In [2]:

```
# Python3 program to print the path from root
# node to destination node for N*N-1 puzzle
# algorithm using Branch and Bound
# The solution assumes that instance of
# puzzle is solvable

# Importing copy for deepcopy function
import copy

# Importing the heap functions from python
# library for Priority Queue
from heapq import heappush, heappop

# This variable can be changed to change
# the program from 8 puzzle(n=3) to 15
# puzzle(n=4) to 24 puzzle(n=5)...
n = 3

# bottom, left, top, right
row = [ 1, 0, -1, 0 ]
col = [ 0, -1, 0, 1 ]

# A class for Priority Queue
class priorityQueue:

    # Constructor to initialize a
    # Priority Queue
    def __init__(self):
        self.heap = []

    # Inserts a new key 'k'
    def push(self, k):
        heappush(self.heap, k)

    # Method to remove minimum element
    # from Priority Queue
    def pop(self):
        return heappop(self.heap)

    # Method to know if the Queue is empty
    def empty(self):
        if not self.heap:
            return True
        else:
            return False

# Node structure
class node:

    def __init__(self, parent, mat, empty_tile_pos,
                 cost, level):

        # Stores the parent node of the
        # current node helps in tracing
        # path when the answer is found
        self.parent = parent

        # Stores the matrix
        self.mat = mat
```

```
        # Stores the position at which the
        # empty space tile exists in the matrix
        self.empty_tile_pos = empty_tile_pos

        # Storesthe number of misplaced tiles
        self.cost = cost

        # Stores the number of moves so far
        self.level = level

    # This method is defined so that the
    # priority queue is formed based on
    # the cost variable of the objects
    def __lt__(self, nxt):
        return self.cost < nxt.cost

# Function to calculate the number of
# misplaced tiles ie. number of non-blank
# tiles not in their goal position
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:

    # Copy data from parent matrix to current matrix
    new_mat = copy.deepcopy(mat)

    # Move tile by 1 position
    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]

    # Set number of misplaced tiles
    cost = calculateCost(new_mat, final)

    new_node = node(parent, new_mat, new_empty_tile_pos,
                    cost, level)
    return new_node

# Function to print the N x N matrix
def printMatrix(mat):

    for i in range(n):
        for j in range(n):
            print("%d " % (mat[i][j]), end = " ")

        print()

# Function to check if (x, y) is a valid
```

```
# matrix coordinate
def isSafe(x, y):

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

# Print path from root node to destination node
def printPath(root):

    if root == None:
        return

    printPath(root.parent)
    printMatrix(root.mat)
    print()

# Function to solve N*N - 1 puzzle algorithm
# using Branch and Bound. empty_tile_pos is
# the blank tile position in the initial state.
def solve(initial, empty_tile_pos, final):

    # Create a priority queue to store live
    # nodes of search tree
    pq = priorityQueue()

    # Create the root node
    cost = calculateCost(initial, final)
    root = node(None, initial,
                empty_tile_pos, cost, 0)

    # Add root to list of live nodes
    pq.push(root)

    # Finds a live node with least cost,
    # add its children to list of live
    # nodes and finally deletes it from
    # the list.
    while not pq.empty():

        # Find a live node with least estimated
        # cost and delete it form the list of
        # live nodes
        minimum = pq.pop()

        # If minimum is the answer node
        if minimum.cost == 0:

            # Print the path from root to
            # destination;
            printPath(minimum)
            return

        # Generate all possible children
        for i in range(n):
            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]):

                # Create a child node
                child = newNode(minimum.mat,
```

```
                                minimum.empty_tile_pos,
                                new_tile_pos,
                                minimum.level + 1,
                                minimum, final,)

                # Add child to list of live nodes
                pq.push(child)

# Driver Code

# Initial configuration
# Value 0 is used for empty space
initial = [ [ 1, 2, 3 ],
            [ 5, 6, 0 ],
            [ 7, 8, 4 ] ]

# Solvable Final configuration
# Value 0 is used for empty space
final = [ [ 1, 2, 3 ],
          [ 5, 8, 6 ],
          [ 0, 7, 4 ] ]

# Blank tile coordinates in
# initial configuration
empty_tile_pos = [ 1, 2 ]

# Function call to solve the puzzle
solve(initial, empty_tile_pos, final)

# This code is contributed by Kevin Joshi
```

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

1  2  3
5  0  6
7  8  4

1  2  3
5  8  6
7  0  4

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