PROGRAM [2]:

```
import copy
from heapq import heappush, heappop
n = 3
rows = [1, 0, -1, 0]
cols = [0, -1, 0, 1]
class PriorityQueue:
    def __init__(self):
        self.heap = []
    def push(self, key):
        heappush(self.heap, key)
    def pop(self):
        return heappop(self.heap)
    def empty(self):
        return not bool(self.heap)
class Nodes:
    def __init__(self, parent, mats, empty_tile_posi, costs, levels):
        self.parent = parent
        self.mats = mats
        self.empty_tile_posi = empty_tile_posi
        self.costs = costs
        self.levels = levels
    def __lt__(self, nxt):
        return self.costs < nxt.costs
def calculate_costs(mats, final):
```

OUTPUT [2]:

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

Solved with Manhattan distance exploring 1 states
Solved with Manhattan least squares exploring 1 states
Solved with linear distance exploring 1 states
Solved with linear least squares exploring 1 states
```

```
    return sum(1 for i in range(n) for j in range(n) if mats[i][j] and mats[i][j] != final[i][j])
def new_nodes(mats, empty_tile_posi, new_empty_tile_posi, levels, parent, final):
    new_mats = copy.deepcopy(mats)
    x1, y1 = empty_tile_posi
    x2, y2 = new_empty_tile_posi
    new_mats[x1][y1], new_mats[x2][y2] = new_mats[x2][y2], new_mats[x1][y1]
    costs = calculate_costs(new_mats, final)
    return Nodes(parent, new_mats, new_empty_tile_posi, costs, levels)
def print_matrix(mats):
    for row in mats:
        print(*row)
    print()
def is_safe(x, y):
    return 0 <= x < n and 0 <= y < n
def print_path(root):
    if root is None:
        return
    print_path(root.parent)
    print_matrix(root.mats)
def solve(initial, empty_tile_posi, final):
    pq = PriorityQueue()
    costs = calculate_costs(initial, final)
    root = Nodes(None, initial, empty_tile_posi, costs, 0)
    pq.push(root)
    while not pq.empty():
        minimum = pq.pop()
```

```
    if minimum.costs == 0:
        print_path(minimum)
    return
    for i in range(n):
        new_tile_posi = [minimum.empty_tile_posi[0] + rows[i], minimum.empty_tile_posi[1] + cols[i]]
        if is_safe(*new_tile_posi):
            child = new_nodes(minimum.mats, minimum.empty_tile_posi, new_tile_posi, minimum.levels + 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_posi = [1, 2]
solve(initial, empty_tile_posi, final)
```