```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Feb 24th February
ENPM 661
Project 1

@author: Rishikesh Jadhav
UID: 119256534
"""
# IMPORTING PACKAGES

from collections import deque

import numpy as np


# FUNCTION DEFINITIONS


# Define a function to print a matrix.
def print_matrix(matrix):
        row_counter = 0
        for row_index in range(0, len(matrix), 3):
        if row_counter == 0 :
        print("-------------")
        for element_index in range(row_counter, len(matrix), 3):
        if element_index <= row_counter:
                print("|", end=" ")
        print(int(matrix[element_index]), "|", end=" ")
        row_counter += 1
        print("\n-------------")

# Define a function to plot a path based on a text file.
def plot_path():
        filename = 'nodePath.txt'
        # Load the data from the text file as a NumPy array.
        data = np.loadtxt(filename)
        # If the array does not have the expected number of columns, print an error message
and exit the function.
        if len(data[1]) != 9:
        print("The format of the text file is incorrect. Please try again.")
        else:
        for i in range(len(data)):
```

```
        if i == 0:
            print("Starting node:")
        elif i == len(data) - 1:
            print("Goal node achieved:")
        else:
            print("Step", i)
        print_matrix(data[i])
        print("\n\n")
```

```
# Define a function to get the user-defined initial state.
def getInitialState():
        """
        Prompts the user to enter the start state, and checks if the entered state is valid.

        """
        flag = False
        while not flag:
        state = [int(item) for item in input("Enter the initial state: ").split()]
        temp = state.copy()    #
        valid = [0, 1, 2, 3, 4, 5, 6, 7, 8]
        temp.sort()
        if temp == valid:
        flag = True
        else:
        print("\nEnter a valid state!\n")
        return state
```

```
# Define a function to get the user-defined goal state.
def getGoalState():
        """
        Prompts the user to enter the goal state, and checks if the entered state is valid.

        """
        flag = False
        while not flag:
        state = [int(item) for item in input("Enter the goal state: ").split()]
        temp = state.copy()
        valid = [0, 1, 2, 3, 4, 5, 6, 7, 8]
        temp.sort()
        if temp == valid:
        flag = True
        else:
```

```
        print("\nEnter a valid state!\n")
        return state


def find_blank_tile_location(node):
        """
        Finds the index of the tile with number 0 in a given state.
        """
        index = node.index(0)
        row = (index % 3)
        column = (index // 3)
        return [row, column]


def move_blank_tile_up(node):
        """
        Swaps the blank tile with the adjacent tile in the up direction.

        """
        [row, column] = find_blank_tile_location(node)
        new_node = node.copy()
        if row == 0:
        status = False  # Move not valid if the blank tile is in the top row.
        else:
        status = True
        # Find the index of the blank tile and the tile above it.
        blank_index = (3 * column) + row
        up_index = (3 * column) + (row - 1)
        up_tile = node[up_index]
        # Swap the tiles.
        new_node[up_index] = 0
        new_node[blank_index] = up_tile
        return [status, new_node]

def ActionMoveUp(node):
        """
        Swaps the blank tile with the adjacent tile in the up direction

        """

        [i,j] = find_blank_tile_location(node)
        new_node = node.copy()
        if i == 0:
        status = False
```

```
        else:
        status = True
        # Position of the zero tile and the tile above are found according to the list.
        zero_index = (3 * j) + i
        up_index = (3 * j) + (i - 1)
        up_tile = node[up_index]
        # Swapping tiles
        new_node[up_index] = 0
        new_node[zero_index] = up_tile
        return [status,new_node]

def ActionMoveRight(node):
        """
        Swaps the blank tile with the adjacent tile in the right direction

        """

        [i,j] = find_blank_tile_location(node)
        new_node = node.copy()
        if j == 2:
        status = False
        else:
        status = True
        # Position of the zero tile and the tile on right are found according to the list.
        zero_index = (3 * j) + i
        right_index = (3 * (j + 1)) + i
        right_element = node[right_index]
        # Swapping tiles
        new_node[right_index] = 0
        new_node[zero_index] = right_element
        return [status,new_node]

def ActionMoveDown(node):
        """
        Swaps the blank tile with the adjacent tile in the down direction

        """

        [i,j] = find_blank_tile_location(node)
        new_node = node.copy()
        if i == 2:
        status = False
        else:
        status = True
```

```
        # Position of the zero tile and the tile below are found according to the list.
        zero_index = (3 * j) + i
        down_index = (3 * j) + (i + 1)
        down_element = node[down_index]
        # Swapping tiles
        new_node[down_index] = 0
        new_node[zero_index] = down_element
        return [status,new_node]

def ActionMoveLeft(node):
        """
        Swaps the blank tile with the adjacent tile in the left direction

        """

        [i,j] = find_blank_tile_location(node)
        new_node = node.copy()
        if j == 0:
        status = False
        else:
        status = True
        # Position of the zero tile and the tile on left are found according to the list.
        zero_ind = (3 * j) + i
        left_ind = (3 * (j - 1)) + i
        left_element = node[left_ind]
        # Swapping tiles
        new_node[left_ind] = 0
        new_node[zero_ind] = left_element
        return [status,new_node]


def generate_path(nodes):
        """
        Returns the path from the start state to the goal state.

