-Cabeças INICIAL:")
    # criando vetores e realizando o split de elementos através do separador
    # espaço designa o separados entre itens
    # 0 => peça faltante
    vetorEntrada = []
    vetorObjetivo = []
    elemento = input().split(" ")
    k = 0
    try:
        for i in range(0, 3):
            vetorEntrada += [0]
        for i in range(0, 3):
            vetorEntrada[i] = [0] * 3
        for i in range(0, 3):
            for j in range(0, 3):
                vetorEntrada[i][j] = int(elemento[k])
                k += 1
    except (ValueError, IndexError):
        print("Por favor, separe os itens com espaços :D")
        return [], []
    print("Insira o Quebra-Cabeças OBJETIVO:")
    elemento = input().split(" ")
    k = 0
    try:
        for i in range(0, 3):
            vetorObjetivo += [0]
        for i in range(0, 3):
            vetorObjetivo[i] = [0] * 3
        for i in range(0, 3):
            for j in range(0, 3):
                vetorObjetivo[i][j] = int(elemento[k])
                k += 1
    except (ValueError, IndexError):
        print("Insira os valores (utiliza espaço como separador):")
        return vetorEntrada, []
    return vetorEntrada, vetorObjetivo
```

Função Principal

- Função Main para chamadas do algoritmo

In [9]:
```
def main():
    vetorEntrada, vetorObjetivo = entradaJogo()
    if vetorEntrada and vetorObjetivo:
        print('o Estado INICIAL é:')
        imprimirEstado(vetorEntrada)
        print('o Estado OBJETIVO é')
        imprimirEstado(vetorObjetivo)
        inicial = aEstrela(vetorEntrada)
        objetivo = aEstrela(vetorObjetivo)
        print("Solução utilizando heurística Manhattan:")
        inicial.solucao(inicial, objetivo)
        print("\n Solução utilizando heurística MisplacedTiles:")
        inicial.solucao(inicial, objetivo, 'misplacedTiles')
```

Entradas JOGO

- PROJETO: 0 1 2 3 4 5 6 7 8 -> ENTRADA | 1 2 3 4 5 6 7 8 0 -> OBJETIVO

SCRIPT ALGORITMO

In [11]:
```
import math
import copy

class aEstrela:
    def __init__(self, estadoInicial=None):
        self.estado = estadoInicial
        self.h = 0
        self.g = 0
        self.f = 0
        self.nodePai = None
        self.mover = None
    def next(self, estado):
        movimentoAceito = []
        for i in range(0, 3):
            for j in range(0, 3):
                if estado[i][j] == 0:
                    r, c = i, j

        if r > 0:
            node = copy.deepcopy(estado)
            r1 = r - 1
            node[r][c] = node[r1][c]
            node[r1][c] = 0
            movimentoAceito.append((node, 'CIMA'))
        if c > 0:
            node = copy.deepcopy(estado)
            c1 = c - 1
            node[r][c] = node[r][c1]
            node[r][c1] = 0
            movimentoAceito.append((node, 'ESQUERDA'))
        if r < 2:
            node = copy.deepcopy(estado)
            r1 = r + 1
            node[r][c] = node[r1][c]
            node[r1][c] = 0
            movimentoAceito.append((node, 'BAIXO'))
        if c < 2:
            node = copy.deepcopy(estado)
            c1 = c + 1
            node[r][c] = node[r][c1]
            node[r][c1] = 0
            movimentoAceito.append((node, 'DIREITA'))
        return movimentoAceito

    def caminho(self, node):
        movimentos = []
        caminho = []
        custoCaminho = node.g
        while node:
            caminho.append(node.estado)
            movimentos.append(node.mover)
            node = node.nodePai
        movimentos.remove(None)
```

```
        print('o CAMINHO performado foi:')
        for node in reversed(caminho):
            imprimirEstado(node)
        print('as OPERAÇÕES performadas foram: ')
        movimentos = reversed(movimentos)
        sequenciaMovimentos = ", ".join(movimentos)
        print(sequenciaMovimentos)
        print('o CUSTO do CAMINHO é: ', custoCaminho)

