```
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):
        return not self.heap

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, y1 = empty_tile_pos
    x2, y2 = new_empty_tile_pos
    new_mat[x1][y1], new_mat[x2][y2] = new_mat[x2][y2], new_mat[x1][y1]
    cost = calculateCost(new_mat, final)
    return node(parent, new_mat, new_empty_tile_pos, cost, level)

def printMatrix(mat):
    for i in range(n):
        for j in range(n):
            print(f"{mat[i][j]} ", end="")
        print()

def isSafe(x, y):
    return 0 <= x < n and 0 <= y < n

def printPath(root):
    if root is None:
        return
    printPath(root.parent)
    printMatrix(root.mat)
    print()

def solve(initial, empty_tile_pos, final):
    pq = 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 = [[1, 2, 3], [5, 6, 0], [7, 8, 4]]
final = [[1, 2, 3], [5, 8, 6], [0, 7, 4]]
empty_tile_pos = [1, 2]

solve(initial, empty_tile_pos, final)

print('Adithya Ravikeerthi')
print('DFS')
```

```
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

Adithya Ravikeerthi
DFS
```

```
class PuzzleIDS:
    def __init__(self, start, goal):
        self.start = start
        self.goal = goal

    def find_zero(self, state):
        for i in range(3):
            for j in range(3):
                if state[i][j] == 0:
                    return (i, j)

    def get_neighbors(self, state):
        x, y = self.find_zero(state)
        neighbors = []
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]

        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < 3 and 0 <= ny < 3:
                new_state = [row[:] for row in state]
                new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
                neighbors.append(new_state)
        return neighbors

    def dls(self, state, depth, path, visited):
        if state == self.goal:
            return path + [state]
        if depth == 0:
            return None

        visited.add(tuple(tuple(row) for row in state))

        for neighbor in self.get_neighbors(state):
            neighbor_tuple = tuple(tuple(row) for row in neighbor)
            if neighbor_tuple not in visited:
                result = self.dls(neighbor, depth - 1, path + [state], visited)
                if result:
                    return result
        visited.remove(tuple(tuple(row) for row in state))
        return None

    def ids(self, max_depth=50):
        for depth in range(max_depth):
            visited = set()
            result = self.dls(self.start, depth, [], visited)
            if result:
                return result
        return None

start_state = [[1, 2, 3], [4, 0, 5], [7, 8, 6]]
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

puzzle = PuzzleIDS(start_state, goal_state)
solution = puzzle.ids()
```

```
if solution:
    for step in solution:
        print(step)
else:
    print("No solution found.")

print('Adithya Ravikeerthi')
print('IDS')
```

```
[[1, 2, 3], [4, 0, 5], [7, 8, 6]]
[[1, 2, 3], [4, 5, 0], [7, 8, 6]]
[[1, 2, 3], [4, 5, 6], [7, 8, 0]]
Adithya Ravikeerthi
IDS
```