Project 1

ENPM661

Vishnu Mandala
119452608

Problem Statement:

Find all the possible states of the 8-Puzzle starting from the given initial state until the goal node is reached. Note that, the states should be unique (no repetitions)

- From the initial state of the puzzle, use different moves in all the directions to generate new states, check the validity of the newly generated node
- You should use Python for programing
- Use the Breath First Search(BFS) to find the path to reach the goal
- Implement back tracking to find the plan to solve the problem

Suggested Parameter Names:

Information to be stored in the data structure for each node:

- **Node_State_i:** The state of node i is represented by a 3 by 3 matrix, for example [1 2 3; 4 5 6; 7 8 0].
- **Node_Index_i :** The index of node i
- **Parent_Node_Index_i:** The index of parent for node i.

Steps:

1. You should first select a data structure type to save your generated nodes. There are different types of data structure that could be used. The type of data structure is optional.

2. Write a function that given the state of the current node in the 8puzzle problem, calculates the location of the blank tile in the 3 by 3 matrix and returns the output as a pair, (i,j). Where $0<i<4$ and $0<j<4$. Blank tile can be located using either two for loops or by using inbuilt functions of numpy.

3. Write 4 subfunctions to move the blank tile in 4 different directions
```
[Status, NewNode] = ActionMoveLeft(CurrentNode);       % Moves blank tile left, if possible
[Status, NewNode] = ActionMoveRight(CurrentNode);      % Moves blank tile right, if possible
[Status, NewNode] = ActionMoveUp(CurrentNode);         % Moves blank tile up, if possible
[Status, NewNode] = ActionMoveDown(CurrentNode);       % Moves blank tile down, if possible
```

4. Append All the possible new nodes in a list Additional: Focus on checking the node to be explored with the visited list you maintain and if it is present do not repeat the action set for that node.

5. Write a subfunctions (generate_path) that uses back tracking to find the path between the initial node and final node.

Output textfile to be generated from the code:

Textfile1:

Nodes.txt	-	All the explored states should be present
NodesInfo.txt	-	Will contain the information about Node Index, Parent Node Index, Node First Column: Node Index Second Column: Parent Node Index Third Column: Node

Textfile 2:

nodePath.txt	-	Will contain the generated path to the goal from initial state

The elements are being stored column-wise, i.e. for this goal state 1 4 7 2 5 8 3 6 0, the eight puzzle state is:

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

The order of the states should be from start node to the goal node.

