Q-2 Write a program to implement BFS 8 Puzzle problem

```
import numpy as np
import os

class Node :
    def __init__(self, node_no, data, parent, act,
                 cost ):
        self.data = data
        self.parent = parent
        self.act = act
        self.node_no = node_no
        self.cost = cost

def get_initial ():
    print (
        "Please enter number from 0-8, no
         number should be repeated or be out
         of this range"
    )
    initial_state = np.zeros (9)
    for i in range (9):
        states = int(input ("Enter the " + str (i+1) +
                            "number: "))
        if states < 0 or states > 8:
            print ("Please only enter states which
                    are [0-8], run code again")
            exit (0)
        else :
            initial_state[i] = np.array (states)
    return np.reshape (initial_state, (3, 3))
```

```
def find_index(puzzle):
    i, j = np.where(puzzle == 0)
    i = int(i)
    j = int(j)
    return i, j

def move_left(data):
    i, j = find_index(data)
    if j == 0:
        return None
    else:
        temp_arr = np.copy(data)
        temp = temp_arr[i, j-1]
        temp_arr[i, j] = temp
        temp_arr[i, j-1] = 0
        return temp_arr

def move_right(data):
    i, j = find_index(data)
    if j == 2:
        return None
    else:
        temp_arr = np.copy(data)
        temp = temp_arr[i, j+1]
        temp_arr[i, j] = temp
        temp_arr[i, j+1] = 0
        return temp_arr
```

```
def move_up(data):
    i, j = find_index(data)
    if i == 0:
        return None
    else:
        temp_arr = np.copy(data)
        temp = temp_arr[i-1, j]
        temp_arr[i, j] = temp
        temp_arr[i-1, j] = 0
        return temp_arr

def move_down(data):
    i, j = find_index(data)
    if i == 2:
        return None
    else:
        temp_arr = np.copy(data)
        temp = temp_arr[i+1, j]
        temp_arr[i, j] = temp
        temp_arr[i+1, j] = 0
        return temp_arr

def move_tile(action, data):
    if action == "up":
        return move_up(data)
    if action == "down":
        return move_down(data)
    if action == "left":
        return move_left(data)
    if action == "right":
        return move_right(data)
    else:
        return None
```

```
def print_states(list_final):
    print("printing final solution")
    for i in list_final:
        print(
            "Move : "
            + str(i.act)
            + "\n"
            + "Result : "
            + "\n"
            + str(i.data)
            + "\t"
            + "node number : "
            + str(i.node_no)
        )

def write_path(path_formed):
    if os.path.exists("Path_file.txt"):
        os.remove("Path_file.txt")

    f = open("Path_file.txt", "a")
    for node in path_formed:
        if node.parent is not None:
            f.write(
                str(node.node_no)
                + "\t"
                + str(node.parent.node_no)
                + "\t"
                + str(node.cost)
                + "\n"
            )
    f.close()
```

```
def write_node_explored(explored):
    if os.path.exists("Nodes.txt"):
        os.remove("Nodes.txt")

    f = open("Nodes.txt", "a")
    for element in explored:
        f.write("[")
        for i in range(len(element)):
            for j in range(len(element)):
                f.write(str(element[j][i])+" ")
        f.write("]")
        f.write("\n")
    f.close()

def write_node_info(visited):
    if os.path.exists("Node_info.txt"):
        os.remove("Node_info.txt")

    f = open("Node_info.txt", "a")
    for n in visited:
        if n.parent is not None:
            f.write(
                str(n.node_no)
                + "\t"
                + str(n.parent.node_no)
                + "\t"
                + str(n.cost)
                + "\n"
            )
    f.close()
```

```
def path(node):
    p = []
    p.append(node)
    parent_node = node.parent
    while parent_node is not None:
        p.append(parent_node)
        parent_node = parent_node.parent
    return list(reversed(p))

def exploring_nodes(node):
    print("Exploring Nodes")
    actions = ["down", "up", "left", "right"]
    goal_node = np.array([[1,2,3],
                          [4,5,6],
                          [7,8,0]])
    node_q = [node]
    final_nodes = []
    visited = []
    final_nodes.append(node_q[0].data.tolist())
    node_counter = 0

    while node_q:
        current_root = node_q.pop(0)
        if current_root.data.tolist() == goal_node.tolist():
            print("Goal reached")
            return current_root, final_nodes, visited
```

```
for move in actions:
    temp_data = move_tile(move, current_root.data)
    if temp_data is not None:
        node_counter += 1
        child_node = Node(
            node_counter, np.array(temp_data),
            current_root, move, 0)

        if(
            child_node.data.tolist() not in final_nodes
        ):
            node_q.append(child_node)
            final_nodes.append(child_node.data.tolist())
            visited.append(child_node)
            if child_node.data.tolist() == goal_node.tolist():
                print("Goal reached")
                return child_node, final_nodes, visited
return None, None, None

def check_correct_input(L):
    array = np.reshape(L, 9)
    for i in range(9):
        counter_appear = 0
        f = array[i]
        for j in range(9):
            if f == array[j]:
                counter_appear += 1
        if counter_appear >= 2:
            print("invalid input, Same number entered 2 times")
            exit(0)
```

```
def check_solvable(g):
    arr = np.reshape(g, 9)
    counter_states = 0
    for i in range(9):
        if not arr[i] == 0:
            check_elem = arr[i]
            for x in range(i+1, 9):
                if check_elem < arr[x] or arr[x] == 0:
                    continue
                else:
                    counter_states += 1
    if counter_states % 2 == 0:
        print("The puzzle is solvable, generating path")
    else:
        print("The puzzle is insolvable, still creating
              nodes")

k = get_initial()

check_correct_input(k)
check_solvable(k)

root = Node(0, k, None, None, 0)

goal, s, v = exploring_nodes(root)

if goal is None and s is None and v is None:
    print("Goal State could not be reached,
          Sorry")
```

```
else:
    print_states(path(goal))
    write_path(path(goal))
    write_node_explored(s)
    write_node_info(v)
```