        """
        parent = nodes[-1][9]
        path_nodes = [parent]
        while parent != -1:
        parent_node = nodes[path_nodes[-1]]
        parent = parent_node[9]
        path_nodes.append(parent)
        path = [nodes[-1][0:9]]
        for ind in path_nodes:
```

```
        if ind == -1:
        break
        else:
        path.insert(0, nodes[ind][0:9])
        return path

def AddNode(new_node, nodes, node_set):
        """
        Adds a new node to the list of nodes and set of visited nodes.

        """
        visited = False
        if tuple(new_node[0:9]) in node_set:
        visited = True
        else:
        nodes.append(new_node)
        node_set.add(tuple(new_node[0:9]))
        return [nodes, node_set, visited]


def check_neighbors(cur_node,direction):
        """
        Checks the neighbors of a node in given direction and returns a neighbor if valid.
        """
        if direction == 'Up':
        [status,new_node] = ActionMoveUp(cur_node)
        if direction == 'Right':
        [status,new_node] = ActionMoveLeft(cur_node)
        if direction == 'Down':
        [status,new_node] = ActionMoveDown(cur_node)
        if direction == 'Left':
        [status,new_node] = ActionMoveRight(cur_node)

        return [status,new_node]


def breadth_first_search(start_node, goal_node):
        """
        Applies breadth-first search to find a path from start_node to goal_node.

        """
        nodes = [start_node]
        node_set = {tuple(start_node)}
```

```
    start_node.append(-1)

    queue = deque()
    queue.append(0)  # Add the start node as the first node in the queue

    is_goal_reached = False
    search_successful = False

    neighbors = ['Up', 'Right', 'Down', 'Left']

    while queue:
    parent = queue.popleft()
    cur_node = nodes[parent]

    for direction in neighbors:
    status, new_node = check_neighbors(cur_node, direction)

    if status:
        new_node[9] = parent
        nodes, node_set, visited = AddNode(new_node, nodes, node_set)
        if not visited:
        queue.append(len(nodes) - 1)

    if new_node[0:9] == goal_node:
        is_goal_reached = True
        break

    if is_goal_reached:
    search_successful = True
    break

    return search_successful, nodes


def initialize_puzzle():
    """
    Initializes the puzzle with a user-defined start and goal state.

    """
    print("This solver considers inputs to be nine unique non-negative integers in domain:
[0,8].")
    print("A sample state is shown below:\n")
    sample = [1, 4, 7, 2, 5, 8, 3, 6, 0]
    print_matrix(sample)
```

```
    print("\nTo provide a user-defined state, enter the numbers of the state in a column-wise
fashion.")
    print("For example, for the above sample state, the input should be: 1 4 7 2 5 8 3 6 0\n")
    print("---------------------------------------")

    start_node = getInitialState()
    goal_node = getGoalState()

    return start_node, goal_node


def solve_puzzle(start_node, goal_node):
    """
    Solves the puzzle using breadth-first search.

    """
    search_successful, nodes = breadth_first_search(start_node, goal_node)
    if not search_successful:
    print("The puzzle could not be solved for the provided states.\n")
    else:
    # Generate output files
    node_path_file = open("nodePath.txt", "w+")
    nodes_file = open("Nodes.txt", "w+")
    nodes_info_file = open("NodesInfo.txt", "w+")
    path = generate_path(nodes)
    for node in path:
    for tile in node:
        node_path_file.write(str(tile) + " ")
    node_path_file.write('\n')
    node_path_file.close()

    for i,node in enumerate(nodes):
    for j,tile in enumerate(node):
    if j != 9:
        nodes_file.write(str(tile) + " ")
    nodes_file.write('\n')

    nodes_info_file.write(str(i) + " ")
    nodes_info_file.write(str(node[9]) + " ")
    nodes_info_file.write(str(0) + " ")
    nodes_info_file.write('\n')
    nodes_file.close()
    nodes_info_file.close()
```

```
plot_path()
```