Code:

```
import numpy as np
from queue import Queue

'''Define a function to find the coordinates of the blank tile (value 0) in the given state'''
def find_blank_tile(state):
    i, j = np.where(state == 0)
    return i[0], j[0]

'''Define four functions to move the blank tile left, right, up, or down in the given state, respectively
Each function checks if the move is valid
(i.e., whether the blank tile is at the edge of the board in the given direction),
and returns a boolean flag indicating success and the resulting state after the move.'''

def ActionMoveLeft(state):
    i, j = find_blank_tile(state)
    if j == 0:
        return False, None
    else:
        new_state = np.copy(state)
        new_state[i][j] = new_state[i][j-1]
        new_state[i][j-1] = 0
        return True, new_state

def ActionMoveRight(state):
    i, j = find_blank_tile(state)
    if j == 2:
        return False, None
    else:
        new_state = np.copy(state)
        new_state[i][j] = new_state[i][j+1]
        new_state[i][j+1] = 0
        return True, new_state

def ActionMoveUp(state):
    i, j = find_blank_tile(state)
    if i == 0:
        return False, None
    else:
        new_state = np.copy(state)
        new_state[i][j] = new_state[i-1][j]
        new_state[i-1][j] = 0
        return True, new_state

def ActionMoveDown(state):
    i, j = find_blank_tile(state)
    if i == 2:
        return False, None
    else:
        new_state = np.copy(state)
        new_state[i][j] = new_state[i+1][j]
        new_state[i+1][j] = 0
        return True, new_state

'''Define a function to generate all possible states that can be obtained by moving the blank tile in the given
state in any of the four directions.
The function loops through the four move functions, applies each move to the given state, and adds the resulting
state to a list if the move was successful.'''

def generate_next_states(state):
    next_states = []
    for move in [ActionMoveLeft, ActionMoveRight, ActionMoveUp, ActionMoveDown]:
        success, new_state = move(state)
        if success:
            next_states.append(new_state)
    return next_states
```

```
'''Define a function called "bfs" that takes two arguments: the starting state of the puzzle board and the goal
state'''
def bfs(start_state, goal_state):

    # Initialize the queue and visited set
    queue = Queue()
    visited = set()

    # Add the start state to the queue with parent index -1
    start_index = 0
    parent_node_index_i = -1
    queue.put((start_index, parent_node_index_i, start_state))
    visited.add(tuple(map(tuple, start_state)))

    # Initialize the nodes list and nodes_info list
    nodes = [start_state]
    nodes_info = [(start_index, parent_node_index_i, start_state)]

    # Start BFS algorithm
    while not queue.empty():
        # Pop the first node from the queue
        node_index_i, parent_node_index_i, current_node = queue.get()

        # Generate next possible states
        next_states = generate_next_states(current_node)

        # Loop through each next state
        for next_node in next_states:
            # Check if the next state is the goal state
            if np.array_equal(current_node, goal_state):
                # Generate the path and write to nodePath.txt
                path = generate_path(nodes_info, node_index_i)
                with open("nodePath.txt", "w") as f:
                    for node in path:
                        np.savetxt(f, np.array(path).reshape(-1,9), fmt="%d", delimiter='\t')

                # Write explored states to Nodes.txt
                with open("Nodes.txt", "w") as f:
                    for node in nodes:
                        np.savetxt(f, node.reshape(-1,9), fmt="%d", delimiter='\t')

                # Write nodes_info to NodesInfo.txt
                with open("NodesInfo.txt", "w") as f:
                    for info in nodes_info:
                        f.write(f"{info[0]} {info[1]} ")
                        np.savetxt(f, info[2].reshape(-1,9), fmt="%d", delimiter=",")

                return True # Return True to indicate that a solution has been found.

            # Check if the next state has not been visited
            if tuple(map(tuple, next_node)) not in visited:
                # Add the next state to the queue and visited set
                next_index = len(nodes)
                queue.put((next_index, node_index_i, next_node))
                visited.add(tuple(map(tuple, next_node)))

                # Add the next state to nodes and nodes_info lists
                nodes.append(next_node)
                nodes_info.append((next_index, node_index_i, next_node))

    return False    # Return False to indicate that no solution has been found.

'''Define a function called "generate_path" that takes two arguments: the nodes information list and the index of
the goal state'''
def generate_path(nodes_info, goal_index):
    # Create an empty list for the path and set the current index to the index of the goal state.
    path = []
    current_index = goal_index

    # Traverse the nodes_info list backwards to generate the path
    while current_index != -1:
        node = nodes_info[current_index][2]
        path.append(node)
        current_index = nodes_info[current_index][1]
    # Reverse the path to get it in the correct order
    path.reverse()
    return path

'''Take input for the start state in a single line'''
start_state = np.zeros((3,3), dtype=int)
print("Enter the start state of the 8-puzzle board in a single line (space-separated, enter 0 for blank tile):")
input_line = input()
input_list = list(map(int, input_line.strip().split()))

for i in range(3):
    for j in range(3):
        start_state[i][j] = input_list[3*j+i]

print("Start State:")
print(start_state,'\n')

'''Take input for the goal state in a single line'''
goal_state = np.zeros((3,3), dtype=int)
print("Enter the goal state of the 8-puzzle board in a single line (space-separated, enter 0 for blank tile):")
input_line = input()
input_list = list(map(int, input_line.strip().split()))

for i in range(3):
    for j in range(3):
        goal_state[i][j] = input_list[3*j+i]

print('Goal State:')
print(goal_state,'\n')

# Transpose so that format is readable by plot_path.py
start_state = np.transpose(start_state)
goal_state = np.transpose(goal_state)

'''Run the BFS algorithm and check if it finds a solution'''
if bfs(start_state, goal_state):
    print("Solution found!\n")
else:
    print("No Solution found\n")
```

Output:

Test Case 1 –

```
Enter the start state of the 8-puzzle board in a single line (space-separated, enter 0 for blank tile):
1 6 7 2 0 5 4 3 8
Start State:
[[1 2 4]
 [6 0 3]
 [7 5 8]]

Enter the goal state of the 8-puzzle board in a single line (space-separated, enter 0 for blank tile):
1 4 7 2 5 8 3 0 6
Goal State:
[[1 2 3]
 [4 5 0]
 [7 8 6]]

Solution found!
```

Test Case 2 –

```
Enter the start state of the 8-puzzle board in a single line (space-separated, enter 0 for blank tile):
4 7 8 2 1 5 3 6 0
Start State:
[[4 2 3]
 [7 1 6]
 [8 5 0]]

Enter the goal state of the 8-puzzle board in a single line (space-separated, enter 0 for blank tile):
1 4 7 2 5 8 3 6 0
Goal State:
[[1 2 3]
 [4 5 6]
 [7 8 0]]

Solution found!
```