    def solve(self, inicial, objetivo, func='Manhattan'):
        contadorNodes = 0
        contadorExpandidos = 0
        margemManhattan = []
        expandido = []
        if inicial.estado == objetivo.estado:
            print("SHOW! Solução encontrada")
            self.caminho(inicial)
            print("[MÉTRICAS] Contador de nós gerados:: ", contadorNodes)
            print("[MÉTRICAS] Contador de nós expandidos: ", contadorExpandidos)
            return
        if func == 'misplacedTiles':
            inicial.h = h1_Misplaced(inicial.estado, objetivo.estado)
        else:
            inicial.h = h2_Manhattan(inicial.estado, objetivo.estado)
        inicial.f = inicial.g + inicial.h
        inicial.nodePai = None
        inicial.mover = None
        margemManhattan.append(inicial)
        while margemManhattan:
            pilha = margemManhattan.pop(0)
            vizinhanca = self.next(pilha.estado)
            expandido.append(pilha)
            contadorExpandidos += 1
            for vizinho in vizinhanca:
                filho = aEstrela()
                filho.estado = vizinho[0]
                filho.mover = vizinho[1]
                filho.g = pilha.g + 1
                if func == 'misplacedTiles':
                    filho.h = h1_Misplaced(filho.estado, objetivo.estado)
                else:
                    filho.h = h2_Manhattan(filho.estado, objetivo.estado)
                filho.f = filho.g + filho.h
                filho.nodePai = pilha
                contadorNodes += 1
                if (filho.estado == objetivo.estado):
                    print("SHOW! Solução encontrada")
                    self.caminho(filho)
                    print("[MÉTRICAS] Contador de nós gerados:", contadorNodes)
                    print("[MÉTRICAS] Contador de nós expandidos:", contadorExpandidos)
                    return

                isexpandido = False
                try:
                    expandido.index(filho.estado, )
                except ValueError:
                    isexpandido = False

                if not isexpandido:
                    encontrado = False
                    k = 0
```

```
                    for item in margemManhattan:
                        if item.estado == filho.estado:
                            encontrado = True
                            if filho.f < item.f:
                                item.f = filho.f
                                margemManhattan[k] = item
                                break
                        k += 1

                    if not encontrado:
                        margemManhattan.append(filho)
                margemManhattan = sorted(margemManhattan, key=lambda x: x.f)
        print('EITA! Sem solução')
        return

def h2_Manhattan(estado1, estado2):
    vetorManhattan = []
    distanciaManhattan = 0
    for i in range(0, 3):
        for j in range(0, 3):
            vetorManhattan.append(estado2[i][j])

    for i in range(0, 3, 1):
        for j in range(0, 3, 1):
            pilhaent_ij = estado1[i][j]
            i_pilhaent = i
            j_pilhaent = j
            index = vetorManhattan.index(pilhaent_ij)
            i_objetivo, j_objetivo = index // 3, index % 3
            if pilhaent_ij != 0:
                distanciaManhattan += (math.fabs(i_objetivo - i_pilhaent) + math.fabs(j
    return distanciaManhattan

def h1_Misplaced(estado1, estado2):
    h = 0
    for i in range(0, 3, 1):
        for j in range(0, 3, 1):
            if estado1[i][j] != estado2[i][j] and estado1[i][j] != 0:
                h += 1
    return h

def imprimirEstado(estado):
    for i in range(3):
        resultado = ""
        for j in range(3):
            resultado += str(estado[i][j]) + " "
        print(resultado)
    print("")

def userInput():
    print("insira o ESTADO INICIAL:")
    vetorEntrada = []
    vetorObjetivo = []
    elemento = input().split(" ")
    k = 0
    try:
        for i in range(0, 3):
            vetorEntrada += [0]
        for i in range(0, 3):
            vetorEntrada[i] = [0] * 3
        for i in range(0, 3):
```

```
            for j in range(0, 3):
                vetorEntrada[i][j] = int(elemento[k])
                k += 1
    except (ValueError, IndexError):
        print("EI! Utilize espaço para separar os valores")
        return [], []
    print("insira o ESTADO OBJETIVO:")
    elemento = input().split(" ")
    k = 0
    try:
        for i in range(0, 3):
            vetorObjetivo += [0]
        for i in range(0, 3):
            vetorObjetivo[i] = [0] * 3
        for i in range(0, 3):
            for j in range(0, 3):
                vetorObjetivo[i][j] = int(elemento[k])
                k += 1
    except (ValueError, IndexError):
        print("EI! Utilize espaço para separar os valores")
        return vetorEntrada, []
    return vetorEntrada, vetorObjetivo

def main():
    vetorEntrada, vetorObjetivo = userInput()
    if vetorEntrada and vetorObjetivo:
        print('ESTADO INICIAL:')
        imprimirEstado(vetorEntrada)
        print('ESTADO OBJETIVO:')
        imprimirEstado(vetorObjetivo)
        inicial = aEstrela(vetorEntrada)
        objetivo = aEstrela(vetorObjetivo)
        print("A * utilizando heurística de DISTÂNCIA DE MANHATTAN:")
        inicial.solve(inicial, objetivo)
        print("\nA * utilizando heurística de PEÇAS FALTANTES:")
        inicial.solve(inicial, objetivo, 'misplacedTiles')

main()
```

```
insira o ESTADO INICIAL:
1 5 4 3 7 2 6 8 0
insira o ESTADO OBJETIVO:
1 2 3 4 5 6 7 8 0
ESTADO INICIAL:
1 5 4
3 7 2
6 8 0

ESTADO OBJETIVO:
1 2 3
4 5 6
7 8 0

A * utilizando heurística de DISTÂNCIA DE MANHATTAN:
SHOW! Solução encontrada
o CAMINHO performado foi:
1 5 4
3 7 2
6 8 0

1 5 4
3 7 2
6 0 8
```

```
1 5 4
3 0 2
6 7 8

1 5 4
0 3 2
6 7 8

1 5 4
6 3 2
0 7 8

1 5 4
6 3 2
7 0 8

1 5 4
6 0 2
7 3 8

1 5 4
0 6 2
7 3 8

0 5 4
1 6 2
7 3 8

5 0 4
1 6 2
7 3 8

5 4 0
1 6 2
7 3 8

5 4 2
1 6 0
7 3 8

5 4 2
1 0 6
7 3 8

5 0 2
1 4 6
7 3 8

0 5 2
1 4 6
7 3 8

1 5 2
0 4 6
7 3 8

1 5 2
4 0 6
7 3 8

1 5 2
4 3 6
7 0 8
```

```
1 5 2
4 3 6
7 8 0

1 5 2
4 3 0
7 8 6

1 5 2
4 0 3
7 8 6

1 0 2
4 5 3
7 8 6

1 2 0
4 5 3
7 8 6

1 2 3
4 5 0
7 8 6

1 2 3
4 5 6
7 8 0

as OPERAÇÕES performadas foram:
ESQUERDA, CIMA, ESQUERDA, BAIXO, DIREITA, CIMA, ESQUERDA, CIMA, DIREITA, DIREITA, BAIXO,
ESQUERDA, CIMA, ESQUERDA, BAIXO, DIREITA, BAIXO, DIREITA, CIMA, ESQUERDA, CIMA, DIREITA,
BAIXO, BAIXO
o CUSTO do CAMINHO é:  24
[MÉTRICAS] Contador de nós gerados: 7175
[MÉTRICAS] Contador de nós expandidos: 2691

A * utilizando heurística de PEÇAS FALTANTES:
SHOW! Solução encontrada
o CAMINHO performado foi:
1 5 4
3 7 2
6 8 0

1 5 4
3 7 2
6 0 8

1 5 4
3 0 2
6 7 8

1 5 4
0 3 2
6 7 8

1 5 4
6 3 2
0 7 8

1 5 4
6 3 2
7 0 8

1 5 4
6 0 2
```

```
7 3 8

1 5 4
0 6 2
7 3 8

0 5 4
1 6 2
7 3 8

5 0 4
1 6 2
7 3 8

5 4 0
1 6 2
7 3 8

5 4 2
1 6 0
7 3 8

5 4 2
1 0 6
7 3 8

5 0 2
1 4 6
7 3 8

0 5 2
1 4 6
7 3 8

1 5 2
0 4 6
7 3 8

1 5 2
4 0 6
7 3 8

1 5 2
4 3 6
7 0 8

1 5 2
4 3 6
7 8 0

1 5 2
4 3 0
7 8 6

1 5 2
4 0 3
7 8 6

1 0 2
4 5 3
7 8 6

1 2 0
4 5 3
7 8 6
```

```
1 2 3
4 5 0
7 8 6

1 2 3
4 5 6
7 8 0

as OPERAÇÕES performadas foram:
ESQUERDA, CIMA, ESQUERDA, BAIXO, DIREITA, CIMA, ESQUERDA, CIMA, DIREITA, DIREITA, BAIXO,
ESQUERDA, CIMA, ESQUERDA, BAIXO, DIREITA, BAIXO, DIREITA, CIMA, ESQUERDA, CIMA, DIREITA,
BAIXO, BAIXO
o CUSTO do CAMINHO é:  24
[MÉTRICAS] Contador de nós gerados: 84416
[MÉTRICAS] Contador de nós expandidos: 31086
```

In [ ]: