A \* Algorithm

Find the path from A-G for the given graph by using A\* algorithm.

Code:

import copy

from heapq import heappush, heappop

n = 3

row = [ 1, 0, -1, 0 ]

col = [ 0, -1, 0, 1 ]

class priorityQueue:

    def \_\_init\_\_(self):

        self.heap = []

    def push(self, k):

        heappush(self.heap, k)

    def pop(self):

        return heappop(self.heap)

    def empty(self):

        if not self.heap:

            return True

        else:

            return False

class node:

    def \_\_init\_\_(self, parent, mat, empty\_tile\_pos,

                 cost, level):

        self.parent = parent

        self.mat = mat

        self.empty\_tile\_pos = empty\_tile\_pos

        self.cost = cost

        self.level = level

    def \_\_lt\_\_(self, nxt):

        return self.cost < nxt.cost

def calculateCost(mat, final) -> int:

    count = 0

    for i in range(n):

        for j in range(n):

            if ((mat[i][j]) and

                (mat[i][j] != final[i][j])):

                count += 1

    return count

def newNode(mat, empty\_tile\_pos, new\_empty\_tile\_pos,

            level, parent, final) -> node:

    new\_mat = copy.deepcopy(mat)

    x1 = empty\_tile\_pos[0]

    y1 = empty\_tile\_pos[1]

    x2 = new\_empty\_tile\_pos[0]

    y2 = new\_empty\_tile\_pos[1]

    new\_mat[x1][y1], new\_mat[x2][y2] = new\_mat[x2][y2], new\_mat[x1][y1]

    cost = calculateCost(new\_mat, final)

    new\_node = node(parent, new\_mat, new\_empty\_tile\_pos,

                    cost, level)

    return new\_node

def printMatrix(mat):

    for i in range(n):

        for j in range(n):

            print("%d " % (mat[i][j]), end = " ")

        print()

def isSafe(x, y):

    return x >= 0 and x < n and y >= 0 and y < n

def printPath(root):

    if root == None:

        return

    printPath(root.parent)

    printMatrix(root.mat)

    print()

def solve(initial, empty\_tile\_pos, final):

q = priorityQueue()

    cost = calculateCost(initial, final)

    root = node(None, initial,

                empty\_tile\_pos, cost, 0)

    pq.push(root)

    while not pq.empty():

        minimum = pq.pop()

        if minimum.cost == 0:

            printPath(minimum)

            return

        for i in range(4):

            new\_tile\_pos = [

                minimum.empty\_tile\_pos[0] + row[i],

                minimum.empty\_tile\_pos[1] + col[i], ]

            if isSafe(new\_tile\_pos[0], new\_tile\_pos[1]):

                child = newNode(minimum.mat,

                                minimum.empty\_tile\_pos,

                                new\_tile\_pos,

                                minimum.level + 1,

                                minimum, final,)

                pq.push(child)

initial = [ [ 2, 8, 3 ],

            [ 1, 6, 4 ],

            [ 7, 0, 5 ] ]

final = [ [ 1, 2, 3 ],

          [ 8, 0, 4 ],

          [ 7, 6, 5 ] ]

empty\_tile\_pos = [ 2, 1 ]

solve(initial, empty\_tile\_pos, final)

Output:

