# Python code to display the way from the root

# node to the final destination node for N\*N-1 puzzle

# algorithm by the help of Branch and Bound technique

# The answer assumes that the instance of the

# puzzle can be solved

# Importing the 'copy' for deepcopy method

import copy

# Importing the heap methods from the python

# library for the Priority Queue

from heapq import heappush, heappop

# This particular var can be changed to transform

# the program from 8 puzzle(n=3) into 15

# puzzle(n=4) and so on ...

n = 3

# bottom, left, top, right

rows = [ 1, 0, -1, 0 ]

cols = [ 0, -1, 0, 1 ]

# creating a class for the Priority Queue

class priorityQueue:

# Constructor for initializing a

# Priority Queue

def \_\_init\_\_(self):

self.heap = []

# Inserting a new key 'key'

def push(self, key):

heappush(self.heap, key)

# funct to remove the element that is minimum,

# from the Priority Queue

def pop(self):

return heappop(self.heap)

# funct to check if the Queue is empty or not

def empty(self):

if not self.heap:

return True

else:

return False

# structure of the node

class nodes:

def \_\_init\_\_(self, parent, mats, empty\_tile\_posi,

costs, levels):

# This will store the parent node to the

# current node And helps in tracing the

# path when the solution is visible

self.parent = parent

# Useful for Storing the matrix

self.mats = mats

# useful for Storing the position where the

# empty space tile is already existing in the matrix

self.empty\_tile\_posi = empty\_tile\_posi

# Store no. of misplaced tiles

self.costs = costs

# Store no. of moves so far

self.levels = levels

# This func is used in order to form the

# priority queue based on

# the costs var of objects

def \_\_lt\_\_(self, nxt):

return self.costs < nxt.costs

# method to calc. the no. of

# misplaced tiles, that is the no. of non-blank

# tiles not in their final posi

def calculateCosts(mats, final) -> int:

count = 0

for i in range(n):

for j in range(n):

if ((mats[i][j]) and

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

count += 1

return count

def newNodes(mats, empty\_tile\_posi, new\_empty\_tile\_posi,

levels, parent, final) -> nodes:

# Copying data from the parent matrixes to the present matrixes

new\_mats = copy.deepcopy(mats)

# Moving the tile by 1 position

x1 = empty\_tile\_posi[0]

y1 = empty\_tile\_posi[1]

x2 = new\_empty\_tile\_posi[0]

y2 = new\_empty\_tile\_posi[1]

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

# Setting the no. of misplaced tiles

costs = calculateCosts(new\_mats, final)

new\_nodes = nodes(parent, new\_mats, new\_empty\_tile\_posi,

costs, levels)

return new\_nodes

# func to print the N by N matrix

def printMatsrix(mats):

for i in range(n):

for j in range(n):

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

print()

# func to know if (x, y) is a valid or invalid

# matrix coordinates

def isSafe(x, y):

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

# Printing the path from the root node to the final node

def printPath(root):

if root == None:

return

printPath(root.parent)

printMatsrix(root.mats)

print()

# method for solving N\*N - 1 puzzle algo

# by utilizing the Branch and Bound technique. empty\_tile\_posi is

# the blank tile position initially.

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

# Creating a priority queue for storing the live

# nodes of the search tree

pq = priorityQueue()

# Creating the root node

costs = calculateCosts(initial, final)

root = nodes(None, initial,

empty\_tile\_posi, costs, 0)

# Adding root to the list of live nodes

pq.push(root)

# Discovering a live node with min. costs,

# and adding its children to the list of live

# nodes and finally deleting it from

# the list.

while not pq.empty():

# Finding a live node with min. estimatsed

# costs and deleting it form the list of the

# live nodes

minimum = pq.pop()

# If the min. is ans node

if minimum.costs == 0:

# Printing the path from the root to

# destination;

printPath(minimum)

return

# Generating all feasible children

for i in range(n):

new\_tile\_posi = [

minimum.empty\_tile\_posi[0] + rows[i],

minimum.empty\_tile\_posi[1] + cols[i], ]

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

# Creating a child node

child = newNodes(minimum.mats,

minimum.empty\_tile\_posi,

new\_tile\_posi,

minimum.levels + 1,

minimum, final,)

# Adding the child to the list of live nodes

pq.push(child)

# Main Code

# Initial configuration

# Value 0 is taken here as an empty space

initial = [ [ 1, 2, 3 ],

[ 5, 6, 0 ],

[ 7, 8, 4 ] ]

# Final configuration that can be solved

# Value 0 is taken as an empty space

final = [ [ 1, 2, 3 ],

[ 5, 8, 6 ],

[ 0, 7, 4 ] ]

# Blank tile coordinates in the

# initial configuration

empty\_tile\_posi = [ 1, 2 ]

# Method call for solving the puzzle

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